Python by example:

```
import math


def say_hi(name):
    """<---- Multi-Line Comments and Docstrings
    This is where you put your content for help() to inform the user
    about what your function does and how to use it
    """
    print(f"Hello {name}!")


print(say_hi("Bryan"))  # Should get the print inside the function, then None
# Boolean Values
# Work the same as in JS, except they are title case: True and False
a = True
b = False
# Logical Operators
# ! = not, || = or, && = and
print(True and True)
print(True and not True)
print(True or True)
# Truthiness - Everything is True except...
# False - None, False, '', [], (), set(), range(0)
# Number Values
# Integers are numbers without a floating decimal point
print(type(3))  # type returns the type of whatever argument you pass in
# Floating Point values are numbers with a floating decimal point
print(type(3.5))
# Type Casting
# You can convert between ints and floats (along with other types...)
print(float(3))  # If you convert a float to an int, it will truncate the decimal
print(int(4.5))
print(type(str(3)))
# Python does not automatically convert types like JS
# print(17.0 + ' heyooo ' + 17)  # TypeError
# Arithmetic Operators
# ** - exponent (comparable to Math.pow(num, pow))
# // - integer division
# There is no ++ or -- in Python
# String Values
# We can use single quotes, double quotes, or f'' for string formats
# We can use triple single quotes for multiline strings
print(
    """This here's a story
All about how
My life got twist
Turned upside down
"""
```

```
)
# Three double quotes can also be used, but we typically reserve these for
# multi-line comments and function docstrings (refer to lines 6-9)(Nice :D)
# We use len() to get the length of something
print(len("Bryan G"))  # 7 characters
print(len(["hey", "ho", "hey", "hey", "ho"]))  # 5 list items
print(len({1, 2, 3, 4, 5, 6, 7, 9}))  # 8 set items
# We can index into strings, list, etc..self.
name = "Bryan"
for i in range(len(name)):
    print(name[i])  # B, r, y, a, n
# We can index starting from the end as well, with negatives
occupation = "Full Stack Software Engineer"
print(occupation[-3])  # e
# We can also get ranges in the index with the [start:stop:step] syntax
print(occupation[0:4:1])  # step and stop are optional, stop is exclusive
print(occupation[::4])  # beginning to end, every 4th letter
print(occupation[4:14:2])  # Let's get weird with it!
# NOTE: Indexing out of range will give you an IndexError
# We can also get the index og things with the .index() method, similar to
indexOf()
print(occupation.index("Stack"))
print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
# We can count how many times a substring/item appears in something as well
print(occupation.count("S"))
print(
    """Now this here's a story all about how
My life got twist turned upside down
I forget the rest but the the the potato
smells like the potato""".count(
        "the"
    )
)
# We concatenate the same as Javascript, but we can also multiply strings
print("dog " + "show")
print("ha" * 10)
# We can use format for a multitude of things, from spaces to decimal places
first_name = "Bryan"
last_name = "Guner"
print("Your name is {0} {1}".format(first_name, last_name))
# Useful String Methods
print("Hello".upper())  # HELLO
print("Hello".lower())  # hello
print("HELLO".islower())  # False
print("HELLO".isupper())  # True
print("Hello".startswith("he"))  # False
print("Hello".endswith("lo"))  # True
print("Hello There".split())  # [Hello, There]
print("hello1".isalpha())  # False,  must consist only of letters
print("hello1".isalnum())  # True, must consist of only letters and numbers
print("3215235123".isdecimal())  # True, must be all numbers
# True, must consist of only spaces/tabs/newlines
print("       \n     ".isspace())
# False, index 0 must be upper case and the rest lower
```

```
print("Bryan Guner".istitle())
print("Michael Lee".istitle())  # True!
# Duck Typing - If it walks like a duck, and talks like a duck, it must be a duck
# Assignment - All like JS, but there are no special keywords like let or const
a = 3
b = a
c = "heyoo"
b = ["reassignment", "is", "fine", "G!"]
# Comparison Operators - Python uses the same equality operators as JS, but no ===
# < - Less than
# > - Greater than
# <= - Less than or Equal
# >= - Greater than or Equal
# == - Equal to
# != - Not equal to
# is - Refers to exact same memory location
# not - !
# Precedence - Negative Signs(not) are applied first(part of each number)
#            - Multiplication and Division(and) happen next
#            - Addition and Subtraction(or) are the last step
#  NOTE: Be careful when using not along with ==
print(not a == b)  # True
# print(a == not b) # Syntax Error
print(a == (not b))  # This fixes it. Answer: False
# Python does short-circuit evaluation
# Assignment Operators - Mostly the same as JS except Python has **= and //= (int
division)
# Flow Control Statements - if, while, for
# Note: Python smushes 'else if' into 'elif'!
if 10 < 1:
    print("We don't get here")
elif 10 < 5:
    print("Nor here...")
else:
    print("Hey there!")
# Looping over a string
for c in "abcdefgh":
    print(c)
# Looping over a range
for i in range(5):
    print(i + 1)
# Looping over a list
lst = [1, 2, 3, 4]
for i in lst:
    print(i)
# Looping over a dictionary
spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
for v in spam.values():
    print(v)
# Loop over a list of tuples and destructuring the values
# Assuming spam.items returns a list of tuples each containing two items (k, v)
for k, v in spam.items():
    print(f"{k}: {v}")
# While loops as long as the condition is True
```

```
#  - Exit loop early with break
#  - Exit iteration early with continue
spam = 0
while True:
    print("Sike That's the wrong Numba")
    spam += 1
    if spam < 5:
        continue
    break

# Functions - use def keyword to define a function in Python


def printCopyright():
    print("Copyright 2021, Bgoonz")


# Lambdas are one liners! (Should be at least, you can use parenthesis to disobey)
def avg(num1, num2):
    return print(num1 + num2)


avg(1, 2)
# Calling it with keyword arguments, order does not matter
avg(num2=20, num1=1252)
printCopyright()
# We can give parameters default arguments like JS


def greeting(name, saying="Hello"):
    print(saying, name)


greeting("Mike")  # Hello Mike
greeting("Bryan", saying="Hello there...")
# A common gotcha is using a mutable object for a default parameter
# All invocations of the function reference the same mutable object


def append_item(item_name, item_list=[]):  # Will it obey and give us a new list?
    item_list.append(item_name)
    return item_list


# Uses same item list unless otherwise stated which is counterintuitive
print(append_item("notebook"))
print(append_item("notebook"))
print(append_item("notebook", []))
# Errors - Unlike JS, if we pass the incorrect amount of arguments to a function,
#          it will throw an error
# avg(1)  # TypeError
# avg(1, 2, 2) # TypeError
# ---------------------------------- DAY 2 -------------------------------------
--
```

```
# Functions - * to get rest of position arguments as tuple
#           - ** to get rest of keyword arguments as a dictionary
# Variable Length positional arguments


def add(a, b, *args):
    # args is a tuple of the rest of the arguments
    total = a + b
    for n in args:
        total += n
    return total


print(add(1, 2))  # args is None, returns 3
print(add(1, 2, 3, 4, 5, 6))  # args is (3, 4, 5, 6), returns 21
# Variable Length Keyword Arguments


def print_names_and_countries(greeting, **kwargs):
    # kwargs is a dictionary of the rest of the keyword arguments
    for k, v in kwargs.items():
        print(greeting, k, "from", v)


print_names_and_countries(
    "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
)
# We can combine all of these together


def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
    pass


# Lists are mutable arrays
empty_list = []
roomates = ["Beau", "Delynn"]
# List built-in function makes a list too
specials = list()
# We can use 'in' to test if something is in the list, like 'includes' in JS
print(1 in [1, 2, 4])  # True
print(2 in [1, 3, 5])  # False
# Dictionaries - Similar to JS POJO's or Map, containing key value pairs
a = {"one": 1, "two": 2, "three": 3}
b = dict(one=1, two=2, three=3)
# Can use 'in' on dictionaries too (for keys)
print("one" in a)  # True
print(3 in b)  # False
# Sets - Just like JS, unordered collection of distinct objects
bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}
# bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
school_bag = set(
    ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"]
)
```

```
print(school_bag)
print(bedroom)
# We can use 'in' on sets as wel
print(1 in {1, 2, 3})  # True
print(4 in {1, 3, 5})  # False
# Tuples are immutable lists of items
time_blocks = ("AM", "PM")
colors = "red", "green", "blue"  # Parenthesis not needed but encouraged
# The tuple built-in function can be used to convert things to tuples
print(tuple("abc"))
print(tuple([1, 2, 3]))
# 'in' may be used on tuples as well
print(1 in (1, 2, 3))  # True
print(5 in (1, 4, 3))  # False
# Ranges are immutable lists of numbers, often used with for loops
#   - start - default: 0, first number in sequence
#   - stop - required, next number past last number in sequence
#   - step - default: 1, difference between each number in sequence
range1 = range(5)  # [0,1,2,3,4]
range2 = range(1, 5)  # [1,2,3,4]
range3 = range(0, 25, 5)  # [0,5,10,15,20]
range4 = range(0)  # []
for i in range1:
    print(i)
# Built-in functions:
# Filter


def isOdd(num):
    return num % 2 == 1


filtered = filter(isOdd, [1, 2, 3, 4])
print(list(filtered))
for num in filtered:
    print(f"first way: {num}")
print("--" * 20)
[print(f"list comprehension: {i}")
 for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
# Map


def toUpper(str):
    return str.upper()


upperCased = map(toUpper, ["a", "b", "c", "d"])
print(list(upperCased))
# Sorted
sorted_items = sorted(["john", "tom", "sonny", "Mike"])
print(list(sorted_items))  # Notice uppercase comes before lowercase
# Using a key function to control the sorting and make it case insensitive
sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
print(sorted_items)
```

```
# You can also reverse the sort
sorted_items = sorted(["john", "tom", "sonny", "Mike"],
                      key=str.lower, reverse=True)
print(sorted_items)
# Enumerate creates a tuple with an index for what you're enumerating
quarters = ["First", "Second", "Third", "Fourth"]
print(list(enumerate(quarters)))
print(list(enumerate(quarters, start=1)))
# Zip takes list and combines them as key value pairs, or really however you need
keys = ("Name", "Email")
values = ("Buster", "cheetoh@johhnydepp.com")
zipped = zip(keys, values)
print(list(zipped))
# You can zip more than 2
x_coords = [0, 1, 2, 3, 4]
y_coords = [4, 6, 10, 9, 10]
z_coords = [20, 10, 5, 9, 1]
coords = zip(x_coords, y_coords, z_coords)
print(list(coords))
# Len reports the length of strings along with list and any other object data type
# doing this to save myself some typing


def print_len(item):
    return print(len(item))


print_len("Mike")
print_len([1, 5, 2, 10, 3, 10])
print_len({1, 5, 10, 9, 10})  # 4 because there is a duplicate here (10)
print_len((1, 4, 10, 9, 20))
# Max will return the max number in a given scenario
print(max(1, 2, 35, 1012, 1))
# Min
print(min(1, 5, 2, 10))
print(min([1, 4, 7, 10]))
# Sum
print(sum([1, 2, 4]))
# Any
print(any([True, False, False]))
print(any([False, False, False]))
# All
print(all([True, True, False]))
print(all([True, True, True]))
# Dir returns all the attributes of an object including it's methods and dunder
methods
user = {"Name": "Bob", "Email": "bob@bob.com"}
print(dir(user))
# Importing packages and modules
#  - Module - A Python code in a file or directory
#  - Package - A module which is a directory containing an __init__.py file
#  - Submodule - A module which is contained within a package
#  - Name - An exported function, class, or variable in a module
# Unlike JS, modules export ALL names contained within them without any special
```

```
export key
# Assuming we have the following package with four submodules
#  math
#  |  __init__.py
#  | addition.py
#  | subtraction.py
#  | multiplication.py
#  | division.py
# If we peek into the addition.py file we see there's an add function
# addition.py
# We can import 'add' from other places because it's a 'name' and is automatically
exported


# def add(num1, num2):
#     return num1 + num2


# Notice the . syntax because this package can import it's own submodules.
# Our __init__.py has the following files
# This imports the 'add' function
# And now it's also re-exported in here as well
# from .addition import add
# These import and re-export the rest of the functions from the submodule
# from .subtraction import subtract
# from .division import divide
# from .multiplication import multiply
# So if we have a script.py and want to import add, we could do it many ways
# This will load and execute the 'math/__init__.py' file and give
# us an object with the exported names in 'math/__init__.py'
# print(math.add(1,2))
# This imports JUST the add from 'math/__init__.py'
# from math import add
# print(add(1, 2))
# This skips importing from 'math/__init__.py' (although it still runs)
# and imports directly from the addition.py file
# from math.addition import add
# This imports all the functions individually from 'math/__init__.py'
# from math import add, subtract, multiply, divide
# print(add(1, 2))
# print(subtract(2, 1))
# This imports 'add' renames it to 'add_some_numbers'
# from math import add as add_some_numbers
# ------------------------------------ DAY 3 ---------------------------------
-----
# Classes, Methods, and Properties


class AngryBird:
    # Slots optimize property access and memory usage and prevent you
    # from arbitrarily assigning new properties the instance
    __slots__ = ["_x", "_y"]
    # Constructor
```

```
    def __init__(self, x=0, y=0):
        # Doc String
        """
        Construct a new AngryBird by setting it's position to (0, 0)
        """
        # Instance Variables
        self._x = x
        self._y = y

    # Instance Method

    def move_up_by(self, delta):
        self._y += delta

    # Getter

    @property
    def x(self):
        return self._x

    # Setter

    @x.setter
    def x(self, value):
        if value < 0:
            value = 0
        self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        self._y = value

    # Dunder Repr... called by 'print'

    def __repr__(self):
        return f"<AngryBird ({self._x}, {self._y})>"


# JS to Python Classes cheat table
#        JS                    Python
#   constructor()         def __init__(self):
#      super()            super().__init__()
#   this.property           self.property
#    this.method            self.method()
# method(arg1, arg2){}    def method(self, arg1, ...)
# get someProperty(){}    @property
# set someProperty(){}    @someProperty.setter
# List Comprehensions are a way to transform a list from one format to another
#  - Pythonic Alternative to using map or filter
#  - Syntax of a list comprehension
```

```
#     - new_list = [value loop condition]
# Using a for loop
squares = []
for i in range(10):
    squares.append(i ** 2)
print(squares)
# value = i ** 2
# loop = for i in range(10)
squares = [i ** 2 for i in range(10)]
print(list(squares))
sentence = "the rocket came back from mars"
vowels = [character for character in sentence if character in "aeiou"]
print(vowels)
# You can also use them on dictionaries. We can use the items() method
# for the dictionary to loop through it getting the keys and values out at once
person = {"name": "Corina", "age": 32, "height": 1.4}
# This loops through and capitalizes the first letter of all keys
newPerson = {key.title(): value for key, value in person.items()}
print(list(newPerson.items()))
```