

Lecture 0

- Creating Code with Python
- Functions
- Bugs
- Improving Your First Python Program
 - Variables
 - Comments
 - Pseudocode
- Further Improving Your First Python Program
- Strings and Parameters
 - A small problem with quotation marks
- Formatting Strings
- More on Strings
- Integers or int
- Readability Wins
- Float Basics
- More on Floats
- Def
- Returning Values
- Summing Up

VS Code is a special type of text editor that is called a compiler. At the top, you'll notice a text editor and, at the bottom you will see a terminal where you can execute commands.

In the terminal, you can execute code `hello.py` to start coding.

In the text editor above, you can type `print("hello, world")`. This is a famous canonical program that nearly all coders write during their learning process.

In the terminal window, you can execute commands. To run this program, you are going to need to move your cursor to the bottom of

the screen, clicking in the terminal window. You can now type a second command in the terminal window. Next to the dollar sign, type `python hello.py` and press the enter key on your keyboard. Recall, computers really only understand zeros and ones. Therefore, when you run `python hello.py`, python will interpret the text that you created in `hello.py` and translate it into the zeros and ones that the computer can understand.

The result of running the `python hello.py` program is `hello, world`. Congrats! You just created your first program.

Functions are verbs or actions that the computer or computer language will already know how to perform.

In your `hello.py` program, the `print` function knows how to print to the terminal window.

The `print` function takes arguments. In this case, `"hello, world"` are the arguments that the `print` function takes.

Bugs are a natural part of coding. These are mistakes, problems for you to solve! Don't get discouraged! This is part of the process of becoming a great programmer.

Imagine in our `hello.py` program that accidentally typed `print("hello, world"` notice that we missed the final `)` required by the compiler. If I purposefully make this mistake, you'll the compiler will output an error in the terminal window!

Often, the error messages will inform you of your mistake and provide you clues on how to fix them. However, there will be many times that the compiler is not this kind.

We can personalize your first Python program.

In our text editor in `hello.py` we can add another function. `input` is a function that takes a prompt as an argument. We can edit our code

to say

```
input("What's your name? ")  
print("hello, world")
```

This edit alone, however, will not allow your program to output what your user inputs. For that, we will need to introduce you to variables

A variable is just a container for a value within your own program.

In your program, you can introduce your own variable in your program by editing it to read

```
name = input("What's your name? ")  
print("hello, world")
```

Notice that this equal = sign in the middle of `name = input("What's your name? ")` has a special role in programming. This equal sign literally assigns what is on the right to what is on the left. Therefore, the value returned by `input("What's your name? ")` is assigned to `name`.

If you edit your code as follows, you will notice an error

```
name = input("What's your name? ")  
print("hello, name")
```

The program will return `hello, name` in the terminal window regardless of what the user types.

Further editing our code, you could type

```
name = input("What's your name? ")
```

```
print("hello,")  
print(name)
```

The result in the terminal window would be

```
What's your name? David  
hello  
David
```

We are getting closer to the result we might intend!

You can learn more in Python's documentation on [data types](#).

Comments are a way for programmers to track what they are doing in their programs and even inform others about their intentions for a block of code. In short, they are notes for yourself and others that will see your code!

You can add comments to your program to be able to see what it is that your program is doing. You might edit your code as follows:

```
# Ask the user for their name  
name = input("What's your name? ")  
print("hello,")  
print(name)
```

Comments can also serve as to-do list for you.

Pseudocode is an important type of comment that becomes a special type of to-do list, especially when you don't understand how to accomplish a coding task. For example, in your code, you might edit your code to say:

```
# Ask the user for their name
name = input("What's your name? ")

# Print hello
print("hello,")

# Print the name inputted
print(name)
```

We can further edit our code as follows:

```
# Ask the user for their name
name = input("What's your name? ")

# Print hello and the inputted name
print("hello, " + name)
```

It turns out that some functions take many arguments.

We can use a comma , to pass in multiple arguments by editing our code as follows:

```
# Ask the user for their name
name = input("What's your name? ")

# Print hello and the inputted name
print("hello,", name)
```

The output in the terminal, if we typed "David" we would be hello, David. Success.

A string, known as a `str` in Python, is a sequence of text.

Rewinding a bit in our code back to the following, there was a visual

side effect of having the result appear on multiple lines:

```
# Ask the user for their name
name = input("What's your name? ")
print("hello,")
print(name)
```

Functions take arguments that influence their behavior. If we look at the documentation for [print](#) you'll notice we can learn a lot about the arguments that the print function takes.

Looking at this documentation, you'll learn that the print function automatically include a piece of code `end='\n'`. This `\n` indicates that the print function will automatically create a line break when run. The print function takes an argument called `end`` and the default is to create a new line.

However, we can technically provide an argument for end ourselves such that a new line is not created!

We can modify our code as follows:

```
# Ask the user for their name
name = input("What's your name? ")
print("hello,", end="")
print(name)
```

By providing `end=""` we are over-writing the default value of end such that it never creates a new line after this first print statement.

Providing the name as "David", the output in the terminal window will be hello, David.

Parameters, therefore, are arguments that can be taken by a function.

You can learn more in Python's documentation on [print](#).

Notice how adding quotation marks as part of your string is challenging.

`print("hello, "friend")` will not work and the compiler will throw an error.

Generally, there are two approaches to fixing this. First, you could simply change the quotes to single quote marks.

Another, more commonly used approach would be code as `print("hello, \"friend\")`. The backslashes tell the compiler that the following character should be considered a quotation mark in the string and avoid a compiler error.

Probably the most elegant way to use strings would be as follows:

```
# Ask the user for their name
name = input("What's your name? ")
print(f"hello, {name}")
```

Notice the `f` in `print(f"hello, {name}")`. This `f` is a special indicator to Python to treat this string a special way, different than previous approaches we have illustrated in this lecture. Expect that you will be using this style of strings quite frequently in this course.

You should never expect your user will cooperate as intended. Therefore, you will need to ensure that the input of your user is corrected or checked.

It turns out that built into strings is the ability to remove whitespace from a string.

By utilizing the method `strip` on `name` as `name = name.strip()`, it will strip all the whitespaces on the left and right of the users input. You

can modify your code to be:

```
# Ask the user for their name
name = input("What's your name? ")

# Remove whitespace from the str
name = name.strip()

# Print the output
print(f"hello, {name}")
```

Rerunning this program, regardless of how many spaces you type before or after the name, it will strip off all the whitespace.

Using the `title` method, it would title case the user's name:

```
# Ask the user for their name
name = input("What's your name? ")

# Remove whitespace from the str
name = name.strip()

# Capitalize the first letter of each word
name = name.title()

# Print the output
print(f"hello, {name}")
```

By this point, you might be very tired of typing python repeatedly in the terminal window. You cause us the up arrow of your keyboard to recall the most recent terminal commands you have made.

Notice that you can modify your code to be more efficient:

```
# Ask the user for their name
```



```
name = input("What's your name? ")

# Remove whitespace from the str and capitalize the first letter of each word
name = name.strip().title()

# Print the output
print(f"hello, {name}")
```

This creates the same result as your previous code.

We could even go further!

```
# Ask the user for their name, remove whitespace from the str and capitalize the first letter of each word
name = input("What's your name? ").strip().title()

# Print the output
print(f"hello, {name}")
```

You can learn more about strings in Python's documentation on [str](#)

In Python, an integer is referred to as an `int`.

In the world of mathematics, we are familiar with `+`, `-`, `*`, `/`, and `%` operators. That last operator `%` or modulo operator may not be very familiar to you.

You don't have to use the text editor window in your compiler to run Python code. Down in your terminal, you can run `python` alone. You will be presented with `>>>` in the terminal window. You can then run live, interactive code. You could type `1+1` and it will run that calculation. This mode will not commonly be used during this course. Opening up VS Code again, we can type `code calculator.py` in the terminal. This will create a new file in which we will create our own calculator.

First, we can declare a few variables.

```
x = 1
y = 2

z = x + y

print(z)
```

Naturally, when we run `python calculator.py` we get the result in the terminal window of 3. We can make this more interactive using the `input` function.

```
x = input("What's x? ")
y = input("What's y? ")

z = x + y

print(z)
```

Running this program, we discover that the output is incorrect as 12. Why might this be?

Prior, we have seen how the `+` sign concatenates two strings. Because your input from your keyboard on your computer comes into the compiler as text, it is treated a string. We, therefore, need to convert this input from a string to an integer. We can do so as follows:

```
x = input("What's x? ")
y = input("What's y? ")

z = int(x) + int(y)

print(z)
```

The result is now correct. The use of `int(x)`, is called "casting" where a value is temporarily changed from one type of variable (in this case a string) to another (here, an integer).

We can further improve our program as follows:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

print(x + y)
```

This illustrates that you can run functions on functions. The most inner function is run first, and then the outer one is run. First, the `input` function is run. Then, the `int` function.

You can learn more in Python's Documentation of [int](#).

When deciding on your approach to a coding task, remember that one could make a reasonable argument for many approaches to the same problem.

Regardless of what approach you take to a programming task, remember that your code must be readable. You should use comments to give yourself and others clues about what your code is doing. Further, you should create code in a way that is readable.

A floating point value is a real number that has a decimal point in it, such as `0.52`.

You can change your code to support floats as follows:

```
x = float(input("What's x? "))
y = float(input("What's y? "))
```

```
print(x + y)
```

This change allows your user to enter 1.2 and 3.4 to present a total of 4.6.

Let's imagine, however, that you want to round the total to the nearest integer. Looking at the Python documentation for `round` you'll see that the available arguments are `round(number[n, ndigits])`. Those square brackets indicate that something optional can be specified by the programmer. Therefore, you could do `round(n)` to round a digit to its nearest integer. Alternatively, you could code as follows:

```
# Get the user's input
x = float(input("What's x? "))
y = float(input("What's y? "))

# Create a rounded result
z = round(x + y)

# Print the result
print(z)
```

The output will be rounded to the nearest integer.

What if we wanted to format the output of long numbers? For example, rather than seeing 1000, you may wish to see 1,000. You could modify your code as follows:

```
# Get the user's input
x = float(input("What's x? "))
y = float(input("What's y? "))
```

```
# Create a rounded result
z = round(x + y)

# Print the formatted result
print(f"{z:,.}")
```

Though quite cryptic, that `print(f"{z:,.}")` creates a scenario where the outputted `z` will include commas where the result could look like `1,000` or `2,500`.

How can we round floating point values? First, modify your code as follows:

```
# Get the user's input
x = float(input("What's x? "))
y = float(input("What's y? "))

# Calculate the result
z = x / y

# Print the result
print(z)
```

When inputting 2 as `x` and 3 as `y`, the result `z` is `0.6666666666` seemingly going on to infinite as we might expect.

Let's imagine that we want to round this down, we could modify our code as follows:

```
# Get the user's input
x = float(input("What's x? "))
y = float(input("What's y? "))

# Calculate the result and round
```

```
z = round(x / y, 2)
```

```
# Print the result  
print(z)
```

As we might expect, this will round the result to the nearest two decimal points.

We could also use `fstring` to format the output as follows:

```
# Get the user's input  
x = float(input("What's x? "))  
y = float(input("What's y? "))
```

```
# Calculate the result  
z = x / y
```

```
# Print the result  
print(f"{z:.2f}")
```

This cryptic `fstring` code displays the same as our prior rounding strategy.

You can learn more in Python's documentation of [float](#).

Wouldn't it be nice to create our own functions?

Let's bring back our final code of `hello.py` by typing code `hello.py` into the terminal window. Your starting code should look as follows:

```
# Ask the user for their name, remove whitespace from the str and capi  
name = input("What's your name? ").strip().title()
```

```
# Print the output
```

```
print(f"hello, {name}")
```

We can better our code to create our own special function that says "hello" for us!

Erasing all our code in our text editor, let's start from scratch:

```
name = input("What's your name? ")  
hello()  
print(name)
```

Attempting to run this code, your compiler will throw an error. After all, there is no defined function for `hello`.

We can create our own function called `hello` as follows:

```
def hello():  
    print("hello")
```

```
name = input("What's your name? ")  
hello()  
print(name)
```

Notice that everything under `def hello()` is indented. Python is an indented language. It uses indentation to understand what is part of the above function. Therefore, everything in the `hello` function must be indented. When something is not indented, it treats it as if it is not inside the `hello` function. Running `python hello.py` in the terminal window, you'll see that your output is not exactly as you may want.

We can further improve our code:

```
# Create our own function
def hello(to):
    print("hello,", to)

# Output using our own function
name = input("What's your name? ")
hello(name)
```

Here, in the first lines, you are creating your `hello` function. This time, however, you are telling the compiler that this function takes a single parameter: a variable called `to`. Therefore, when you call `hello(name)` the computer passes `name` into the `hello` function as `to`. This is how we pass values into functions. Very useful! Running `python hello.py` in the terminal window, you'll see that the output is much closer to our ideal presented earlier in this lecture.

We can change our code to add a default value to `hello`:

```
# Create our own function
def hello(to="world"):
    print("hello,", to)

# Output using our own function
name = input("What's your name? ")
hello(name)

# Output without passing the expected arguments
hello()
```

Test out your code yourself. Notice how the first `hello` will behave as you might expect and the second `hello`, which is not passed a value,

will by default output hello, world.

We don't have to have our function at the start of our program. We can move it down, but we need to tell the compiler that we have a main function and we have a separate hello function.

```
def main():

    # Output using our own function
    name = input("What's your name? ")
    hello(name)

    # Output without passing the expected arguments
    hello()

# Create our own function
def hello(to="world"):
    print("hello,", to)
```

This alone, however, will create an error of sorts. If we run python hello.py nothing happens! The reason for this is that nothing in this code is actually calling the main function and bringing our program to life.

The following very small modification will call the main function and restore our program to working order:

```
def main():

    # Output using our own function
    name = input("What's your name? ")
    hello(name)
```

```
# Output without passing the expected arguments
hello()

# Create our own function
def hello(to="world"):
    print("hello,", to)

main()
```

You can imagine many scenarios where you don't just want a function to perform an action, but also to return a value back to the main function. For example, rather than simply printing the calculation of $x + y$, you may want a function to return the value of this calculation back to another part of your program. This "passing back" of a value we call a return value.

Returning to our `calculator.py` code by typing `calculator.py`. Erase all code there. Rework the code as follows:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n * n

main()
```

Effectively, x is passed to `square`. Then, the calculation of $x * x$ is returned back to the main function.

Through the work of this single lecture, you have learned abilities that you will use countless times in your own programs. You have learned about...

Creating your first programs in Python;

Functions;

Bugs;

Variables;

Comments;

Pseudocode;

Strings;

Parameters;

Formatted Strings;

Integers;

Principles of readability;

Floats;

Creating your own functions; and

Return values.

Lecture 4

- Libraries
- Random
- Statistics
- Command-Line Arguments
- `slice`
- Packages
- APIs
- Making Your Own Libraries
- Summing Up

Generally, libraries are bits of code written by you or others can you can use in your program.

Python allows you to share functions or features with others as “modules”.

If you copy and paste code from an old project, chances are you can create such a module or library that you could bring into your new project.

`random` is a library that comes with Python that you could import into your own project.

It's easier as a coder to stand on the shoulders of prior coders.

So, how do you load a module into your own program? You can use the word `import` in your program.

Inside the `random` module, there is a built-in function called `random.choice(seq)`. `random` is the module you are importing. Inside that module, there is the `choice` function. That function takes into it a `seq` or sequence that is a list.

In your terminal window type `code generate.py`. In your text editor,

code as follows:

```
import random

coin = random.choice(["heads", "tails"])
print(coin)
```

Notice that the list within choice has square braces, quotes, and a comma. Since you have passed in two items, Python does the math and gives a 50% chance for heads and tails. Running your code, you will notice that this code, indeed, does function well!

We can improve our code. `from` allows us to be very specific about what we'd like to import. Prior, our `import` line of code is bringing the entire contents of the functions of `random`. However, what if we want to only load a small part of a module? Modify your code as follows:

```
from random import choice

coin = choice(["heads", "tails"])
print(coin)
```

Notice that we now can import just the `choice` function of `random`. From that point forward, we no longer need to code `random.choice`. We can now only code `choice` alone. `choice` is loaded explicitly into our program. This saves system resources and potentially can make our code run faster!

Moving on, consider the function `random.randint(a, b)`. This function will generate a random number between `a` and `b`. Modify your code as follows:

```
import random

number = random.randint(1, 10)
print(number)
```

Notice that our code will randomly generate a number between 1 and 10.

We can introduce into our card `random.shuffle(x)` where it will shuffle a list into a random order.

```
import random

cards = ["jack", "queen", "king"]
random.shuffle(cards)
for card in cards:
    print(card)
```

Notice that `random.shuffle` will shuffle the cards in place. Unlike other functions, it will not return a value. Instead, it will take the `cards` list and shuffle them inside that list. Run your code a few times to see the code functioning.

We now have these three ways above to generate random information.

You can learn more in Python's documentation of [random](#).

Python comes with a built-in `statistics` library. How might we use this module?

`average` is a function of this library that is quite useful. In your terminal window, type `code average.py`. In the text editor window,

modify your code as follows:

```
import statistics

print(statistics.mean([100, 90]))
```

Notice that we imported a different library called `statistics`. The `mean` function takes a list of values. This will print the average of these values. In your terminal window, type `python average.py`.

Consider the possibilities of using the `statistics` module in your own programs.

You can learn more in Python's documentation of [statistics](#).

So far, we have been providing all values within the program that we have created. What if we wanted to be able to take input from the command-line? For example, rather than typing `python average.py` in the terminal, what if we wanted to be able to type `python average.py 100 90` and be able to get the average between 100 and 90?

`sys` is a module that allows us to take arguments at the command line.

`argv` is a function within the `sys` module that allows us to learn about what the user typed in at the command line. Notice how you will see `sys.argv` utilized in the code below. In the terminal window, type `code name.py`. In the text editor, code as follows:

```
import sys

print("hello, my name is", sys.argv[1])
```

Notice that the program is going to look at what the user typed in the command line. Currently, if you type `python name.py David` into the terminal window, you will see `hello, my name is David`. Notice that `sys.argv[1]` is where `David` is being stored. Why is that? Well, in prior lessons, you might remember that lists start at the 0th element. What do you think is held currently in `sys.argv[0]`? If you guessed `name.py`, you would be correct!

There is a small problem with our program as it stands. What if the user does not type in the name at the command line? Try it yourself. Type `python name.py` into the terminal window. An error `list index out of range` will be presented by the compiler. The reason for this is that there is nothing at `sys.argv[1]` because nothing was typed! Here's how we can protect our program from this type of error:

```
import sys

try:
    print("hello, my name is", sys.argv[1])
except IndexError:
    print("Too few arguments")
```

Notice that the user will now be prompted with a useful hint about how to make the program work if they forget to type in a name. However, could we be more defensive to ensure the user inputs the right values?

Our program can be improved as follows:

```
import sys

if len(sys.argv) < 2:
```



```
    print("Too few arguments")
elif len(sys.argv) > 2:
    print("Too many arguments")
else:
    print("hello, my name is", sys.argv[1])
```

Notice how if you test your code, you will see how these exceptions are handled, providing the user with more refined advice. Even if the user types in too many or too few arguments, the user is provided clear instructions about how to fix the issue.

Right now, our code is logically correct. However, there is something very nice about keeping our error checking separate from the remainder of our code. How could we separate out our error handling? Modify your code as follows:

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")
elif len(sys.argv) > 2:
    sys.exit("Too many arguments")

print("hello, my name is", sys.argv[1])
```

Notice how we are using a built-in function of `sys` called `exit` that allows us to exit the program if an error was introduced by the user. We can rest assured now that the program will never execute the final line of code and trigger an error. Therefore, `sys.argv` provides a way by which users can introduce information from the command line. `sys.exit` provides a means by which the program can exit if an error arises.

You can learn more in Python's documentation of [sys](#).

slice

`slice` is a command that allows us to take a `list` and tell the compiler where we want the compiler to consider the start of the `list` and the end of the `list`. For example, modify your code as follows:

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")

for arg in sys.argv:
    print("hello, my name is", arg)
```

Notice that if you type `python name.py David Carter Rongxin` into the terminal window, the compiler will output not just the intended output of the names, but also `hello, my name is name.py`. How then could we ensure that the compiler ignores the first element of the list where `name.py` is currently being stored?

`slice` can be employed in our code to start the list somewhere different! Modify your code as follows:

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")

for arg in sys.argv[1:]:
    print("hello, my name is", arg)
```

Notice that rather than starting the list at 0, we use square brackets to tell the compiler to start at 1 and go to the end using the 1: argument. Running this code, you'll notice that we can improve our code using relatively simple syntax.

One of the reasons Python is so popular is that there are numerous powerful third-party libraries that add functionality. We call these third-party libraries, implemented as a folder, "packages".

PyPI is a repository or directory of all available third-party packages currently available.

cowsay is a well-known package that allows a cow to talk to the user.

Python has a package manager called `pip` that allows you to install packages quickly onto your system.

In the terminal window, you can install the cowsay package by typing `pip install cowsay`. After a bit of output, you can now go about using this package in your code.

In your terminal window type `code say.py`. In the text editor, code as follows:

```
import cowsay
import sys

if len(sys.argv) == 2:
    cowsay.cow("hello, " + sys.argv[1])
```

Notice that the program first checks that the user inputted at least two arguments at the command line. Then, the cow should speak to the user. Type `python say.py David` and you'll see a cow saying "hello" to David.

Further modify your code:

```
import cowsay
import sys

if len(sys.argv) == 2:
    cowsay.trex("hello, " + sys.argv[1])
```

Notice that a t-rex is now saying “hello”.

You now can see how you could install third-party packages.

You can learn more on PyPI’s entry for [cowsay](#).

You can find other third-party packages at [PyPI](#).

APIs or “application program interfaces” allow you to connect to the code of others.

`requests` is a package that allows your program to behave as a web browser would.

In your terminal, type `pip install requests`. Then, type code `itunes.py`.

It turns out that Apple iTunes has its own API that you can access in your programs. In your internet browser, you can visit

<https://itunes.apple.com/search?entity=song&limit=1&term=weezer>

and a text file will be downloaded. David constructed this URL by reading Apple’s API documentation. Notice how this query is looking for a song, with a `limit` of one result, that relates to the term called `weezer`. Looking at this text file that is downloaded, you might find the format to be similar to that we’ve programmed previously in Python.

The format in the downloaded text file is called JSON, a text-based format that is used to exchange text-based data between applications. Literally, Apple is providing a JSON file that we could interpret in our own Python program.

In the terminal window, type code `itunes.py`. Code as follows:

```
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?entity=song&l")
print(response.json())
```

Notice how the returned value of `requests.get` will be stored in `response`. David, having read the Apple documentation about this API, knows that what is returned is a JSON file. Running `python itunes.py weezer`, you will see the JSON file returned by Apple. However, the JSON response is converted by Python into a dictionary. Looking at the output, it can be quite dizzying!

It turns out that Python has a built-in JSON library that can help us interpret the data received. Modify your code as follows:

```
import json
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?entity=song&l")
print(json.dumps(response.json(), indent=2))
```

Notice that `json.dumps` is implemented such that it utilizes `indent` to make the output more readable. Running `python itunes.py weezer`, you will see the same JSON file. However, this time, it is much more readable. Notice now that you will see a dictionary called `results`

inside the output. Inside that dictionary called `results` there are numerous keys present. Look at the `trackName` value in the output. What track name do you see in your results?

How could we simply output the name of just that track name? Modify your code as follows:

```
import json
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?entity=song&l

o = response.json()
for result in o["results"]:
    print(result["trackName"])
```

Notice how we are taking the result of `response.json()` and storing it in `o` (as in the lowercase letter). Then, we are iterating through the `results` in `o` and printing each `trackName`. Also notice how we have increased the limit number of results to 50. Run your program. See the results.

You can learn more about `requests` through the [library's documentation](#).

You can learn more about JSON in Python's documentation of [JSON](#).

You have the ability as a Python programmer to create your own library!

Imagine situations where you may want to re-use bits of code time

and time again or even share them with others!

We have been writing lots of code to say "hello" so far in this course. Let's create a package to allow us to say "hello" and "goodbye". In your terminal window, type `code sayings.py`. In the text editor, code as follows:

```
def hello(name):  
    print(f"hello, {name}")  
  
def goodbye(name):  
    print(f"goodbye, {name}")
```

Notice that this code in and of itself does not do anything for the user. However, if a programmer were to import this package into their own program, the abilities created by the functions above could be implemented in their code.

Let's see how we could implement code utilizing this package that we created. In the terminal window, type `code say.py`. In this new file in your text editor, type the following:

```
import sys  
  
from saying import goodbye  
  
if len(sys.argv) == 2:  
    goodbye(sys.argv[1])
```

Notice that this code imports the abilities of `goodbye` from the `saying` package. If the user inputted at least two arguments at the command line, it will say "goodbye" along with the string inputted at

the command line.

Libraries extend the abilities of Python. Some libraries are included by default with Python and simply need to be imported. Others are third-party packages that need to be installed using `pip`. You can make your own packages for use by yourself or others! In this lecture, you learned about...

Libraries

Random

Statistics

Command-Line Arguments

Slice

Packages

APIs

Making Your Own Libraries

Lecture 8

- Object-Oriented Programming
- Classes
- `raise`
- Decorators
- Connecting to Previous Work in this Course
- Class Methods
- Static Methods
- Inheritance
- Inheritance and Exceptions
- Operator Overloading
- Summing Up

There are different paradigms of programming. As you learn other languages, you will start recognizing patterns like these.

Up until this point, you have worked procedurally step-by-step.

Object-oriented programming (OOP) is a compelling solution to programming-related problems.

To begin, type code `student.py` in the terminal window and code as follows:

```
name = input("Name: ")
house = input("House: ")
print(f"{name} from {house}")
```

Notice that this program follows a procedural, step-by-step paradigm: Much like you have seen in prior parts of this course.

Drawing on our work from previous weeks, we can create functions to abstract away parts of this program.

```
def main():
```

```

name = get_name()
house = get_house()
print(f"{name} from {house}")

def get_name():
    return input("Name: ")

def get_house():
    return input("House: ")

if __name__ == "__main__":
    main()

```

Notice how `get_name` and `get_house` abstract away some of the needs of our `main` function. Further, notice how the final lines of the code above tell the compiler to run the `main` function.

We can further simplify our program by storing the student as a tuple. A tuple is a sequences of values. Unlike a `list`, a tuple can't be modified. In spirit, we are returning two values.

```

def main():
    name, house = get_student()
    print(f"{name} from {house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return name, house

if __name__ == "__main__":

```

```
main()
```

Notice how `get_student` returns `name`, `house`.

Packing that `tuple`, such that we are able to return both items to a variable called `student`, we can modify our code as follows.

```
def main():
    student = get_student()
    print(f"{student[0]} from {student[1]}")
```

```
def get_student():
    name = input("Name: ")
    house = input("House: ")
    return (name, house)
```

```
if __name__ == "__main__":
    main()
```

Notice that `(name, house)` explicitly tells anyone reading our code that we are returning two values within one. Further, notice how we can index into `tuples` using `student[0]` or `student[1]`.

`tuples` are `immutable`, meaning we cannot change those values. `Immutability` is a way by which we can program defensively.

```
def main():
    student = get_student()
    if student[0] == "Padma":
        student[1] = "Ravenclaw"
    print(f"{student[0]} from {student[1]}")
```

```
def get_student():
```

```

    name = input("Name: ")
    house = input("House: ")
    return name, house

if __name__ == "__main__":
    main()

```

Notice that this code produces an error. Since tuples are immutable, we're not able to reassign the value of `student[1]`.

If we wanted to provide our fellow programmers flexibility, we could utilize a list as follows.

```

def main():
    student = get_student()
    if student[0] == "Padma":
        student[1] = "Ravenclaw"
    print(f"{student[0]} from {student[1]}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return [name, house]

if __name__ == "__main__":
    main()

```

Note the lists are mutable. That is, the order of house and name can be switched by a programmer. You might decide to utilize this in some cases where you want to provide more flexibility at the cost of the security of your code. After all, if the order of those values is changeable, programmers that work with you could make mistakes

down the road.

A dictionary could also be utilized in this implementation. Recall that dictionaries provide a key-value pair.

```
def main():
    student = get_student()
    print(f"{student['name']} from {student['house']}")

def get_student():
    student = {}
    student["name"] = input("Name: ")
    student["house"] = input("House: ")
    return student

if __name__ == "__main__":
    main()
```

Notice in this case, two key-value pairs are returned. An advantage of this approach is that we can index into this dictionary using the keys.

Still, our code can be further improved. Notice that there is an unneeded variable. We can remove `student = {}` because we don't need to create an empty dictionary.

```
def main():
    student = get_student()
    print(f"{student['name']} from {student['house']}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
```

```
    return {"name": name, "house": house}
```

```
if __name__ == "__main__":  
    main()
```

Notice we can utilize {} braces in the return statement to create the dictionary and return it all in the same line.

We can provide our special case with Padma in our dictionary version of our code.

```
def main():  
    student = get_student()  
    if student["name"] == "Padma":  
        student["house"] = "Ravenclaw"  
    print(f"{student['name']} from {student['house']}")
```

```
def get_student():  
    name = input("Name: ")  
    house = input("House: ")  
    return {"name": name, "house": house}
```

```
if __name__ == "__main__":  
    main()
```

Notice how, similar in spirit to our previous iterations of this code, we can utilize the key names to index into our student dictionary.

Classes are a way by which, in object-oriented programming, we can create our own type of data and give them names.

A class is like a mold for a type of data – where we can invent our own data type and give them a name.

We can modify our code as follows to implement our own class called Student:

```
class Student:
    ...

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    student = Student()
    student.name = input("Name: ")
    student.house = input("House: ")
    return student

if __name__ == "__main__":
    main()
```

Notice by convention that Student is capitalized. Further, notice the ... simply means that we will later return to finish that portion of our code. Further, notice that in get_student, we can create a student of class Student using the syntax student = Student(). Further, notice that we utilize "dot notation" to access attributes of this variable student of class Student.

Any time you create a class and you utilize that blueprint to create something, you create what is called an "object" or an "instance". In the case of our code, student is an object.

Further, we can lay some groundwork for the attributes that are expected inside an object whose class is Student. We can modify our

code as follows:

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    student = Student(name, house)
    return student

if __name__ == "__main__":
    main()
```

Notice that within `Student`, we standardize the attributes of this class. We can create a function within `class Student`, called a “method”, that determines the behavior of an object of class `Student`. Within this function, it takes the `name` and `house` passed to it and assigns these variables to this object. Further, notice how the constructor `student = Student(name, house)` calls this function within the `Student` class and creates a `student`. `self` refers to the current object that was just created.

We can simplify our code as follows:

```
class Student:
```



```

def __init__(self, name, house):
    self.name = name
    self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()

```

Notice how `return Student(name, house)` simplifies the previous iteration of our code where the constructor statement was run on its own line.

You can learn more in Python's documentation of [classes](#).

raise

Object-oriented program encourages you to encapsulate all the functionality of a class within the class definition. What if something goes wrong? What if someone tries to type in something random? What if someone tries to create a student without a name? Modify your code as follows:

```

class Student:
    def __init__(self, name, house):
        if not name:

```

```

        raise ValueError("Missing name")
    if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
"Slytherin"]:
        raise ValueError("Invalid house")
    self.name = name
    self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()

```

Notice how we check now that a name is provided and a proper house is designated. It turns out we can create our own exceptions that alerts the programmer to a potential error created by the user called `raise`. In the case above, we raise `ValueError` with a specific error message.

It just so happens that Python allows you to create a specific function by which you can print the attributes of an object. Modify your code as follows:

```

class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")

```

```

        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
"Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patronus: ")
    return Student(name, house, patronus)

if __name__ == "__main__":
    main()

```

Notice how `def __str__(self)` provides a means by which a student is returned when called. Therefore, you can now, as the programmer, print an object, its attributes, or almost anything you desire related to that object.

`__str__` is a built-in method that comes with Python classes. It just so happens that we can create our own methods for a class as well! Modify your code as follows:

```

class Student:

```

```

def __init__(self, name, house, patronus=None):
    if not name:
        raise ValueError("Missing name")
    if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
"Slytherin"]:
        raise ValueError("Invalid house")
    if patronus and patronus not in ["Stag", "Otter", "Jack
Russell terrier"]:
        raise ValueError("Invalid patronus")
    self.name = name
    self.house = house
    self.patronus = patronus




```

```

def __str__(self):
    return f"{self.name} from {self.house}"

```

```

def charm(self):
    match self.patronus:
        case "Stag":
            
            return " "
        case "Otter":
            
            return " "
        case "Jack Russell terrier":
            
            return " "
        case _:
            
            return " "

```

```

def main():
    student = get_student()
    print("Expecto Patronum!")
    print(student.charm())

def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patronus: ") or None
    return Student(name, house, patronus)

if __name__ == "__main__":
    main()

```

Notice how we define our own method `charm`. Unlike dictionaries, classes can have built-in functions called methods. In this case, we define our `charm` method where specific cases have specific results. Further, notice that Python has the ability to utilize emojis directly in our code.

Before moving forward, let us remove our `patronus` code. Modify your code as follows:

```

class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Invalid name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
"Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house

    def __str__(self):

```

```
    return f"{self.name} from {self.house}"
```

```
def main():  
    student = get_student()  
    student.house = "Number Four, Privet Drive"  
    print(student)
```

```
def get_student():  
    name = input("Name: ")  
    house = input("House: ")  
    return Student(name, house)
```

```
if __name__ == "__main__":  
    main()
```

Notice how we have only two methods: `__init__` and `__str__`.

Properties can be utilized to harden our code. In Python, we define properties using function “decorators”, which begin with `@`. Modify your code as follows:

```
class Student:  
    def __init__(self, name, house):  
        if not name:  
            raise ValueError("Invalid name")  
        self.name = name  
        self.house = house  
  
    def __str__(self):  
        return f"{self.name} from {self.house}"  
  
    # Getter for house  
    @property
```

```

def house(self):
    return self._house

# Setter for house
@house.setter
def house(self, house):
    if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
        raise ValueError("Invalid house")
    self._house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()

```

Notice how we've written `@property` above a function called `house`. Doing so defines `house` as a property of our class. With `house` as a property, we gain the ability to define how some attribute of our class, `_house`, should be set and retrieved. Indeed, we can now define a function called a "setter", via `@house.setter`, which will be called whenever the `house` property is set—for example, with `student.house = "Gryffindor"`. Here, we've made our setter validate values of `house` for us. Notice how we raise a `ValueError` if the value of `house` is not any of the Harry Potter houses, otherwise, we'll use `house` to update the value of `_house`. Why `_house` and not

house? house is a property of our class, with functions via which a user attempts to set our class attribute. `_house` is that class attribute itself. The leading underscore, `_`, indicates to users they need not (and indeed, shouldn't!) modify this value directly. `_house` should *only* be set through the house setter. Notice how the house property simply returns that value of `_house`, our class attribute that has presumably been validated using our house setter. When a user calls `student.house`, they're getting the value of `_house` through our house "getter".

In addition to the name of the house, we can protect the name of our student as well. Modify your code as follows:

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    # Getter for name
    @property
    def name(self):
        return self._name

    # Setter for name
    @name.setter
    def name(self, name):
        if not name:
            raise ValueError("Invalid name")
        self._name = name

    @property
    def house(self):
```



```

        return self._house

    @house.setter
    def house(self, house):
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self._house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()

```

Notice how, much like the previous code, we provide a getter and setter for the name.

You can learn more in Python's documentation of [methods](#).

While not explicitly stated in past portions of this course, you have been using classes and objects the whole way through.

If you dig into the documentation of `int`, you'll see that it is a class with a constructor. It's a blueprint for creating objects of type `int`.

You can learn more in Python's documentation of [int](#).

Strings too are also a class. If you have used `str.lower()`, you were using a method that came within the `str` class. You can learn more in

Python's documentation of [str](#).

`list` is also a class. Looking at that documentation for `list`, you can see the methods that are contained therein, like `list.append()`. You can learn more in Python's documentation of [list](#).

`dict` is also a class within Python. You can learn more in Python's documentation of [dict](#).

To see how you have been using classes all along, go to your console and type `code type.py` and then code as follows:

Notice how by executing this code, it will display that the class of `50` is `int`.

We can also apply this to `str` as follows:

```
print(type("hello, world"))
```

Notice how executing this code will indicate this is of the class `str`.

We can also apply this to `list` as follows:

Notice how executing this code will indicate this is of the class `list`.

We can also apply this to a `list` using the name of Python's built-in `list` class as follows:

Notice how executing this code will indicate this is of the class `list`.

We can also apply this to `dict` as follows:

Notice how executing this code will indicate this is of the class `dict`.

We can also apply this to a `dict` using the name of Python's built in `dict` class as follows:

Notice how executing this code will indicate this is of the class `dict`.

Sometimes, we want to add functionality to a class itself, not to instances of that class.

`@classmethod` is a function that we can use to add functionality to a class as a whole.

Here's an example of *not* using a class method. In your terminal window, type `code` `hat.py` and code as follows:

```
import random

class Hat:
    def __init__(self):
        self.houses = ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slyth

    def sort(self, name):
        print(name, "is in", random.choice(self.houses))

hat = Hat()
hat.sort("Harry")
```

Notice how when we pass the name of the student to the sorting hat, it will tell us what house is assigned to the student. Notice that `hat = Hat()` instantiates a hat. The sort functionality is always handled by the *instance* of the class `Hat`. By executing `hat.sort("Harry")`, we pass the name of the student to the sort method of the particular instance of `Hat`, which we've called `hat`.

We may want, though, to run the sort function without creating a particular instance of the sorting hat (there's only one, after all!). We can modify our code as follows:

```

import random

class Hat:

    houses = ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]

    @classmethod
    def sort(cls, name):
        print(name, "is in", random.choice(cls.houses))

Hat.sort("Harry")

```

Notice how the `__init__` method is removed because we don't need to instantiate a hat anywhere in our code. `self`, therefore, is no longer relevant and is removed. We specify this sort as a `@classmethod`, replacing `self` with `cls`. Finally, notice how `Hat` is capitalized by convention near the end of this code, because this is the name of our class.

Returning back to `students.py` we can modify our code as follows, addressing some missed opportunities related to `@classmethods`:

```

class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    @classmethod
    def get(cls):
        name = input("Name: ")

```

```
house = input("House: ")
return cls(name, house)
```

```
def main():
    student = Student.get()
    print(student)
```

```
if __name__ == "__main__":
    main()
```

Notice that `get_student` is removed and a `@classmethod` called `get` is created. This method can now be called without having to create a student first.

It turns out that besides `@classmethods`, which are distinct from instance methods, there are other types of methods as well. Using `@staticmethod` may be something you might wish to explore. While not covered explicitly in this course, you are welcome to go and learn more about static methods and their distinction from class methods.

Inheritance is, perhaps, the most powerful feature of object-oriented programming.

It just so happens that you can create a class that “inherits” methods, variables, and attributes from another class.

In the terminal, execute code `wizard.py`. Code as follows:

```
class Wizard:
    def __init__(self, name):
        if not name:
            raise ValueError("Missing name")
        self.name = name
```

```

...

class Student(Wizard):
    def __init__(self, name, house):
        super().__init__(name)
        self.house = house

...

class Professor(Wizard):
    def __init__(self, name, subject):
        super().__init__(name)
        self.subject = subject

...

wizard = Wizard("Albus")
student = Student("Harry", "Gryffindor")
professor = Professor("Severus", "Defense Against the Dark Arts")
...

```

Notice that there is a class above called `Wizard` and a class called `Student`. Further, notice that there is a class called `Professor`. Both students and professors have names. Also, both students and professors are wizards. Therefore, both `Student` and `Professor` inherit the characteristics of `Wizard`. Within the "child" class `Student`, `Student` can inherit from the "parent" or "super" class `Wizard` as the line `super().__init__(name)` runs the `init` method of `Wizard`. Finally, notice that the last lines of this code create a wizard called `Albus`, a student called `Harry`, and so on.

While we have just introduced inheritance, we have been using this

all along during our use of exceptions.

It just so happens that exceptions come in a hierarchy, where there are children, parent, and grandparent classes. These are illustrated below:

BaseException

```
+-- KeyboardInterrupt
+-- Exception
    +-- ArithmeticError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- KeyError
    +-- NameError
    +-- SyntaxError
    |   +-- IndentationError
    +-- ValueError
...

```

You can learn more in Python's documentation of [exceptions](#).

Some operators such as + and – can be “overloaded” such that they can have more abilities beyond simple arithmetic.

In your terminal window, type code vault.py. Then, code as follows:

```
class Vault:
    def __init__(self, galleons=0, sickles=0, knuts=0):
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f"{self.galleons} Galleons, {self.sickles} Sickles,
```

```

{self.knuts} Knuts"

    def __add__(self, other):
        galleons = self.galleons + other.galleons
        sickles = self.sickles + other.sickles
        knuts = self.knuts + other.knuts
        return Vault(galleons, sickles, knuts)

potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)

total = potter + weasley
print(total)

```

Notice how the `__str__` method returns a formatted string. Further, notice how the `__add__` method allows for the addition of the values of two vaults. `self` is what is on the left of the `+` operand. `other` is what is right of the `+`.

You can learn more in Python's documentation of [operator overloading](#).

Now, you've learned a whole new level of capability through object-oriented programming.

Object-oriented programming

Classes

raise

Class Methods

Static Methods

Inheritance

Operator Overloading

Lecture 3

- Exceptions
- Runtime Errors
- try
- else
- Creating a Function to Get an Integer
- pass
- Summing Up

Exceptions are things that go wrong within our coding.

In our text editor, type code `hello.py` to create a new file. Type as follows (with the intentional errors included):

Notice that we intentionally left out a quotation mark.

Running `python hello.py` in our terminal window, an error is outputted. The compiler states that it is a "syntax error." Syntax errors are those that require you to double-check that you typed in your code correction.

You can learn more in Python's documentation of [Errors and Exceptions](#).

Runtime errors refer to those created by unexpected behavior within your code. For example, perhaps you intended for a user to input a number, but they input a character instead. Your program may throw an error because of this unexpected input from the user.

In your terminal window, run code `number.py`. Code as follows in your text editor:

```
x = int(input("What's x? "))
print(f"x is {x}")
```

Notice that by including the `f`, we tell Python to interpolate what is in the curly braces as the value of `x`. Further, testing out your code, you can imagine how one could easily type in a string or a character instead of a number. Even still, a user could type nothing at all – simply hitting the enter key.

As programmers, we should be defensive to ensure that our users are entering what we expected. We might consider “corner cases” such as `-1`, `0`, or `cat`.

If we run this program and type in “cat”, we’ll suddenly see `ValueError: invalid literal for int() with base 10: 'cat'`. Essentially, the Python interpreter does not like that we passed “cat” to the `print` function.

An effective strategy to fix this potential error would be to create “error handling” to ensure the user behaves as we intend.

You can learn more in Python’s documentation of [Errors and Exceptions](#).

try

In Python `try` and `except` are ways of testing out user input before something goes wrong. Modify your code as follows:

```
try:
    x = int(input("What's x?"))
    print(f"x is {x}")
except ValueError:
```

```
print("x is not an integer")
```

Notice how, running this code, inputting 50 will be accepted. However, typing in cat will produce an error visible to the user, instructing them why their input was not accepted.

This is still not the best way to implement this code. Notice that we are trying to do two lines of code. For best practice, we should only try the fewest lines of code possible that we are concerned could fail. Adjust your code as follows:

```
try:
    x = int(input("What's x?"))
except ValueError:
    print("x is not an integer")

print(f"x is {x}")
```

Notice that while this accomplishes our goal of trying as few lines as possible, we now face a new error! We face a `NameError` where `x` is not defined. Look at this code and consider: Why is `x` not defined in some cases?

Indeed, if you examine the order of operations in `x = int(input("What's x?"))`, working right to left, it could take an incorrectly inputted character and attempt to assign it as an integer. If this fails, the assignment of the value of `x` never occurs. Therefore, there is no `x` to print on our final line of code.

else

It turns out that there is another way to implement `try` that could

catch errors of this nature.

Adjust your code as follows:

```
try:
    x = int(input("What's x?"))
except ValueError:
    print("x is not an integer")
else:
    print(f"x is {x}")
```

Notice that if no exception occurs, it will then run the block of code within `else`. Running `python number.py` and supplying `50`, you'll notice that the result will be printed. Trying again, this time supplying `cat`, you'll notice that the program now catches the error.

Considering improving our code, notice that we are being a bit rude to our user. If our user does not cooperate, we currently simply end our program. Consider how we can use a loop to prompt the user for `x` and if they don't prompt again! Improve your code as follows:

```
while True:
    try:
        x = int(input("What's x?"))
    except ValueError:
        print("x is not an integer")
    else:
        break

print(f"x is {x}")
```

Notice that `while True` will loop forever. If the user succeeds in supplying the correct input, we can break from the loop and then

print the output. Now, a user that inputs something incorrectly will be asked for input again.

Surely, there are many times that we would want to get an integer from our user. Modify your code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")

def get_int():
    while True:
        try:
            x = int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
        else:
            break
    return x

main()
```

Notice that we are manifesting many great properties. First, we have abstracted away the ability to get an integer. Now, this whole program boils down to the first three lines of the program.

Even still, we can improve this program. Consider what else you could do to improve this program. Modify your code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")
```

```
def get_int():
    while True:
        try:
            x = int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
        else:
            return x
```

```
main()
```

Notice that return will not only break you out of a loop, but it will also return a value.

Some people may argue you could do the following:

```
def main():
    x = get_int()
    print(f"x is {x}")
```

```
def get_int():
    while True:
        try:
            return int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
```

```
main()
```

Notice this does the same thing as the previous iteration of our code, simply with fewer lines.

pass

We can make it such that our code does not warn our user, but simply re-asks them our prompting question by modifying our code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")

def get_int():
    while True:
        try:
            return int(input("What's x?"))
        except ValueError:
            pass

main()
```

Notice that our code will still function but will not repeatedly inform the user of their error. In some cases, you'll want to be very clear to the user what error is being produced. Other times, you might decide that you simply want to ask them for input again.

One final refinement that could improve the implementation of this `get_int` function. Right now, notice that we are relying currently upon the honor system that the `x` is in both the `main` and `get_int` functions. We probably want to pass in a prompt that the user sees when asked for input. Modify your code as follows.

```
def main():
    x = get_int("What's x? ")
```



```
print(f"x is {x}")

def get_int(prompt):
    while True:
        try:
            return int(input(prompt))
        except ValueError:
            pass

main()
```

You can learn more in Python's documentation of [pass](#).

Errors are inevitable in your code. However, you have the opportunity to use what was learned today to help prevent these errors. In this lecture, you learned about...

Exceptions

Value Errors

Runtime Errors

try

else

pass

Lecture 5

- Unit Tests
- assert
- pytest
- Testing Strings
- Organizing Tests into Folders
- Summing Up

Up until now, you have been likely testing your own code using `print` statements.

Alternatively, you may have been relying upon CS50 to test your code for you!

It's most common in industry to write code to test your own programs.

In your console window, type `code calculator.py`. Note that you may have previously coded this file in a previous lecture. In the text editor, make sure that your code appears as follows:

```
def main():  
    x = int(input("What's x? "))  
    print("x squared is", square(x))
```

```
def square(n):  
    return n * n
```

```
if __name__ == "__main__":  
    main()
```

Notice that you could plausibly test the above code on your own

using some obvious numbers such as 2. However, consider why you might want to create a test that ensures that the above code functions appropriately.

Following convention, let's create a new test program by typing code `test_calculator.py` and modify your code in the text editor as follows:

```
from calculator import square

def main():
    test_square()

def test_square():
    if square(2) != 4:
        print("2 squared was not 4")
    if square(3) != 9:
        print("3 squared was not 9")

if __name__ == "__main__":
    main()
```

Notice that we are importing the `square` function from `square.py` on the first line of code. By convention, we are creating a function called `test_square`. Inside that function, we define some conditions to test.

In the console window, type `python test_calculator.py`. You'll notice that nothing is being outputted. It could be that everything is running fine! Alternatively, it could be that our test function did not discover one of the "corner cases" that could produce an error. Right now, our code tests two conditions. If we wanted to test many

more conditions, our test code could easily become bloated. How could we expand our test capabilities without expanding our test code?

assert

Python's `assert` command allows us to tell the compiler that something, some assertion, is true. We can apply this to our test code as follows:

```
from calculator import square
```

```
def main():  
    test_square()
```

```
def test_square():  
    assert square(2) == 4  
    assert square(3) == 9
```

```
if __name__ == "__main__":  
    main()
```

Notice that we are definitively asserting what `square(2)` and `square(3)` should equal. Our code is reduced from four test lines down to two.

We can purposely break our calculator code by modifying it as follows:

```
def main():  
    x = int(input("What's x? "))
```

```
print("x squared is", square(x))

def square(n):
    return n + n

if __name__ == "__main__":
    main()
```

Notice that we have changed the `*` operator to a `+` in the square function.

Now running `python test_square.py` in the console window, you will notice that an `AssertionError` is raised by the compiler. Essentially, this is the compiler telling us that one of our conditions was not met.

One of the challenges that we are now facing is that our code could become even more burdensome if we wanted to provide more descriptive error output to our users. Plausibly, we could code as follows:

```
from calculator import square

def main():
    test_square()

def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared is not 4")
    try:
```

```

        assert square(3) == 9
    except AssertionError:
        print("3 squared is not 9")
    try:
        assert square(-2) == 4
    except AssertionError:
        print("-2 squared is not 4")
    try:
        assert square(-3) == 9
    except AssertionError:
        print("-3 squared is not 9")
    try:
        assert square(0) == 0
    except AssertionError:
        print("0 squared is not 0")

if __name__ == "__main__":
    main()

```

Notice that running this code will produce multiple errors. However, it's not producing all the errors above. This is a good illustration that it's worth testing multiple cases such that you might catch situations where there are coding mistakes.

The above code illustrates a major challenge: How could we make it easier to test your code without dozens of lines of code like the above?

You can learn more in Python's documentation of [assert](#).

pytest

pytest is a third-party library that allows you to unit test your program. That is, you can test your functions within your program.

To utilize pytest please type `pip install pytest` into your console window.

Before applying pytest to our own program, modify your `test_calculator` function as follows:

```
from calculator import square

def test_assert():
    assert square(2) == 4
    assert square(3) == 9
    assert square(-2) == 4
    assert square(-3) == 9
    assert square(0) == 0
```

Notice how the above code asserts all the conditions that we want to test.

pytest allows us to run our program directly through it, such that we can more easily view the results of our test conditions.

In the terminal window, type `pytest test_calculator.py`. You'll immediately notice that output will be provided. Notice the red F near the top of the output, indicating that something in your code failed. Further, notice that the red E provides some hints about the errors in your `calculator.py` program. Based upon the output, you can imagine a scenario where $3 * 3$ has outputted 6 instead of 9. Based on the results of this test, we can go correct our `calculator.py` code as follows:

```
def main():
    x = int(input("What's x? "))
```

```
print("x squared is", square(x))

def square(n):
    return n * n

if __name__ == "__main__":
    main()
```

Notice that we have changed the + operator to a * in the square function, returning it to a working state.

Re-running `pytest test_calculator.py`, notice how no errors are produced. Congratulations!

At the moment, it is not ideal that pytest will stop running after the first failed test. Again, let's return our `calculator.py` code back to its broken state:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n + n

if __name__ == "__main__":
    main()
```

Notice that we have changed the * operator to a + in the square function, returning it to a broken state.

To improve our test code, let's modify `test_calculator.py` to divide the code into different groups of tests:

```
from calculator import square
```

```
def test_positive():  
    assert square(2) == 4  
    assert square(3) == 9
```

```
def test_negative():  
    assert square(-2) == 4  
    assert square(-3) == 9
```

```
def test_zero():  
    assert square(0) == 0
```

Notice that we have divided the same five tests into three different functions. Testing frameworks like `pytest` will run each function, even if there was a failure in one of them. Re-running `pytest test_calculator.py`, you will notice that many more errors are being displayed. More error output allows you to further explore what might be producing the problems within your code.

Having improved our test code, return your `calculator.py` code to fully working order:

```
def main():  
    x = int(input("What's x? "))  
    print("x squared is", square(x))
```

```
def square(n):  
    return n * n  
  
if __name__ == "__main__":  
    main()
```

Notice that we have changed the + operator to a * in the square function, returning it to a working state.

Re-running `pytest test_calculator.py`, you will notice that no errors are found.

In summary, it's up to you as a coder to define as many test conditions as you see fit!

You can learn more in Pytest's documentation of [pytest](#).

Going back in time, consider the following code `hello.py`:

```
def main():  
    name = input("What's your name? ")  
    hello(name)  
  
def hello(to="world"):  
    print("hello,", to)  
  
if __name__ == "__main__":  
    main()
```

Notice that we may wish to test the result of the `hello` function.

Consider the following code for `test_hello.py`:

```
from hello import hello
```

```
def test_hello():  
    assert hello("David") == "hello, David"  
    assert hello() == "hello, world"
```

Looking at this code, do you think that this approach to testing will work well? Why might this test not work well? Notice that the `hello` function in `hello.py` prints something: That is, it does not return a value!

We can change our `hello` function within `hello.py` as follows:

```
def main():  
    name = input("What's your name? ")  
    print(hello(name))
```

```
def hello(to="world"):  
    return f"hello, {to}"
```

```
if __name__ == "__main__":  
    main()
```

Notice that we changed our `hello` function to return a string. This effectively means that we can now use `pytest` to test the `hello` function.

Running `pytest test_hello.py`, our code will pass all tests!

As with our previous test case in this lesson, we can break out our tests separately:

```
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    assert hello("David") == "hello, David"
```

Notice that the above code separates our test into multiple functions such that they will all run, even if an error is produced.

Unit testing code using multiple tests is so common that you have the ability to run a whole folder of tests with a single command. First, in the terminal window, execute `mkdir test` to create a folder called `test`.

Then, to create a test within that folder, type in the terminal window `code test/hello.py`. Notice that `test/` instructs the terminal to create `hello.py` in the folder called `test`.

In the text editor window, modify the file to include the following code:

```
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    assert hello("David") == "hello, David"
```

Notice that we are creating a test just as we did before.

pytest will not allow us to run tests as a folder simply with this file (or a whole set of files) alone without a special `__init__.py` file. In your terminal window, create this file by typing `test/__init__.py`. Note the `test/` as before, as well as the double underscores on either side of `init`. Even leaving this `__init__.py` file empty, pytest is informed that the whole folder containing `__init__.py` has tests that can be run.

Now, typing `pytest test` in the terminal, you can run the entire test folder of code.

You can learn more in Pytest's documentation of [import mechanisms](#).

Testing your code is a natural part of the programming process. Unit tests allow you to test specific aspects of your code. You can create your own programs that test your code. Alternatively, you can utilize frameworks like pytest to run your unit tests for you. In this lecture, you learned about...

Unit tests

assert

pytest

Lecture 2

- Loops
- While Loops
- For Loops
- Improving with User Input
- More About Lists
- Length
- Dictionaries
- Mario
- Summing Up

Essentially, loops are a way to do something over and over again. Begin by typing code `cat.py` in the terminal window.

In the text editor, begin with the following code:

```
print("meow")
print("meow")
print("meow")
```

Running this code by typing `python cat.py`, you'll notice that the program meows three times.

In developing as a programmer, you want to consider how one could improve areas of one's code where one types the same thing over and over again. Imagine where one might want to "meow" 500 times. Would it be logical to type that same expression of `print("meow")` over and over again?

Loops enable you to create a block of code that executes over and over again.

The `while` loop is nearly universal throughout all coding languages. Such a loop will repeat a block of code over and over again.

In the text editor window, edit your code as follows:

```
i = 3
while i != 0:
    print("meow")
```

Notice how even though this code will execute `print("meow")` multiple times, it will never stop! It will loop forever. `while` loops work by repeatedly asking if the condition of the loop has been fulfilled. In this case, the compiler is asking "does `i` not equal zero?" When you get stuck in a loop that executes forever, you can press `control-c` on your keyboard to break out of the loop.

To fix this loop that lasts forever, we can edit our code as follows

```
i = 3
while i != 0:
    print("meow")
    i = i - 1
```

Notice that now our code executes properly, reducing `i` by 1 for each "iteration" through the loop. This term iteration has special significance within coding. By iteration, we mean one cycle through the loop. The first iteration is the "0th" iteration through the loop. The second is the "1st" iteration. In programming we count starting with 0, then 1, then 2.

We can further improve our code as follows:

```
i = 1
while i <= 3:
    print("meow")
    i = i + 1
```

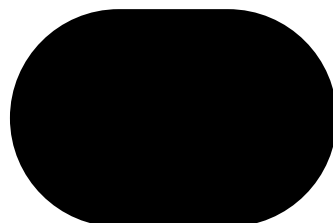
Notice that when we code `i = i + 1` we assign the value of `i` from the right to the left. Above, we are starting `i` at one, like most humans count (1, 2, 3). If you execute the code above, you'll see it meows three times. It's best practice in programming to begin counting with zero.

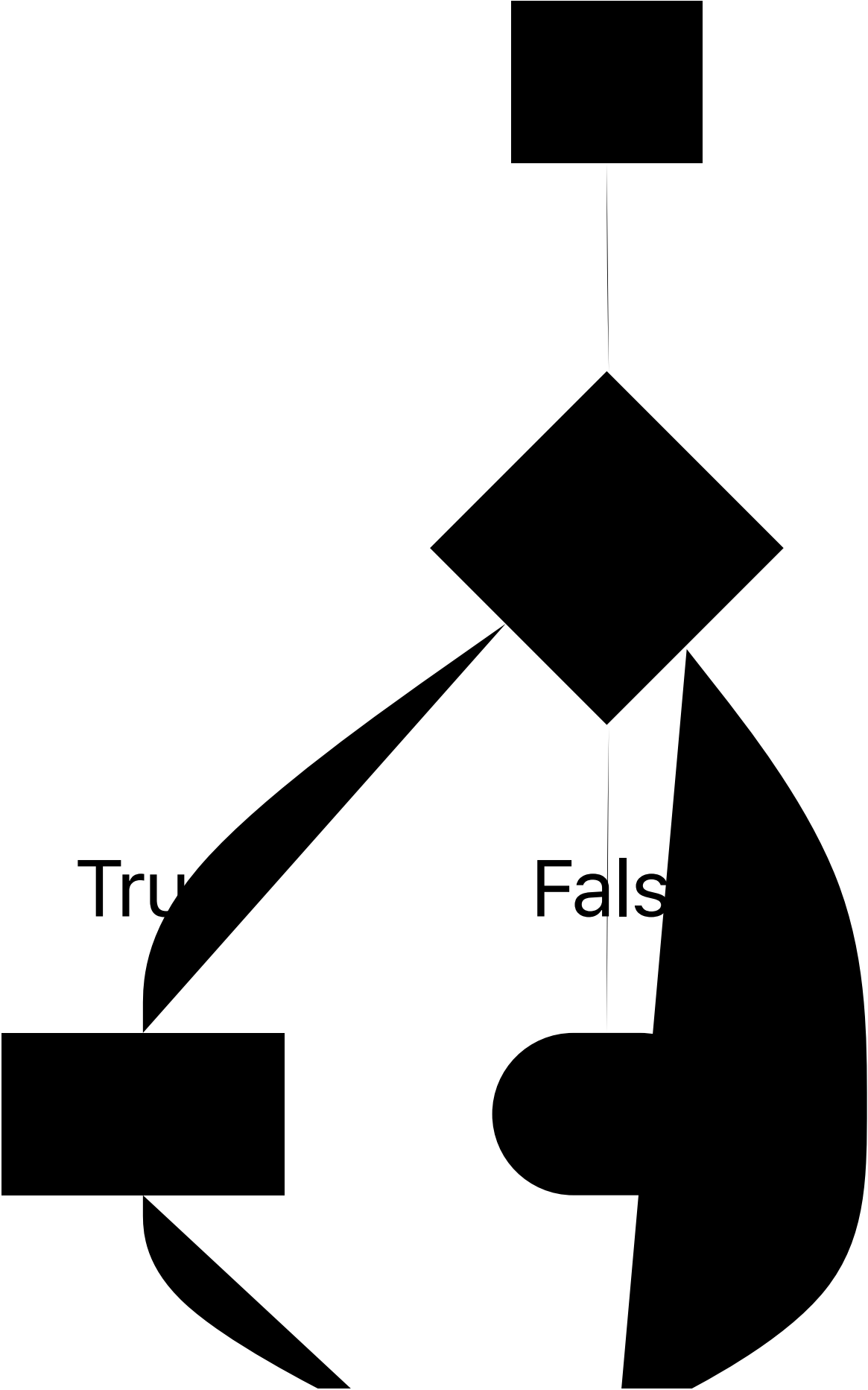
We can improve our code, to start counting with zero:

```
i = 0
while i < 3:
    print("meow")
    i += 1
```

Notice how changing the operator to `i < 3` allows our code to function as intended. We begin by counting with 0 and it iterates through our loop three times, producing three meows. Also, notice how `i += 1` is the same as saying `i = i + 1`.

Our code at this point is illustrated as follows:







Notice how our loop counts `i` up to, but not through 3.

A `for` loop is a different type of loop.

To best understand a `for` loop, it's best to begin by talking about a new variable type called a `list` in Python. As in other areas of our lives, we can have a grocery list, a to-do list, etc.

A `for` loop iterates through a `list` of items. For example, in the text editor window, modify your `cat.py` code as follows:

```
for i in [0, 1, 2]:  
    print("meow")
```

Notice how clean this code is compared to your previous `while` loop code. In this code, `i` begins with 0, meows, `i` is assigned 1, meows, and, finally, `i` is assigned 2, meows, and then ends.

While this code accomplishes what we want, there are some possibilities for improving our code for extreme cases. At first glance, our code looks great. However, what if you wanted to iterate up to a million? It's best to create code that can work with such extreme cases. Accordingly, we can improve our code as follows:

```
for i in range(3):  
    print("meow")
```

Notice how `range(3)` provides back three values (0, 1, and 2) automatically. This code will execute and produce the intended effect, meowing three times.

Our code can be further improved. Notice how we never use `i` explicitly in our code. That is, while Python needs the `i` as a place to store the number of the iteration of the loop, we never use it for any other purpose. In Python, if such a variable does not have any other significance in our code, we can simply represent this variable as a single underscore `_`. Therefore, you can modify your code as follows:

```
for _ in range(3):  
    print("meow")
```

Notice how changing the `i` to `_` has zero impact on the functioning of our program.

Our code can be further improved. To stretch your mind to the possibilities within Python, consider the following code:

Notice how it will meow three times, but the program will produce `meowmeowmeow` as the result. Consider: How could you create a line break at the end of each meow?

Indeed, you can edit your code as follows:

```
print("meow\n" * 3, end="")
```

Notice how this code produces three meows, each on a separate line. By adding `end=""` and the `\n` we tell the compiler to add a line break at the end of each meow.

Perhaps we want to get input from our user. We can use loops as a way of validating the input of the user.

A common paradigm within Python is to use a `while` loop to validate the input of the user.

For example, let's try prompting the user for a number greater than or equal 0:

```
while True:
    n = int(input("What's n? "))
    if n < 0:
        continue
    else:
        break
```

Notice that we've introduced two new keywords in Python, `continue` and `break`. `continue` explicitly tells Python to go to the next iteration of a loop. `break`, on the other hand, tells Python to "break out" of a loop early, before it has finished all of its iterations. In this case, we'll continue to the next iteration of the loop when `n` is less than 0—ultimately reprompting the user with "What's n?". If though, `n` is greater than or equal to 0, we'll break out of the loop and allow the rest of our program to run.

It turns out that the `continue` keyword is redundant in this case. We can improve our code as follows:

```
while True:
    n = int(input("What's n? "))
    if n > 0:
        break

for _ in range(n):
    print("meow")
```

Notice how this while loop will always run (forever) until `n` is greater than 0. When `n` is greater than 0, the loop breaks.

Bringing in our prior learning, we can use functions to further to improve our code:

```
def main():
    number = get_number()
    meow(number)

def get_number():
    while True:
        n = int(input("What's n? "))
        if n > 0:
            break
    return n

def meow(n):
    for _ in range(n):
        print("meow")
```

Notice how not only did we change your code to operate in multiple functions, but we also used a return statement to return the value of `n` back to the main function.

Consider the world of Hogwarts from the famed Harry Potter universe.

In the terminal, type `code hogwarts.py`.

In the text editor, code as follows:

```
students = ["Hermoine", "Harry", "Ron"]
```

```
print(students[0])  
print(students[1])  
print(students[2])
```

Notice how we have a `list` of students with their names as above. We then print the student who is at the 0th location, "Hermoine". Each of the other students are printed as well.

Just as we illustrated previously, we can use a loop to iterate over the list. You can improve your code as follows:

```
students = ["Hermoine", "Harry", "Ron"]  
  
for student in students:  
    print(student)
```

Notice that for each `student` in the `students` list, it will print the student as intended. You might wonder why we did not use the `_` designation as discussed prior. We choose not to do this because `student` is explicitly used in our code.

You can learn more in Python's documentation of [lists](#).

We can utilize `len` as a way of checking the length of the `list` called `students`.

Imagine that you don't simply want to print the name of the student, but also their position in the list. To accomplish this, you can edit your code as follows:

```
students = ["Hermoine", "Harry", "Ron"]
```

```
for i in range(len(students)):
    print(i + 1, students[i])
```

Notice how executing this code results in not only getting the position of each student plus one using `i + 1`, but also prints the name of each student. `len` allow you to dynamically see how long the list of the students is regardless how much it grows.

You can learn more in Python's documentation of [len](#).

`dicts` or `dictionaries` is a data structure that allows you to associate keys with values.

Where a `list` is a list of multiple values, a `dict` associates a key with a value.

Considering the houses of Hogwarts, we might assign specific students to specific houses.

We could use `lists` alone to accomplish this:

```
students = ["Hermoine", "Harry", "Ron", "Draco"]
houses = ["Gryffindor", "Gryffindor", "Griffindor", "Slytherin"]
```

Notice that we could promise that we will always keep these lists in order. The individual at the first position of `students` is associated with the house at the first position of the `houses` list, and so on. However, this can become quite cumbersome as our lists grow!

We can better our code using a `dict` as follows:

```
students = {
    "Hermoine": "Gryffindor",
```

```
"Harry": "Gryffindor",  
"Ron": "Gryffindor",  
"Draco": "Slytherin",  
}  
print(students["Hermoine"])  
print(students["Harry"])  
print(students["Ron"])  
print(students["Draco"])
```

Notice how we use {} curly braces to create a dictionary. Where lists use numbers to iterate through the list, dicts allow us to use words.

Run your code and make sure your output is as follows:

```
$ python hogwarts.py  
Gryffindor  
Gryffindor  
Gryffindor  
Slytherin
```

We can improve our code as follows:

```
students = {  
    "Hermoine": "Gryffindor",  
    "Harry": "Gryffindor",  
    "Ron": "Gryffindor",  
    "Draco": "Slytherin",  
}  
for student in students:  
    print(student)
```

Notice how executing this code, the for loop will only iterate through all the keys, resulting in a list of the names of the students. How

could we print out both values and keys?

Modify your code as follows:

```
students = {  
    "Hermoine": "Gryffindor",  
    "Harry": "Gryffindor",  
    "Ron": "Gryffindor",  
    "Draco": "Slytherin",  
}  
for student in students:  
    print(student, students[student])
```

Notice how `students[student]` will go to each student's key and find the value of the their house. Execute your code and you'll notice how the output is a bit messy.

We can clean up the print function by improving our code as follows:

```
students = {  
    "Hermoine": "Gryffindor",  
    "Harry": "Gryffindor",  
    "Ron": "Gryffindor",  
    "Draco": "Slytherin",  
}  
for student in students:  
    print(student, students[student], sep=", ")
```

Notice how this creates a clean separation of a , between each item printed.

If you execute `python hogwarts.py`, you should see the following:

```
$ python hogwarts.py
```

Hermoine, Gryffindor
Harry, Gryffindor
Ron, Gryffindor
Draco, Slytherin

What if we have more information about our students? How could we associate more data with each of the students?

You can imagine wanting to have lots of data associated multiple things with one key. Enhance your code as follows:

```
students = [  
    {"name": "Hermoine", "house": "Gryffindor", "patronus": "Otter"},  
    {"name": "Harry", "house": "Gryffindor", "patronus": "Stag"},  
    {"name": "Ron", "house": "Gryffindor", "patronus": "Jack Russell t  
    {"name": "Draco", "house": "Slytherin", "patronus": None},  
]
```

Notice how this code creates a list of dicts. The list called students has four dicts within it: One for each student. Also, notice that Python has a special None designation where there is no value associated with a key.

Now, you have access to a whole host of interesting data about these students. Now, further modify you code as follows:

```
students = [  
    {"name": "Hermoine", "house": "Gryffindor", "patronus": "Otter"},  
    {"name": "Harry", "house": "Gryffindor", "patronus": "Stag"},  
    {"name": "Ron", "house": "Gryffindor", "patronus": "Jack Russell t  
    {"name": "Draco", "house": "Slytherin", "patronus": None},  
]
```

```
for student in students:
    print(student["name"], student["house"], student["patronus"], sep=
```

Notice how the for loop will iterate through each of the dicts inside the list called students.

You can learn more in Python's Documentation of [dicts](#).

Remember that the classic game Mario has a hero jumping over bricks. Let's create a textual representation of this game.

Begin coding as follows:

```
print("#")
print("#")
print("#")
```

Notice how we are copying and pasting the same code over and over again.

Consider how we could better the code as follows:

```
for _ in range(3):
    print("#")
```

Notice how this accomplishes essentially what we want to create.

Consider: Could we further abstract for solving more sophisticated problems later with this code? Modify your code as follows:

```
def main():
    print_column(3)
```

```
def print_column(height):  
    for _ in range(height):  
        print("#")
```

```
main()
```

Notice how our column can grow as much as we want without any hard coding.

Now, let's try to print a row horizontally. Modify your code as follows:

```
def main():  
    print_row(4)
```

```
def print_row(width):  
    print "?" * width
```

```
main()
```

Notice how we now have code that can create left to right blocks.

Examining the slide below, notice how Mario has both rows and columns of blocks.

Consider, how could we implement both rows and columns within our code? Modify your code as follows:

```
def main():  
    print_square(3)
```

```
def print_square(size):  
  
    # For each row in square  
    for i in range(size):  
  
        # For each brick in row  
        for j in range(size):  
  
            # Print brick  
            print("#", end="")  
  
        # Print blank line  
        print()  
  
main()
```

Notice that we have an outer loop addresses each row in the square. Then, we have an inner loop that prints a brick in each row. Finally, we have a print statement that prints a blank line.

We can further abstract away our code:

```
def main():  
    print_square(3)  
  
def print_square(size):  
    for i in range(size):  
        print_row(size)  
  
def print_row(width):  
    print("#" * width)
```

```
main()
```

You now have another power in your growing list of your Python abilities. In this lecture we addressed...

Loops

while

for

len

list

dict

Lecture 9

- Et Cetera
- set
- Global Variables
- Constants
- Type Hints
- Docstrings
- argparse
- Unpacking
- args and kwargs
- map
- List Comprehensions
- filter
- Dictionary Comprehensions
- enumerate
- Generators and Iterators
- Congratulations!
- This was CS50!

Over the many past lessons, we have covered so much related to Python!

In this lesson, we will be focusing upon many of the “et cetera” items not previously discussed. “Et cetera” literally means “and the rest”! Indeed, if you look at the Python documentation, you will find quite “the rest” of other features.

set

In math, a set would be considered a set of numbers without any duplicates.

In the text editor window, code as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
    {"name": "Padma", "house": "Ravenclaw"},  
]  
  
houses = []  
for student in students:  
    if student["house"] not in houses:  
        houses.append(student["house"])  
  
for house in sorted(houses):  
    print(house)
```

Notice how we have a list of dictionaries, each being a student. An empty list called `houses` is created. We iterate through each student in `students`. If a student's house is not in `houses`, we append to our list of houses.

It turns out we can use the built-in set features to eliminate duplicates.

In the text editor window, code as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
    {"name": "Padma", "house": "Ravenclaw"},  
]
```



```
houses = set()
for student in students:
    houses.add(student["house"])

for house in sorted(houses):
    print(house)
```

Notice how no checking needs to be included to ensure there are no duplicates. The set object takes care of this for us automatically.

You can learn more in Python's documentation of [set](#).

In other programming languages, there is the notion of global variables that are accessible to any function.

We can leverage this ability within Python. In the text editor window, code as follows:

```
balance = 0

def main():
    print("Balance:", balance)

if __name__ == "__main__":
    main()
```

Notice how we create a global variable called `balance`, outside of any function.

Since no errors are presented by executing the code above, you'd think all is well. However, it is not! In the text editor window, code as

follows:

```
balance = 0
```

```
def main():  
    print("Balance:", balance)  
    deposit(100)  
    withdraw(50)  
    print("Balance:", balance)
```

```
def deposit(n):  
    balance += n
```

```
def withdraw(n):  
    balance -= n
```

```
if __name__ == "__main__":  
    main()
```

Notice how we now add the functionality to add and withdraw funds to and from `balance`. However, executing this code, we are presented with an error! We see an error called `UnboundLocalError`. You might be able to guess that, at least in the way we've currently coded `balance` and our `deposit` and `withdraw` functions, we can't reassign it a value inside a function.

To interact with a global variable inside a function, the solution is to use the `global` keyword. In the text editor window, code as follows:

```
balance = 0
```

```

def main():
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)

def deposit(n):
    global balance
    balance += n

def withdraw(n):
    global balance
    balance -= n

if __name__ == "__main__":
    main()

```

Notice how the `global` keyword tells each function that `balance` does not refer to a local variable: instead, it refers to the global variable we originally placed at the top of our code. Now, our code functions!

Utilizing our powers from our experience with object-oriented programming, we can modify our code to use a class instead of a global variable. In the text editor window, code as follows:

```

class Account:
    def __init__(self):
        self._balance = 0

    @property

```

```

def balance(self):
    return self._balance

def deposit(self, n):
    self._balance += n

def withdraw(self, n):
    self._balance -= n

def main():
    account = Account()
    print("Balance:", account.balance)
    account.deposit(100)
    account.withdraw(50)
    print("Balance:", account.balance)

if __name__ == "__main__":
    main()

```

Notice how we use `account = Account()` to create an account. Classes allow us to solve this issue of needing a global variable more cleanly because these instance variables are accessible to all the methods of this class utilizing `self`.

Generally speaking, global variables should be used quite sparingly, if at all!

Some languages allow you to create variables that are unchangeable, called "constants". Constants allow one to program defensively and reduce the opportunities for important values to be altered.

In the text editor window, code as follows:

```
MEOWS = 3
```

```
for _ in range(MEOWS):  
    print("meow")
```

Notice MEOWS is our constant in this case. Constants are typically denoted by capital variable names and are placed at the top of our code. Though this *looks* like a constant, in reality, Python actually has no mechanism to prevent us from changing that value within our code! Instead, you're on the honor system: if a variable name is written in all caps, just don't change it!

One can create a class "constant", now in quotes because we know Python doesn't quite support "constants". In the text editor window, code as follows:

```
class Cat:  
    MEOWS = 3  
  
    def meow(self):  
        for _ in range(Cat.MEOWS):  
            print("meow")
```

```
cat = Cat()  
cat.meow()
```

Because MEOWS is defined outside of any particular class method, all of them have access to that value via Cat.MEOWS.

In other programming languages, one expresses explicitly what variable type you want to use.

As we saw earlier in the course, Python does require the explicit declaration of types.

Nevertheless, it's good practice need to ensure all of your variables are of the right type.

mypy is a program that can help you test to make sure all your variables are of the right type.

You can install mypy by executing in your terminal window: `pip install mypy`.

In the text editor window, code as follows:

```
def meow(n):  
    for _ in range(n):  
        print("meow")  
  
number = input("Number: ")  
meow(number)
```

You may already see that `number = input("Number: ")` returns a string, not an int. But `meow` will likely want an int!

A type hint can be added to give Python a hint of what type of variable `meow` should expect. In the text editor window, code as follows:

```
def meow(n: int):  
    for _ in range(n):  
        print("meow")  
  
number = input("Number: ")  
meow(number)
```

Notice, though, that our program still throws an error.

After installing `mypy`, execute `mypy meows.py` in the terminal window. `mypy` will provide some guidance about how to fix this error.

You can annotate all your variables. In the text editor window, code as follows:

```
def meow(n: int):  
    for _ in range(n):  
        print("meow")  
  
number: int = input("Number: ")  
meow(number)
```

Notice how `number` is now provided a type hint.

Again, executing `mypy meows.py` in the terminal window provides much more specific feedback to you, the programmer.

We can fix our final error by coding as follows:

```
def meow(n: int):  
    for _ in range(n):  
        print("meow")  
  
number: int = int(input("Number: "))  
meow(number)
```

Notice how running `mypy` now produces no errors because we cast our input as an integer.

Let's introduce a new error by assuming that `meow` will return to us a string, or `str`. In the text editor window, code as follows:

```
def meow(n: int):  
    for _ in range(n):  
        print("meow")  
  
number: int = int(input("Number: "))  
meows: str = meow(number)  
print(meows)
```

Notice how the `meow` function has only a side effect. Because we only attempt to print "meow", not return a value, an error is thrown when we try to store the return value of `meow` in `meows`.

We can further use type hints to check for errors, this time annotating the return values of functions. In the text editor window, code as follows:

```
def meow(n: int) -> None:  
    for _ in range(n):  
        print("meow")  
  
number: int = int(input("Number: "))  
meows: str = meow(number)  
print(meows)
```

Notice how the notation `-> None` tells mypy that there is no return value.

We can modify our code to return a string if we wish:


```
def meow(n: int) -> str:
    return "meow\n" * n
```

```
number: int = int(input("Number: "))
meows: str = meow(number)
print(meows, end="")
```

Notice how we store in `meows` multiple `str`s. Running `mypy` produces no errors.

You can learn more in Python's documentation of [Type Hints](#). You can learn more about [mypy](#) through the program's own documentation.

A standard way of commenting your function's purpose is to use a docstring. In the text editor window, code as follows:

```
def meow(n):
    """Meow n times."""
    return "meow\n" * n

number = int(input("Number: "))
meows = meow(number)
print(meows, end="")
```

Notice how the three double quotes designate what the function does.

You can use docstrings to standardize how you document the features of a function. In the text editor window, code as follows: s

```
def meow(n):
    """
```

Meow n times.

```
:param n: Number of times to meow
:type n: int
:raise TypeError: If n is not an int
:return: A string of n meows, one per line
:rtype: str
"""
return "meow\n" * n
```

```
number = int(input("Number: "))
meows = meow(number)
print(meows, end="")
```

Notice how multiple docstring arguments are included. For example, it describes the parameters taken by the function and what is returned by the function.

Established tools, such as [Sphinx](#), can be used to parse docstrings and automatically create documentation for us in the form of web pages and PDF files such that you can publish and share with others. You can learn more in Python's documentation of [docstrings](#).

argparse

Suppose we want to use command-line arguments in our program. In the text editor window, code as follows:

```
import sys

if len(sys.argv) == 1:
    print("meow")
elif len(sys.argv) == 3 and sys.argv[1] == "-n":
    n = int(sys.argv[2])
```

```
    for _ in range(n):
        print("meow")
else:
    print("usage: meows.py [-n NUMBER]")
```

Notice how `sys` is imported, from which we get access to `sys.argv`—an array of command-line arguments given to our program when run. We can use several `if` statements to check whether the user has run our program properly.

Let's assume that this program will be getting much more complicated. How could we check all the arguments that could be inserted by the user? We might give up if we have more than a few command-line arguments!

Luckily, `argparse` is a library that handles all the parsing of complicated strings of command-line arguments. In the text editor window, code as follows:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-n")
args = parser.parse_args()

for _ in range(int(args.n)):
    print("meow")
```

Notice how `argparse` is imported instead of `sys`. An object called `parser` is created from an `ArgumentParser` class. That class's `add_argument` method is used to tell `argparse` what arguments we should expect from the user when they run our program. Finally, running the `parser`'s `parse_args` method ensures that all of the

arguments have been included properly by the user.

We can also program more cleanly, such that our user can get some information about the proper usage of our code when they fail to use the program correctly. In the text editor window, code as follows:

```
import argparse

parser = argparse.ArgumentParser(description="Meow like a cat")
parser.add_argument("-n", help="number of times to meow")
args = parser.parse_args()

for _ in range(int(args.n)):
    print("meow")
```

Notice how the user is provided some documentation. Specifically, a help argument is provided. Now, if the user executes `python meows.py --help` or `-h`, the user will be presented with some clues about how to use this program.

We can further improve this program. In the text editor window, code as follows:

```
import argparse

parser = argparse.ArgumentParser(description="Meow like a cat")
parser.add_argument("-n", default=1, help="number of times to meow", t
args = parser.parse_args()

for _ in range(args.n):
    print("meow")
```

Notice how not only is help documentation included, but you can provide a default value when no arguments are provided by the

user.

You can learn more in Python's documentation of [argparse](#).

Would it not be nice to be able to split a single variable into two variables? In the text editor window, code as follows:

```
first, _ = input("What's your name? ").split(" ")
print(f"hello, {first}")
```

Notice how this program tries to get a user's first name by naively splitting on a single space.

It turns out there are other ways to unpack variables. You can write more powerful and elegant code by understanding how to unpack variables in seemingly more advanced ways. In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

print(total(100, 50, 25), "Knuts")
```

Notice how this returns the total value of Knuts.

What if we wanted to store our coins in a list? In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts
```

```
coins = [100, 50, 25]
```

```
print(total(coins[0], coins[1], coins[2]), "Knuts")
```

Notice how a list called `coins` is created. We can pass each value in by indexing using `0`, `1`, and so on.

This is getting quite verbose. Wouldn't it be nice if we could simply pass the list of coins to our function?

To enable the possibility of passing the entire list, we can use unpacking. In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):  
    return (galleons * 17 + sickles) * 29 + knuts
```

```
coins = [100, 50, 25]
```

```
print(total(*coins), "Knuts")
```

Notice how a `*` unpacks the sequence of the list of coins and passes in each of its individual elements to `total`.

Suppose that we could pass in the names of the currency in any order? In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):  
    return (galleons * 17 + sickles) * 29 + knuts
```

```
print(total(galleons=100, sickles=50, knuts=25), "Knuts")
```

Notice how this still calculates correctly.

When you start talking about “names” and “values,” dictionaries might start coming to mind! You can implement this as a dictionary. In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):  
    return (galleons * 17 + sickles) * 29 + knuts  
  
coins = {"galleons": 100, "sickles": 50, "knuts": 25}  
  
print(total(coins["galleons"], coins["sickles"], coins["knuts"]), "Kn
```

Notice how a dictionary called `coins` is provided. We can index into it using keys, such as “galleons” or “sickles”.

Since the `total` function expects three arguments, we cannot pass in a dictionary. We can use unpacking to help with this. In the text editor window, code as follows:

```
def total(galleons, sickles, knuts):  
    return (galleons * 17 + sickles) * 29 + knuts  
  
coins = {"galleons": 100, "sickles": 50, "knuts": 25}  
  
print(total(**coins), "Knuts")
```

Notice how `**` allows you to unpack a dictionary. When unpacking a dictionary, it provides both the keys and values.

args and kwargs

Recall the print documentation we looked at earlier in this course:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

args are positional arguments, such as those we provide to print like `print("Hello", "World")`.

kwargs are named arguments, or "keyword arguments", such as those we provide to print like `print(end="")`.

As we see in the prototype for the `print` function above, we can tell our function to expect a presently unknown number positional arguments. We can also tell it to expect a presently unknown number of keyword arguments. In the text editor window, code as follows:

```
def f(*args, **kwargs):  
    print("Positional:", args)
```

```
f(100, 50, 25)
```

Notice how executing this code will be printed as positional arguments.

We can even pass in named arguments. In the text editor window, code as follows:

```
def f(*args, **kwargs):  
    print("Named:", kwargs)
```

```
f(galleons=100, sickles=50, knuts=25)
```

Notice how the named values are provided in the form of a

dictionary.

Thinking about the `print` function above, you can see how `*objects` takes any number of positional arguments.

You can learn more in Python's documentation of [print](#).

map

Early on, we began with procedural programming.

We later revealed Python is an object oriented programming language.

We saw hints of functional programming, where functions have side effects without a return value. We can illustrate this in the text editor window, type code `yell.py` and code as follows:

```
def main():
    yell("This is CS50")

def yell(word):
    print(word.upper())

if __name__ == "__main__":
    main()
```

Notice how the `yell` function is simply yelled.

Wouldn't it be nice to yell a list of unlimited words? Modify your code as follows:

```
def main():
    yell(["This", "is", "CS50"])
```

```
def yell(words):
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

if __name__ == "__main__":
    main()
```

Notice we accumulate the uppercase words, iterating over each of the words and uppercasing them. The uppercase list is printed utilizing the `*` to unpack it.

Removing the brackets, we can pass the words in as arguments. In the text editor window, code as follows:

```
def main():
    yell("This", "is", "CS50")

def yell(*words):
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

if __name__ == "__main__":
    main()
```

Notice how `*words` allows for many arguments to be taken by the function.

map allows you to map a function to a sequence of values. In practice, we can code as follows:

```
def main():
    yell("This", "is", "CS50")

def yell(*words):
    uppercased = map(str.upper, words)
    print(*uppercased)

if __name__ == "__main__":
    main()
```

Notice how map takes two arguments. First, it takes a function we want applied to every element of a list. Second, it takes that list itself, to which we'll apply the aforementioned function. Hence, all words in words will be handed to the str.upper function and returned to uppercased.

You can learn more in Python's documentation of [map](#).

List comprehensions allow you to create a list on the fly in one elegant one-liner.

We can implement this in our code as follows:

```
def main():
    yell("This", "is", "CS50")

def yell(*words):
    uppercased = [arg.upper() for arg in words]
```

```
print(*uppercased)
```

```
if __name__ == "__main__":  
    main()
```

Notice how instead of using `map`, we write a Python expression within square brackets. For each argument, `.upper` is applied to it.

Taking this concept further, let's pivot toward another program.

In the text editor window, type code `gryffindors.py` and code as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
]
```

```
gryffindors = []  
for student in students:  
    if student["house"] == "Gryffindor":  
        gryffindors.append(student["name"])
```

```
for gryffindor in sorted(gryffindors):  
    print(gryffindor)
```

Notice we have a conditional while we're creating our list. *If* the student's house is Gryffindor, we append the student to the list of names. Finally, we print all the names.

More elegantly, we can simplify this code with a list comprehension

as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
]  
  
gryffindors = [  
    student["name"] for student in students if student["house"] ==  
    "Gryffindor"  
]  
  
for gryffindor in sorted(gryffindors):  
    print(gryffindor)
```

Notice how the list comprehension is on a single line!

filter

Using Python's `filter` function allows us to return a subset of a sequence for which a certain condition is true.

In the text editor window, code as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
]  
  
def is_gryffindor(s):
```

```
return s["house"] == "Gryffindor"
```

```
gryffindors = filter(is_gryffindor, students)
```

```
for gryffindor in sorted(gryffindors, key=lambda s: s["name"]):  
    print(gryffindor["name"])
```

Notice how a function called `is_gryffindor` is created. This is our filtering function that will take a student `s`, and return `True` or `False` depending on whether the student's house is Gryffindor. You can see the new `filter` function takes two arguments. First, it takes the function that will be applied to each element in a sequence—in this case, `is_gryffindor`. Second, it takes the sequence to which it will apply the filtering function—in this case, `students`. In `gryffindors`, we should see only those students who are in Gryffindor.

`filter` can also use lambda functions as follows:

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Ron", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"},  
]
```

```
gryffindors = filter(lambda s: s["house"] == "Gryffindor", students)
```

```
for gryffindor in sorted(gryffindors, key=lambda s: s["name"]):  
    print(gryffindor["name"])
```

Notice how the same list of students is provided.

You can learn more in Python's documentation of [filter](#).

We can apply the same idea behind list comprehensions to dictionaries. In the text editor window, code as follows:

```
students = ["Hermione", "Harry", "Ron"]

gryffindors = []

for student in students:
    gryffindors.append({"name": student, "house": "Gryffindor"})

print(gryffindors)
```

Notice how this code doesn't (yet!) use any comprehensions. Instead, it follows the same paradigms we have seen before.

We can now apply dictionary comprehensions by modifying our code as follows:

```
students = ["Hermione", "Harry", "Ron"]

gryffindors = [{"name": student, "house": "Gryffindor"} for student in students]

print(gryffindors)
```

Notice how all the prior code is simplified into a single line where the structure of the dictionary is provided for each student in students.

We can even simplify further as follows:

```
students = ["Hermione", "Harry", "Ron"]
```

```
gryffindors = {student: "Gryffindor" for student in students}

print(gryffindors)
```

Notice how the dictionary will be constructed with key-value pairs.

enumerate

We may wish to provide some ranking of each student. In the text editor window, code as follows:

```
students = ["Hermione", "Harry", "Ron"]

for i in range(len(students)):
    print(i + 1, students[i])
```

Notice how each student is enumerated when running this code.

Utilizing enumeration, we can do the same:

```
students = ["Hermione", "Harry", "Ron"]

for i, student in enumerate(students):
    print(i + 1, student)
```

Notice how enumerate presents the index and the value of each student.

You can learn more in Python's documentation of [enumerate](#).

In Python, there is a way to protect against your system running out of resources the problems they are addressing become too large. In the United States, it's customary to "count sheep" in one's mind

when one is having a hard time falling asleep.

In the text editor window, type code `sleep.py` and code as follows:

Notice how this program will count the number of sheep you ask of it.

We can make our program more sophisticated by adding a `main` function by coding as follows:

Notice how a `main` function is provided.

We have been getting into the habit of abstracting away parts of our code.

We can call a `sheep` function by modifying our code as follows:

Notice how the `main` function does the iteration.

We can provide the `sheep` function more abilities. In the text editor window, code as follows:

Notice how we create a flock of sheep and return the `flock`.

Executing our code, you might try different numbers of sheep such as 10, 1000, and 10000. What if you asked for 1000000 sheep, your program might completely hang or crash. Because you have attempted to generate a massive list of sheep, your computer may be struggling to complete the computation.

The `yield` generator can solve this problem by returning a small bit of the results at a time. In the text editor window, code as follows:

Notice how `yield` provides only one value at a time while the `for` loop keeps working.

You can learn more in Python's documentation of [generators](#).

You can learn more in Python's documentation of [iterators](#).

As you exit from this course, you have more of a mental model and toolbox to address programming-related problems.

First, you learned about functions and variables.

Second, you learned about conditionals.

Third, you learned about loops.

Fourth, you learned about exceptions.

Fifth, you learned about libraries.

Sixth, you learned about unit tests.

Seventh, you learned about file I/O.

Eighth, you learned about regular expressions.

Most recently, you learned about object-oriented programming.

Today, you learned about many other tools you can use.

Creating a final program together, type code `say.py` in your terminal window and code as follows:

```
import cowsay
import pyttsx3

engine = pyttsx3.init()
this = input("What's this? ")
cowsay.cow(this)
engine.say(this)
engine.runAndWait()
```

Notice how running this program provides you with a spirited send-off.

Our great hope is that you will use what you learned in this course to address real problems in the world, making our globe a better place.

This was CS50!

Lecture 1

- Conditionals
- if Statements
- Control Flow, elif, and else
- or
- and
- Modulo
- Creating Our Own Parity Function
- Pythonic
- match
 - Summing Up

Conditionals allow you, the programmer, to allow your program to make decisions: As if your program has the choice between taking the left-hand road or the right-hand road based upon certain conditions.

Built within Python are a set of "operators" that can be used to ask mathematical questions.

> and < symbols are probably quite familiar to you.

>= denotes "greater than or equal to."

<= denotes "less than or equal to."

== denotes "equals, though do notice the double equal sign! A single equal sign would assign a value. Double equal signs are used to compare variables.

!= denotes "not equal to."

Conditional statements compare a left-hand term to a right-hand term.

In your terminal window, type `code compare.py`. This will create a brand new file called "compare."

In the text editor window, begin with the following:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x < y:
    print("x is less than y")
```

Notice how your program takes the input of the user for both `x` and `y`, casting them as integers and saving them into their respective `x` and `y` variables. Then, the `if` statement compares `x` and `y`. If the condition of `x < y` is met, the `print` statement is executed.

If statements use `bool` or boolean values (`true` or `false`) to decide whether or not to execute. If the statement of `x > y` is true, the compiler will register it as `true` and execute the code.

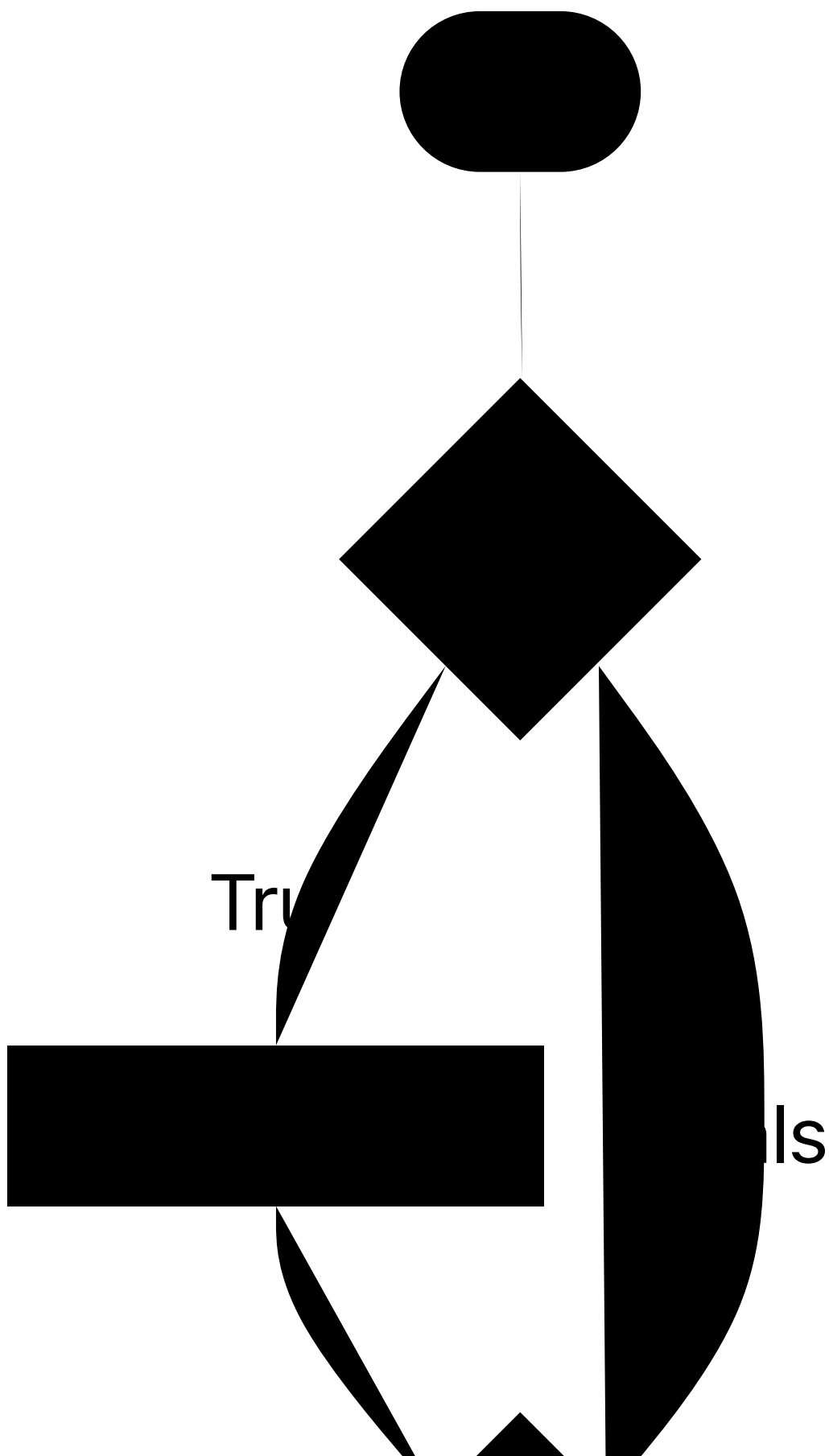
Further revise your code as follows:

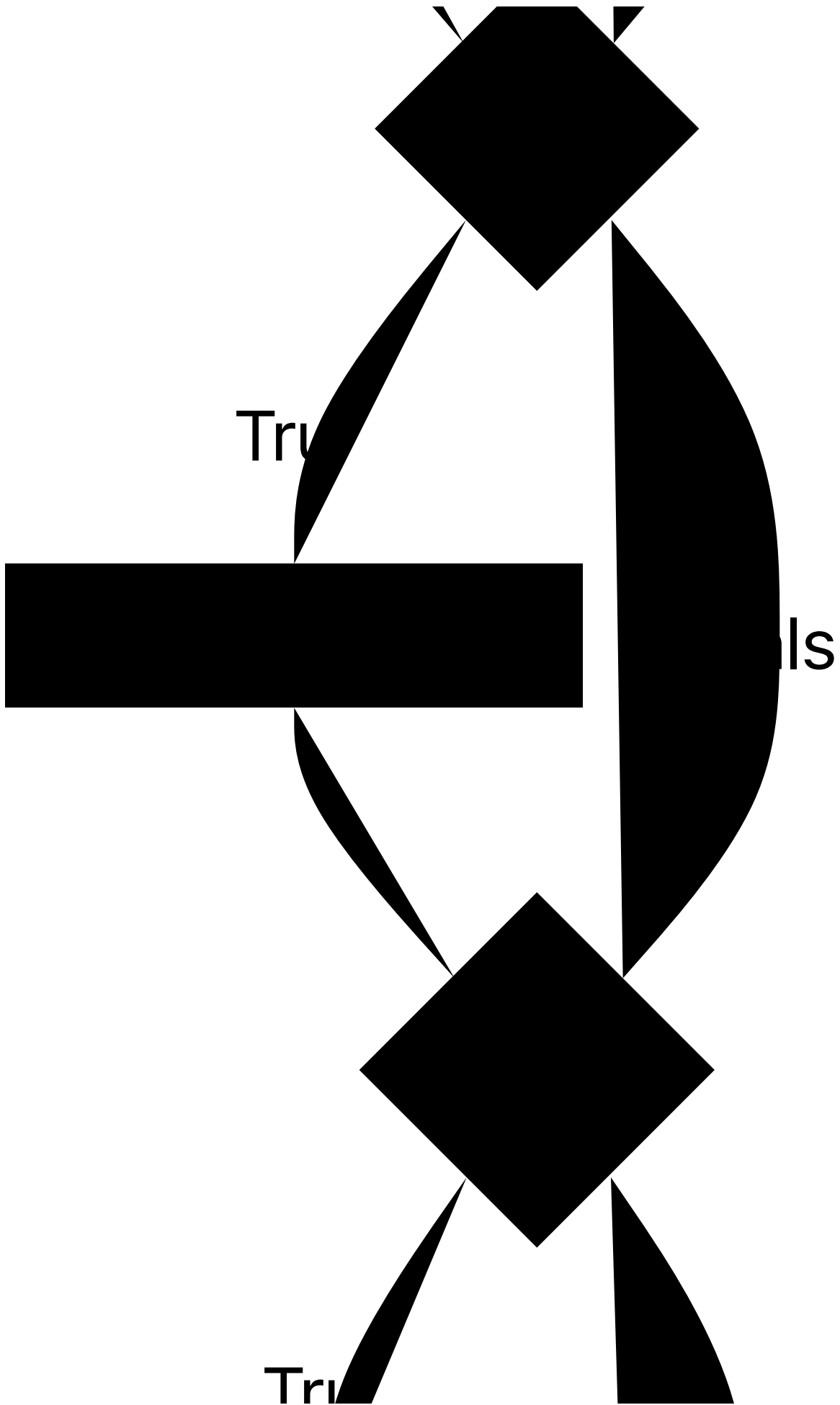
```
x = int(input("What's x? "))
y = int(input("What's y? "))

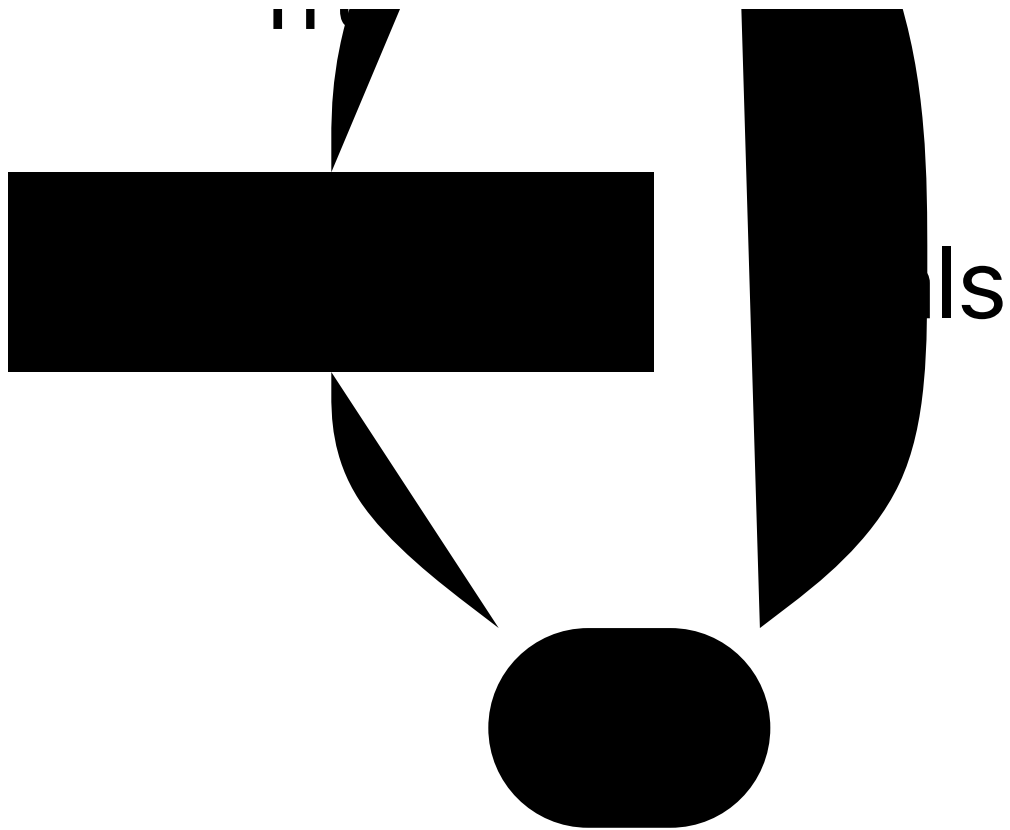
if x < y:
    print("x is less than y")
if x > y:
    print("x is greater than y")
if x == y:
    print("x is equal to y")
```

Notice how you are providing a series of `if` statements. First, the first `if` statement is evaluated. Then, the second `if` statement runs its evaluation. Finally, the last `if` statement runs its evaluation. This flow of decisions is called "control flow."

Our code can be represented as follows:







This program can be improved by not asking three consecutive questions. After all, not all three questions can have an outcome of true! Revise your program as follows:

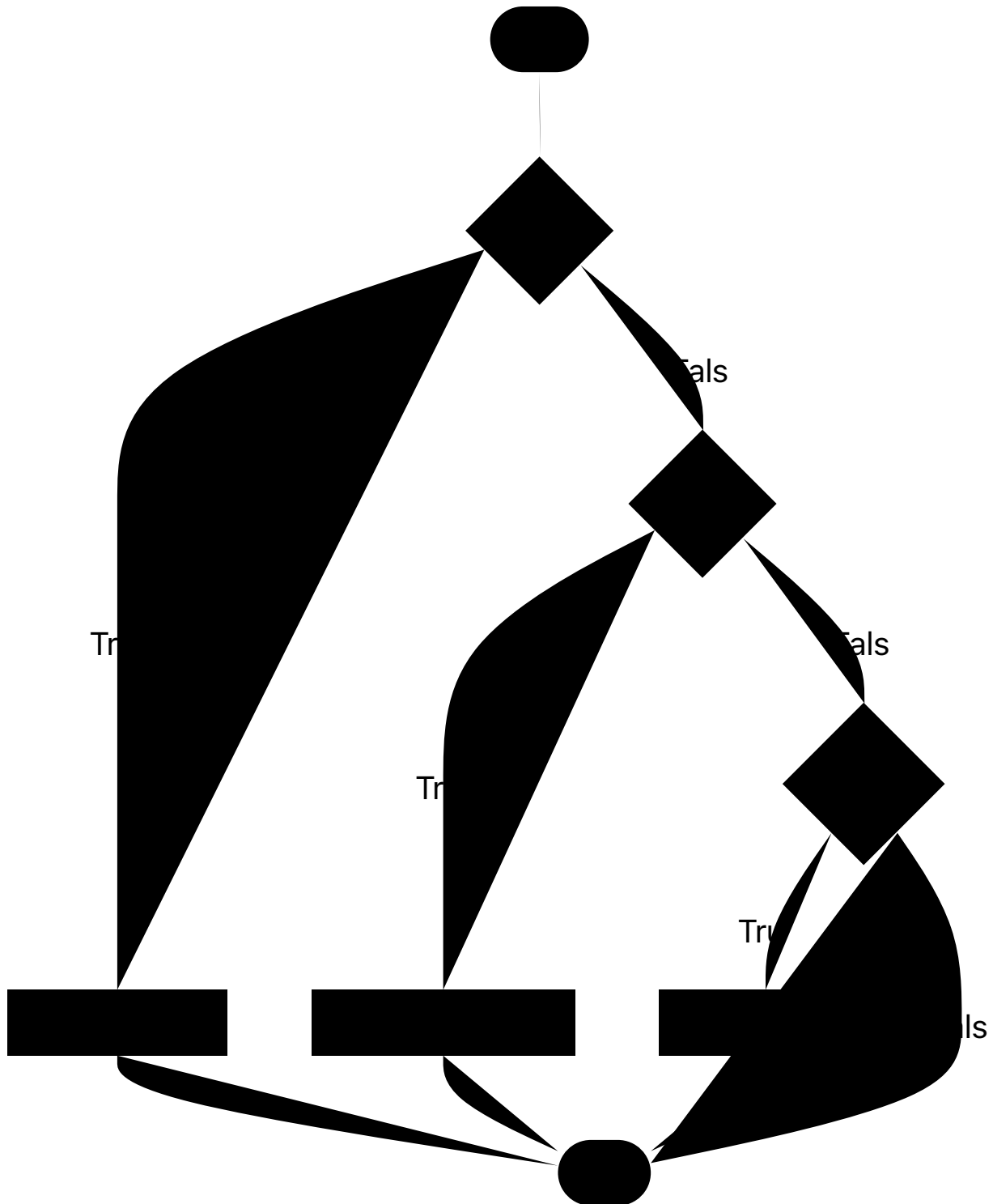
```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
elif x == y:
    print("x is equal to y")
```

Notice how the use of `elif` allows the program to make less decisions. First, the `if` statement is evaluated. If this statement is found to be true, all the `elif` statements not be run at all. However, if

the `if` statement is evaluated and found to be false, the first `elif` will be evaluated. If this is true, it will not run the final evaluation.

Our code can be represented as follows:



While your computer may not notice a difference speed-wise between our first program and this revised program, consider how an

online server running billions or trillions of these types of calculations each day could definitely be impacted by such a small coding decision.

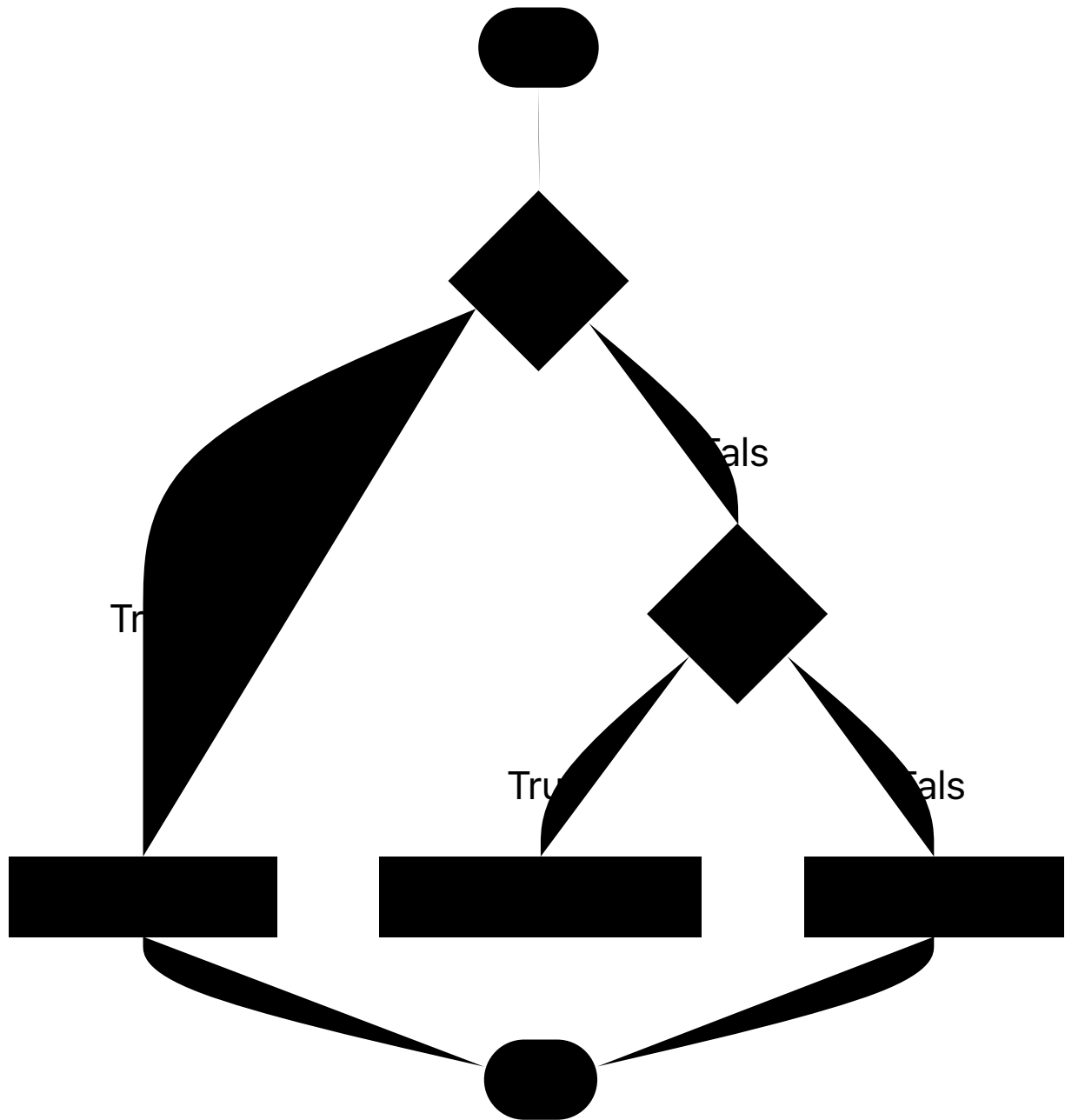
There is one final improvement we can make to our program. Notice how logically `elif x == y` is not a necessary evaluation to run. After all, if logically `x` is not less than `y` AND `x` is not greater than `y`, `x` MUST equal `y`. Therefore, we don't have to run `elif x == y`. We can create a "catch-all," default outcome using an `else` statement. We can revise as follows:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

Notice how the relative complexity of this program has decreased through our revision.

Our code can be represented as follows:



or allows your program to decide between one or more alternatives.
For example, we could further edit our program as follows:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x < y or x > y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

Notice that the result of our program is the same, but the complexity is decreased and the efficiency of our code is increased.

At this point, our code is pretty great. However, could the design be further improved? We could further edit our code as follows:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x != y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

Notice how we removed the or entirely, and simply asked "is x not equal to y?" We ask one and only one question. Very efficient!

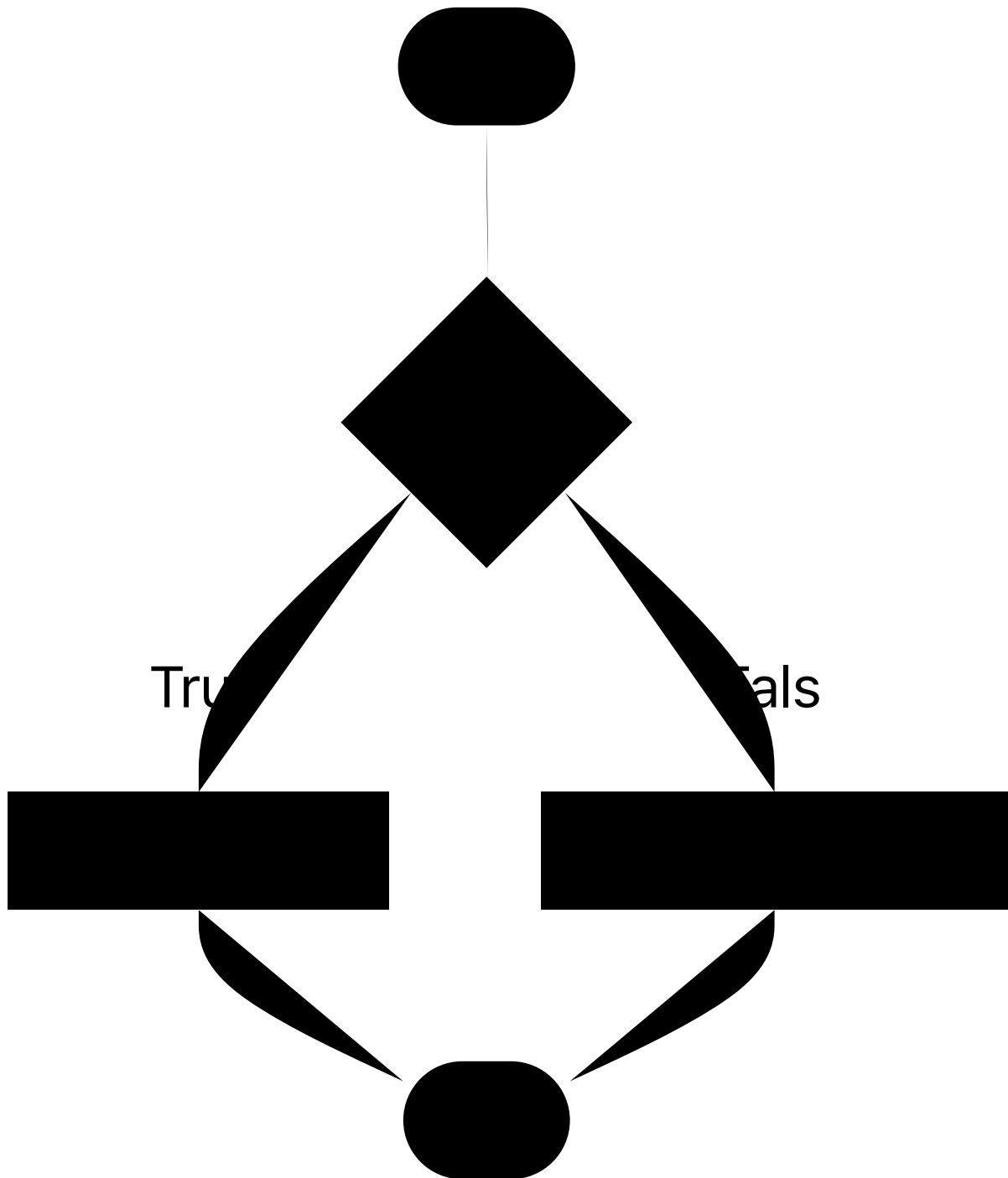
For the purpose of illustration, we could also change our code as follows:

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x == y:
    print("x is equal to y")
else:
    print("x is not equal to y")
```

Notice that the == operator evaluates if what is on the left and right are equal to one another. That use of double equal signs is very important. If you use only one equal sign, an error will likely be thrown by the compiler.

Our code can be illustrated as follows:



Similar to `or`, and can be used within conditional statements.

Execute in the terminal window code `grade.py`. Start your new program as follows:

```
score = int(input("Score: "))

if score >= 90 and score <= 100:
    print("Grade: A")
elif score >=80 and score < 90:
    print("Grade: B")
elif score >=70 and score < 80:
    print("Grade: C")
elif score >=60 and score < 70:
    print("Grade: D")
else:
    print("Grade: F")
```

Notice that executing `python grade.py` you will be able to input a score and get a grade. However, notice how there are potentials for bugs.

Typically, we do not want to ever trust our user to input the correct information. We could improve our code as follows:

```
score = int(input("Score: "))

if 90 <= score <= 100:
    print("Grade: A")
elif 80 <= score < 90:
    print("Grade: B")
elif 70 <= score < 80:
    print("Grade: C")
elif 60 <= score < 70:
    print("Grade: D")
else:
    print("Grade: F")
```

Notice how Python allows you to chain together the operators and

conditions in a way quite uncommon to other programming languages.

Still, we can further improve our program:

```
score = int(input("Score: "))

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

Notice how the program is improved by asking fewer questions. This makes our program easier to read and far more maintainable in the future.

You can learn more in Python's documentation on [control flow](#).

In mathematics, parity refers to whether a number is either even or odd.

The modulo % operator in programming allows one to see if two numbers divide evenly or divide and have a remainder.

For example, $4 \% 2$ would result in zero, because it evenly divides.

However, $3 \% 2$ does not divide evenly and would result in a number other than zero!

In the terminal window, create a new program by typing code parity.py. In the text editor window, type your code as follows:

```
x = int(input("What's x? "))
```

```
if x % 2 == 0:  
    print("Even")  
else:  
    print("Odd")
```

Notice how our users can type in any number 1 or greater to see if it is even or odd.

As discussed in Lecture 0, you will find it useful to create a function of your own!

We can create our own function to check whether a number is even or odd. Adjust your code as follows:

```
def main():  
    x = int(input("What's x? "))  
    if is_even(x):  
        print("Even")  
    else:  
        print("Odd")
```

```
def is_even(n):  
    if n % 2 == 0:  
        return True  
    else:  
        return False
```

```
main()
```

Notice that one reason our `if` statement `is_even(x)` works, even

though there is no operator there. This is because our function returns a `bool` (or boolean), `true` or `false`, back to the main function. The `if` statement simply evaluates whether or not `is_even` of `x` is `true` or `false`.

In the programming world, there are types of programming that are called “Pythonic” in nature. That is, there are ways to program that are sometimes only seen in Python programming. Consider the following revision to our program:

```
def main():
    x = int(input("What's x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")

def is_even(n):
    return True if n % 2 == 0 else False

main()
```

Notice that this `return` statement in our code is almost like a sentence in English. This is a unique way of coding only seen in Python.

We can further revise our code and make it more and more readable:

```
def main():
    x = int(input("What's x? "))
    if is_even(x):
        print("Even")
    else:
```

```
print("Odd")
```

```
def is_even(n):  
    return n % 2 == 0
```

```
main()
```

Notice that the program will evaluate what is happening within the `n % 2 == 0` as either true or false and simply return that to the main function.

match

Similar to `if`, `elif`, and `else` statements, `match` statements can be used to conditionally run code that matches certain values.

Consider the following program:

```
name = input("What's your name? ")  
  
if name == "Harry":  
    print("Gryffindor")  
elif name == "Hermione":  
    print("Gryffindor")  
elif name == "Ron":  
    print("Gryffindor")  
elif name == "Draco":  
    print("Slytherin")  
else:  
    print("Who?")
```

Notice the first three conditional statements print the same response.

We can improve this code slightly with the use of the `or` keyword:

```
name = input("What's your name? ")

if name == "Harry" or name == "Hermione" or name == "Ron":
    print("Gryffindor")
elif name == "Draco":
    print("Slytherin")
else:
    print("Who?")
```

Notice the number of `elif` statements has decreased, improving the readability of our code.

Alternatively, we can use `match` statements to map names to houses. Consider the following code:

```
name = input("What's your name? ")

match name:
    case "Harry":
        print("Gryffindor")
    case "Hermione":
        print("Gryffindor")
    case "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

Notice the use of the `_` symbol in the last case. This will match with any input, resulting in similar behavior as an `else` statement.

A `match` statement compares the value following the `match` keyword

with each of the values following the case keywords. In the event a match is found, the respective indented code section is executed and the program stops the matching.

We can improve the code:

```
name = input("What's your name? ")

match name:
    case "Harry" | "Hermione" | "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

Notice, the use of the single vertical bar |. Much like the or keyword, this allows us to check for multiple values in the same case statement.

You now have the power within Python to use conditional statements to ask questions and have your program take action accordingly. In this lecture, we discussed...

Conditionals;

if Statements;

Control flow, elif, and else;

or;

and;

Modulo;

Creating your own function;

Pythonic coding;

and match.

Lecture 7

- Regular Expressions
- Case Sensitivity
- Cleaning Up User Input
- Extracting User Input
- Summing Up

Regular expressions or “regexes” will enable us to examine patterns within our code. For example, we might want to validate that an email address is formatted correctly. Regular expressions will enable us to examine expressions in this fashion.

To begin, type code `validate.py` in the terminal window. Then, code as follows in the text editor:

```
email = input("What's your email? ").strip()

if "@" in email:
    print("Valid")
else:
    print("Invalid")
```

Notice that `strip` will remove whitespace at the beginning or end of the input. Running this program, you will see that as long as an `@` symbol is inputted, the program will regard the input as valid.

You can imagine, however, that one could input `@@` alone and the input could be regarded as valid. We could regard an email address as having at least one `@` and a `.` somewhere within it. Modify your code as follows:

```
email = input("What's your email? ").strip()
```

```
if "@" in email and "." in email:
    print("Valid")
else:
    print("Invalid")
```

Notice that while this works as expected, our user could be adversarial, typing simply @. would result in the program returning valid.

We can improve the logic of our program as follows:

```
email = input("What's your email? ").strip()

username, domain = email.split("@")

if username and "." in domain:
    print("Valid")
else:
    print("Invalid")
```

Notice how the `strip` method is used to determine if username exists and if `.` is inside the domain variable. Running this program, a standard email address typed in by you could be considered valid. Typing in `malan@harvard` alone, you'll find that the program regards this input as invalid.

We can be even more precise, modifying our code as follows:

```
email = input("What's your email? ").strip()

username, domain = email.split("@")

if username and domain.endswith(".edu"):
    print("Valid")
else:
```

```
print("Invalid")
```

Notice how the `endswith` method will check to see if domain contains `.edu`. Still, however, a nefarious user could still break our code. For example, a user could type in `malan@.edu` and it would be considered valid.

Indeed, we could keep iterating upon this code ourselves. However, it turns out that Python has an existing library called `re` that has a number of built-in functions that can validate user inputs against patterns.

One of the most versatile functions within the library `re` is `search`. The search library follows the signature `re.search(pattern, string, flags=0)`. Following this signature, we can modify our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search("@", email):
    print("Valid")
else:
    print("Invalid")
```

Notice this does not increase the functionality of our program at all. In fact, it is somewhat a step back.

We can further our program's functionality. However, we need to advance our vocabulary around `validation`. It turns out that in the world of regular expressions there are certain symbols that allow us to identify patterns. At this point, we have only been checking for

specific pieces of text like @. It so happens that many special symbols can be passed to the compiler for the purpose of engaging in validation. A non-exhaustive list of those patterns is as follows:

- . any character except a new line
- * 0 or more repetitions
- + 1 or more repetitions
- ? 0 or 1 repetition
- {m} m repetitions
- {m,n} m-n repetitions

Implementing this inside of our code, modify yours as follows:

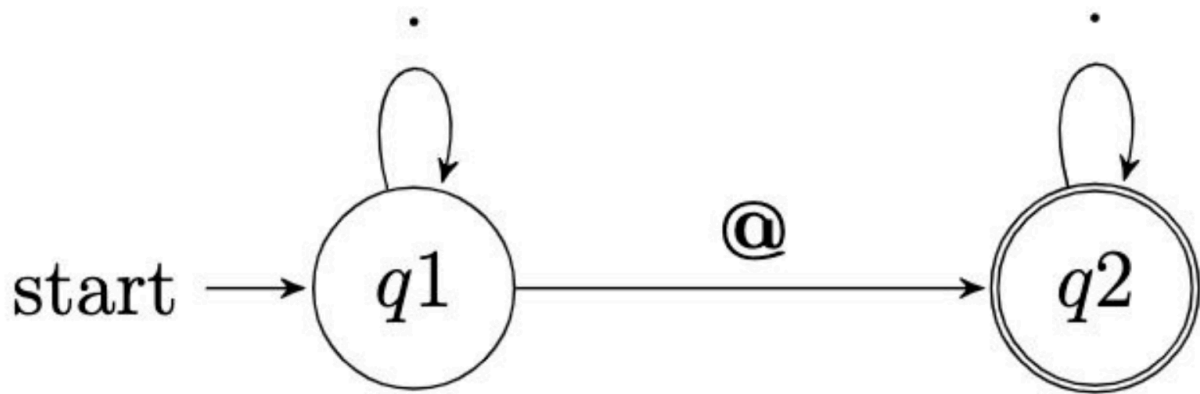
```
import re

email = input("What's your email? ").strip()

if re.search(".*@.*", email):
    print("Valid")
else:
    print("Invalid")
```

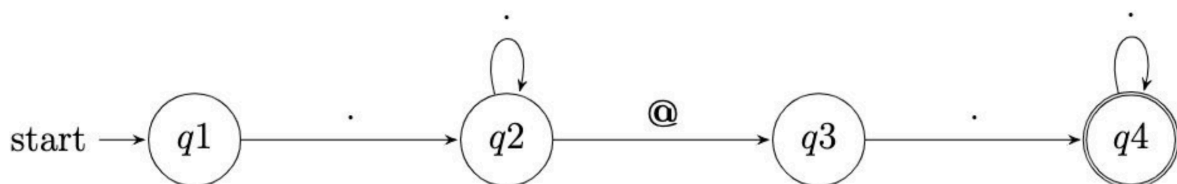
Notice that we don't care what the username or domain is. What we care about is the pattern. .* is used to determine if anything is to the left of the email address and if anything is to the right of the email address. Running your code, typing in `malan@`, you'll notice that the input is regarded as `invalid` as we would hope.

Had we used a regular expression `.*@.*` in our code above, you can visualize this as follows:



Notice the depiction of the state machine of our regular expression. On the left, the compiler begins evaluating the statement from left to right. Once we reach q_1 or question 1, the compiler reads time and time again based on the expression handed to it. Then, the state is changed looking now at q_2 or the second question being validated. Again, the arrow indicates how the expression will be evaluated time and time again based upon our programming. Then, as depicted by the double circle, the final state of state machine is reached.

Considering the regular expression we used in our code, $.+@.+$, you can visualize it as follows:



Notice how q_1 is any character provided by the user, including ' q_2 ' as 1 or more repetitions of characters. This is followed by the '@' symbol. Then, q_3 looks for any character provided by the user,

including q4 as 1 or more repetitions of characters.

The `re` and `re.search` functions and ones like them look for patterns. Continuing our improvement of this code, we could improve our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(".*@.*.edu", email):
    print("Valid")
else:
    print("Invalid")
```

Notice, however, that one could type in `ma lan@harvard?edu` and it could be considered valid. Why is this the case? You might recognize that in the language of validation, a `.` means any character!

We can modify our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(".*@.*\\.edu", email):
    print("Valid")
else:
    print("Invalid")
```

Notice how we utilize the “escape character” or `\` as a way of regarding the `.` as part of our string instead of our validation expression. Testing your code, you will notice that `ma lan@harvard.edu` is regarded as valid, where `ma lan@harvard?edu` is invalid.

Now that we're using escape characters, it's a good time to introduce "raw strings". In Python, raw strings are strings that *don't* format special characters—instead, each character is taken at face-value. Imagine `\n`, for example. We've seen in an earlier lecture how, in a regular string, these two characters become one: a special newline character. In a raw string, however, `\n` is treated not as `\n`, the special character, but as a single `\` and a single `n`. Placing an `r` in front of a string tells the Python interpreter to treat the string as a raw string, similar to how placing an `f` in front of a string tells the Python interpreter to treat the string as a format string:

```
import re

email = input("What's your email? ").strip()

if re.search(r".+@.+\.edu", email):
    print("Valid")
else:
    print("Invalid")
```

Now we've ensured the Python interpreter won't treat `\.` as a special character. Instead, simply as a `\` followed by a `.`—which, in regular expression terms, means matching a literal `"."`.

You can imagine still how our users could create problems for us! For example, you could type in a sentence such as `My email address is malan@harvard.edu` and this whole sentence would be considered valid. We can be even more precise in our coding.

It just so happens we have more special symbols at our disposal in validation:

```
^ matches the start of the string
$ matches the end of the string or just before the newline at the er
```

We can modify our code using our added vocabulary as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^.+@.+\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

Notice this has the effect of looking for this exact pattern matching to the start and end of the expression being validated. Typing in a sentence such as My email address is malan@harvard.edu now is regarded as invalid.

We propose we can do even better! Even though we are now looking for the username at the start of the string, the @ symbol, and the domain name at the end, we could type in as many @ symbols as we wish! malan@@@harvard.edu is considered valid!

We can add to our vocabulary as follows:

```
[]    set of characters
[^]   complementing the set
```

Using these newfound abilities, we can modify our expression as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^^[^@]+@[^@]+\\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

Notice that `^` means to match at the start of the string. All the way at the end of our expression, `$` means to match at the end of the string. `[^@]+` means any character except an `@`. Then, we have a literal `@`. `[^@]+\.` means any character except an `@` followed by an expression ending in `.`. Typing in `ma lan@@@harvard.edu` is now regarded as invalid.

We can still improve this regular expression further. It turns out there are certain requirements for what an email address can be! Currently, our validation expression is far too accomodating. We might only want to allow for characters normally used in a sentence. We can modify our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^[a-zA-Z0-9_]+@[a-zA-Z0-9_]+\.
```

Notice that `[a-zA-Z0-9_]` tells the validation that characters must be between `a` and `z`, between `A` and `Z`, between `0` and `9` and potentially include an `_` symbol. Testing the input, you'll find that many potential user mistakes can be indicated.

Thankfully, common patterns have been built into regular expressions by hard-working programmers. In this case, you can modify your code as follows:

```
import re
```

```
email = input("What's your email? ").strip()

if re.search(r"^\w+@\w+\.\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

Notice that `\w` is the same as `[a-zA-Z0-9_]`. Thanks, hard-working programmers!

Here are some additional patterns we can add to our vocabulary:

```
\d    decimal digit
\D    not a decimal digit
\s    whitespace characters
\S    not a whitespace character
\w    word character, as well as numbers and the underscore
\W    not a word character
```

Now, we know that there are not simply `.edu` email addresses. We could modify our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^\w+@\w+\.\.(com|edu|gov|net|org)$", email):
    print("Valid")
else:
    print("Invalid")
```

Notice that the `|` has the impact of an `or` in our expression.

Adding even more symbols to our vocabulary, here are some more to consider:

A|B either A or B
(...) a group
(?:...) non-capturing version

To illustrate how you might address issues around case sensitivity, where there is a difference between EDU and edu and the like, let's rewind our code to the following:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^\w+@\w+\.\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

Notice how we have removed the | statements provided previously.

Recall that within the re.search function, there is a parameter for flags.

Some built-in flag variables are:

```
re.IGNORECASE
re.MULTILINE
re.DOTALL
```

Consider how you might use these in your code.

Therefore, we can change our code as follows.

```
import re

email = input("What's your email? ").strip()
```

```
if re.search(r"^w+@w+\.edu$", email, re.IGNORECASE):
    print("Valid")
else:
    print("Invalid")
```

Notice how we added a third parameter `re.IGNORECASE`. Running this program with `MALAN@HARVARD.EDU`, the input is now considered valid.

Consider the following email address `malan@cs50.harvard.edu`. Using our code above, this would be considered invalid. Why might that be?

Since there is an additional `.`, the program considers this invalid. It turns out that we can, looking at our vocabulary from before, we can group together ideas.

`A|B` either A or B
`(...)` a group
`(?:...)` non-capturing version

We can modify our code as follows:

```
import re

email = input("What's your email? ").strip()

if re.search(r"^w+@(\w+\.)?\w+\.edu$", email, re.IGNORECASE):
    print("Valid")
else:
    print("Invalid")
```

Notice how the `(\w+\.)?` communicates to the compiler that this new expression can be there once or not at all. Hence, both `malan@cs50.harvard.edu` and `malan@harvard.edu` are considered valid.

Interestingly enough, the edits we have done so far to our code do not fully encompass all the checking that could be done to ensure a valid email address. Indeed, here is the full expression that one would have to type to ensure that a valid email is inputted:

```
^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-
```

There are other functions within the `re` library you might find useful. `re.match` and `re.fullmatch` are ones you might find exceedingly useful.

You can learn more in Python's documentation of [re](#).

You should never expect your users to always follow your hopes for clean input. Indeed, users will often violate your intentions as a programmer.

There are ways to clean up your data.

In the terminal window, type `code format.py`. Then, in the text-editor, code as follows:

```
name = input("What's your name? ").strip()
print(f"hello, {name}")
```

Notice that we have created, essentially, a “hello world” program. Running this program and typing in David, it works well! However, typing in Malan, David notice how the program does not function as intended. How could we modify our program to clean up this input?

Modify your code as follows.

```
name = input("What's your name? ").strip()
if "," in name:
    last, first = name.split(", ")
```

```
name = f"{first} {last}"

print(f"hello, {name}")
```

Notice how `last, first = name.split(", ")` is run if there is a `,` in the name. Then, the name is standardized as first and last. Running our code, typing in `Malan, David`, you can see how this program does clean up at least one scenario where a user types in something unexpected.

You might notice that typing in `Malan,David` with no space causes the compiler to throw an error. Since we now know some regular expression syntax, let's apply that to our code:

```
import re

name = input("What's your name? ").strip()
matches = re.search(r"^(.+), (.+)$", name)
if matches:
    last, first = matches.groups()
    name = f"{first} {last}"
print(f"hello, {name}")
```

Notice that `re.search` can return a set of matches that are extracted from the user's input. If matches are returned by `re.search`. Running this program, typing in `David Malan` notice how the `if` condition is not run and the name is returned. If you run the program by typing `Malan, David`, the name is also returned properly.

It just so happens that we can request specific groups back using `matches.group`. We can modify our code as follows:

```
import re
```

```

name = input("What's your name? ").strip()
matches = re.search(r"^(.+), (.+)$", name)
if matches:
    last = matches.group(1)
    first = matches.group(2)
    name = f"{first} {last}"
print(f"hello, {name}")

```

Notice how, in this implementation, group is not plural (there is no s).

Our code can be further tightened as follows:

```

import re

name = input("What's your name? ").strip()
matches = re.search(r"^(.+), (.+)$", name)
if matches:
    name = matches.group(2) + " " + matches.group(1)
print(f"hello, {name}")

```

Notice how group(2) and group(1) are concatenated together with a space. The first group is that which is left of the comma. The second group is that which is right of the comma.

Recognize still that typing in Malan,David with no space will still break our code. Therefore, we can make the following modification:

```

import re

name = input("What's your name? ").strip()
matches = re.search(r"^(.+), *(.+)$", name)
if matches:
    name = matches.group(2) + " " + matches.group(1)
print(f"hello, {name}")

```

Notice the addition of the `*` in our validation statement. This code will now accept and properly process `Malan,David`. Further, it will properly handle `` David,Malan` with many spaces in front of `David``.

It is very common to utilize `re.search` as we have in the previous examples, where `matches` is on a line of code after. However, we can combine these statements:

```
import re

name = input("What's your name? ").strip()
if matches := re.search(r"^(.+), *(.+)$", name):
    name = matches.group(2) + " " + matches.group(1)
print(f"hello, {name}")
```

Notice how we combine two lines of our code. The walrus `:=` operator assigns a value from right to left and allows us to ask a boolean question at the same time. Turn your head sideways and you'll see why this is called a walrus operator.

You can learn more in Python's documentation of [re](#).

So far, we have validated the user's input and cleaned up the user's input.

Now, let's extract some specific information from user input. In the terminal window, type `code twitter.py` and code as follows in the text editor window:

```
url = input("URL: ").strip()
print(url)
```

Notice that if we type in `https://twitter.com/davidjmalan`, it shows exactly what the user typed. However, how would we be able to

extract just the username and ignore the rest of the URL?

You can imagine how we would simply be able to get rid of the beginning of the standard Twitter URL. We can attempt this as follows:

```
url = input("URL: ").strip()

username = url.replace("https://twitter.com/", "")
print(f"Username: {username}")
```

Notice how the `replace` method allows us to find one item and replace it with another. In this case, we are finding part of the URL and replacing it with nothing. Typing in the full URL `https://twitter.com/davidjmalan`, the program effectively outputs the username. However, what are some shortcomings of this current program?

What if the user simply typed `twitter.com` instead of including the `https://` and the like? You can imagine many scenarios where the user may input or neglect to input parts of the URL that would create strange output by this program. To improve this program, we can code as follows:

```
url = input("URL: ").strip()

username = url.removeprefix("https://twitter.com/")
print(f"Username: {username}")
```

Notice how we utilize the `removeprefix` method. This method will remove the beginning of a string.

Regular expressions simply allow us to succinctly express the patterns and goals.

Within the `re` library, there is a method called `sub`. This method allows us to substitute a pattern with something else. The signature of the `sub` method is as follows

```
re.sub(pattern, repl, string, count=0, flags=0)
```

Notice how `pattern` refers to the regular expression we are looking for. Then, there is a `repl` string that we can replace the pattern with. Finally, there is the `string` that we want to do the substitution on.

Implementing this method in our code, we can modify our program as follows:

```
import re

url = input("URL: ").strip()

username = re.sub(r"https://twitter.com/", "", url)
print(f"Username: {username}")
```

Notice how executing this program and inputting `https://twitter.com/davidjmalan` produces the correct outcome. However, there are some problems still present in our code.

The protocol, subdomain, and the possibility that the user inputted any part of the URL after the username are all reasons that this code is still not ideal. We can further address these shortcomings as follows:

```
import re

url = input("URL: ").strip()

username = re.sub(r"^(https?://)?(www\.)?twitter\.com/", "", url)
print(f"Username: {username}")
```

Notice how the ^ caret was added to the url. Notice also how the . could be interpreted improperly by the compiler. Therefore, we escape it using a \ to make it \. For the purpose of tolerating both http and https, we add a ? to the end of https?, making the s optional. Further, to accommodate www we add (www\.)? to our code. Finally, just in case the user decides to leave out the protocol altogether, the http:// or https:// is made optional using (https?://).

Still, we are blindly expecting that what the user inputted a url that, indeed, has a username.

Using our knowledge of re.search, we can further improve our code.

```
import re

url = input("URL: ").strip()

matches = re.search(r"^https?://(www\.)?twitter\.com/(.+)$", url, re.I)
if matches:
    print(f"Username:", matches.group(2))
```

Notice how we are searching for the regular expression above in the string provided by the user. In particular, we are capturing that which appears at the end of the URL using (.+)\$ regular expression.

Therefore, if the user fails to input a URL without a username, no input will be presented.

Even further tightening up our program, we can utilize our := operator as follows:

```
import re

url = input("URL: ").strip()
```

```
if matches := re.search(r"^https?:/(?:www\.)?twitter\.com/(.+)$", url):
    print(f"Username:", matches.group(1))
```

Notice that the `?:` tells the compiler it does not have to capture what is in that spot in our regular expression.

Still, we can be more explicit to ensure that the username inputted is correct. Using Twitter's documentation, we can add the following to our regular expression:

```
import re

url = input("URL: ").strip()

if matches := re.search(r"^https?:/(?:www\.)?twitter\.com/([a-z0-9_]+)", url):
    print(f"Username:", matches.group(1))
```

Notice that the `[a-z0-9_]+` tells the compiler to only expect `a-z`, `0-9`, and `_` as part of the regular expression. The `+` indicates that we are expecting one or more characters.

You can learn more in Python's documentation of [re](#).

Now, you've learned a whole new language of regular expressions that can be utilized to validate, clean up, and extract user input.

Regular Expressions

Case Sensitivity

Cleaning Up User Input

Extracting User Input

Lecture 6

- File I/O
- open
- with
- CSV
- Binary Files and PIL
- Summing Up

Up until now, everything we've programmed has stored information in memory. That is, once the program is ended, all information gathered from the user or generated by the program is lost.

File I/O is the ability of a program to take a file as input or create a file as output.

To begin, in the terminal window type `code names.py` and code as follows:

```
name = input("What's your name?" )  
print(f"hello, {name}")
```

Notice that running this code has the desired output. The user can input a name. The output is as expected.

However, what if we wanted to allow multiple names to be inputted? How might we achieve this? Recall that a `list` is a data structure that allows us to store multiple values into a single variable. Code as follows:

```
names = []
```

```
for _ in range(3):
```

```
name = input("What's your name?" )
names.append(name)
```

Notice that the user will be prompted three times for input. The `append` method is used to add the name to our names list.

This code could be simplified to the following:

```
names = []

for _ in range(3):
    names.append(input("What's your name?" ))
```

Notice that this has the same result as the prior block of code.

Now, let's enable the ability to print the list of names as a sorted list. Code as follows:

```
names = []

for _ in range(3):
    names.append(input("What's your name?" ))

for name in sorted(names):
    print(f"hello, {name}")
```

Notice that once this program is executed, all information is lost. File I/O allows your program to store this information such that it can be used later.

You can learn more in Python's documentation of [sorted](#).

open

open is a functionality built into Python that allows you to open a file and utilize it in your program. The open function allows you to open a file such that you can read from it or write to it.

To show you how to enable file I/O in your program, let's rewind a bit and code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "w")
file.write(name)
file.close()
```

Notice that the open function opens a file called `names.txt` with writing enabled, as signified by the `w`. The code above assigns that opened file to a variable called `file`. The line `file.write(name)` writes the name to the text file. The line after that closes the file.

Testing out your code by typing `python names.py`, you can input a name and it saves to the text file. However, if you run your program multiple times using different names, you will notice that this program will entirely rewrite the `names.txt` file each time.

Ideally, we want to be able to append each of our names to the file. Remove the existing text file by typing `rm names.txt` in the terminal window. Then, modify your code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "a")
file.write(name)
file.close()
```

Notice that the only change to our code is that the `w` has been changed to `a` for "append". Rerunning this program multiple times, you will notice that names will be added to the file. However, you will notice a new problem!

Examining your text file after running your program multiple times, you'll notice that the names are running together. The names are being appended without any gaps between each of the names. You can fix this issue. Again, remove the existing text file by typing `rm names.txt` in the terminal window. Then, modify your code as follows:

```
name = input("What's your name? ")

file = open("names.txt", "a")
file.write(f"{name}\n")
file.close()
```

Notice that the line with `file.write` has been modified to add a line break at the end of each name.

This code is working quite well. However, there are ways to improve this program. It so happens that it's quite easy to forget to close the file.

You can learn more in Python's documentation of [open](#).

with

The keyword `with` allows you to automate the closing of a file.

Modify your code as follows:

```
name = input("What's your name? ")
```

```
with open("names.txt", "a") as file:  
    file.write(f"{name}\n")
```

Notice that the line below `with` is indented.

Up until this point, we have been exclusively writing to a file. What if we want to read from a file. To enable this functionality, modify your code as follows:

```
with open("names.txt", "r") as file:  
    lines = file.readlines()  
  
for line in lines:  
    print("hello,", line)
```

Notice that `readlines` has a special ability to read all the lines of a file and store them in a file called `lines`. Running your program, you will notice that the output is quite ugly. There seem to be multiple line breaks where there should be only one.

There are many approaches to fix this issue. However, here is a simple way to fix this error in our code:

```
with open("names.txt", "r") as file:  
    lines = file.readlines()  
  
for line in lines:  
    print("hello,", line.rstrip())
```

Notice that `rstrip` has the effect of removing the extraneous line break at the end of each line.

Still, this code could be simplified even further:

```
with open("names.txt", "r") as file:
    for line in file:
        print("hello,", line.rstrip())
```

Notice that running this code, it is correct. However, notice that we are not sorting the names.

This code could be further improved to allow for the sorting of the names:

```
names = []

with open("names.txt") as file:
    for line in file:
        names.append(line.rstrip())

for name in sorted(names):
    print(f"hello, {name}")
```

Notice that names is a blank list where we can collect the names. Each name is appended to the names list in memory. Then, each name in the sorted list in memory is printed. Running your code, you will see that the names are now properly sorted.

What if we wanted the ability to store more than just the names of students? What if we wanted to store both the student's name and their house as well?

CSV stands for "comma separated values".

In your terminal window, type `code students.csv`. Ensure your new

CSV file looks like the following:

```
Hermoine,Gryffindor  
Harry,Gryffindor  
Ron,Gryffindor  
Draco,Slytherin
```

Let's create a new program by typing code `students.py` and code as follows:

```
with open("students.csv") as file:  
    for line in file:  
        row = line.rstrip().split(",")  
        print(f"{row[0]} is in {row[1]}")
```

Notice that `rstrip` removes the end of each line in our CSV file. `split` tells the compiler where to find the end of each of our values in our CSV file. `row[0]` is the first element in each line of our CSV file. `row[1]` is the second element in each line in our CSV file.

The above code is effective at dividing each line or "record" of our CSV file. However, it's a bit cryptic to look at if you are unfamiliar with this type of syntax. Python has built-in ability that could further simplify this code. Modify your code as follows:

```
with open("students.csv") as file:  
    for line in file:  
        name, house = line.rstrip().split(",")  
        print(f"{name} is in {house}")
```

Notice that the `split` function actually returns two values: The one before the comma and the one after the comma. Accordingly, we can

rely upon that functionality to assign two variables at once instead of one!

Imagine that we would again like to provide this list as sorted output? You can modify your code as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append(f"{name} is in {house}")

for student in sorted(students):
    print(student)
```

Notice that we create a list called `students`. We append each string to this list. Then, we output a sorted version of our list.

Recall that Python allows for dictionaries where a key can be associated with a value. This code could be further improved

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {}
        student["name"] = name
        student["house"] = house
        students.append(student)

for student in students:
    print(f"{student['name']} is in {student['house']}")
```


Notice that we create an empty dictionary called `student`. We add the values for each student, including their name and house into the `student` dictionary. Then, we append that student to the list called `students`.

We can improve our code to illustrate this as follows:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name": name, "house": house}
        students.append(student)

for student in students:
    print(f"{student['name']} is in {student['house']}")
```

Notice that this produces the desired outcome, minus the sorting of students.

Unfortunately, we cannot sort the students as we had prior because each student is now a dictionary inside of a list. It would be helpful if Python could sort the `students` list of student dictionaries that sorts this list of dictionaries by the student's name.

To implement this in our code, make the following changes:

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append({"name": name, "house": house})
```

```
def get_name(student):  
    return student["name"]  
  
for student in sorted(students, key=get_name):  
    print(f"{student['name']} is in {student['house']}")
```

Notice that `sorted` needs to know how to get the key of each student. Python allows for a parameter called `key` where we can define on what “key” the list of students will be sorted. Therefore, the `get_name` function simply returns the key of `student["name"]`. Running this program, you will now see that the list is now sorted by name.

Still, our code can be further improved upon. It just so happens that if you are only going to use a function like `get_name` once, you can simplify your code in the manner presented below. Modify your code as follows:

```
students = []  
  
with open("students.csv") as file:  
    for line in file:  
        name, house = line.rstrip().split(",")  
        students.append({"name": name, "house": house})  
  
for student in sorted(students, key=lambda student: student["name"]):  
    print(f"{student['name']} is in {student['house']}")
```

Notice how we use a `lambda` function, an anonymous function, that says “Hey Python, here is a function that has no name: Given a

student, access their name and return that to the key.

Unfortunately, our code is a bit fragile. Suppose that we changed our CSV file such that we indicated where each student grew up. What would be the impact of this upon our program? First, modify your `students.csv` file as follows:

```
Harry,"Number Four, Privet Drive"  
Ron,The Burrow  
Draco,Malfoy Manor
```

Notice how running our program now will produce a number of errors.

Now that we're dealing with homes instead of houses, modify your code as follows:

```
students = []  
  
with open("students.csv") as file:  
    for line in file:  
        name, home = line.rstrip().split(",")  
        students.append({"name": name, "home": home})  
  
for student in sorted(students, key=lambda student: student["name"]):  
    print(f"{student['name']} is in {student['home']}")
```

Notice that running our program still does not work properly. Can you guess why?

The `ValueError: too many values to unpack` error produced by the compiler is a result of the fact that we previously created this program expecting the CSV file is split using a `,` (comma). We

could spend more time addressing this, but indeed someone else has already developed a way to “parse” (that is, to read) CSV files!

Python’s built-in `csv` library comes with an object called a reader. As the name suggests, we can use a reader to read our CSV file despite the extra comma in “Number Four, Privet Drive”. A reader works in a for loop, where each iteration the reader gives us another row from our CSV file. This row itself is a list, where each value in the list corresponds to an element in that row. `row[0]`, for example, is the first element of the given row, while `row[1]` is the second element.

```
import csv

students = []

with open("students.csv") as file:
    reader = csv.reader(file)
    for row in reader:
        students.append({"name": row[0], "home": row[1]})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is from {student['home']}")
```

Notice that our program now works as expected.

Up until this point, we have been relying upon our program to specifically decide what parts of our CSV file are the names and what parts are the homes. It’s better design, though, to bake this directly into our CSV file by editing it as follows:

```
name,home
Harry,"Number Four, Privet Drive"
Ron,The Burrow
Draco,Malfoy Manor
```

Notice how we are explicitly saying in our CSV file that anything reading it should expect there to be a name value and a home value in each line.

We can modify our code to use a part of the `csv` library called a `DictReader` to treat our CSV file with even more flexibility:

```
import csv

students = []

with open("students.csv") as file:
    reader = csv.DictReader(file)
    for row in reader:
        students.append({"name": row["name"], "home": row["home"]})

for student in sorted(students, key=lambda student: student["name"]):
    print(f"{student['name']} is in {student['home']}")
```

Notice that we have replaced `reader` with `DictReader`, which returns one dictionary at a time. Also, notice that the compiler will directly access the row dictionary, getting the name and home of each student. This is an example of coding defensively. As long as the person designing the CSV file has inputted the correct header information on the first line, we can access that information using our program.

Up until this point, we have been reading CSV files. What if we want to write to a CSV file?

To begin, let's clean up our files a bit. First, delete the `students.csv` file by typing `rm students.csv` in the terminal window. This command will only work if you're in the same folder as your `students.csv` file.

Then, in `students.py`, modify your code as follows:

```
import csv

name = input("What's your name? ")
home = input("Where's your home? ")

with open("students.csv", "a") as file:
    writer = csv.DictWriter(file, fieldnames=["name", "home"])
    writer.writerow({"name": name, "home": home})
```

Notice how we are leveraging the built-in functionality of `DictWriter`, which takes two parameters: the file being written to and the `fieldnames` to write. Further, notice how the `writerow` function takes a dictionary as its parameter. Quite literally, we are telling the compiler to write a row with two fields called `name` and `home`.

Note that there are many types of files that you can read from and write to.

You can learn more in Python's documentation of [CSV](#).

Binary Files and PIL

One more type of file that we will discuss today is a binary file. A binary file is simply a collection of ones and zeros. This type of file can store anything including, music and image data.

There is a popular Python library called `PIL` that works well with image files.

Animated GIFs are a popular type of image file that has many image files within it that are played in sequence over and over again, creating a simplistic animation or video effect.

Imagine that we have a series of costumes, as illustrated below. Here is `costume1.gif`.



Here is another one called `costume2.gif`. Notice how the leg positions are slightly different.



Before proceeding, please make sure that you have downloaded the source code files from the course website. It will not be possible for you to code the following without having the two images above in your possession and stored in your IDE.

In the terminal window type `code costumes.py` and code as follows:

```
import sys

from PIL import Image

images = []

for arg in sys.argv[1:]:
```

```
image = Image.open(arg)
images.append(image)

images[0].save(
    "costumes.gif", save_all=True, append_images=[images[1]], duration=
)
```

Notice that we import the Image functionality from PIL. Notice that the first for loop simply loops through the images provided as command-line arguments and stores them into the list called images. The 1: starts slicing argv at its second element. The last lines of code save the first image and also append a second image to it as well, creating an animated gif. Typing `python costumes.py costume1.gif costume2.gif` into the terminal. Now, type `costumes.gif` into the terminal window, and you can now see an animated GIF.

You can learn more in Pillow's documentation of [PIL](#).

Now, we have not only seen that we can write and read files textually—we can also read and write files using ones and zeros. We can't wait to see what you achieve with these new abilities next.

File I/O

open

with

CSV

PIL