Thanks for checking out my notes! I was writing these down as fast as I could because the class moves pretty fast so alot of it doesn't flow like normal sentences, and I was working a lot of the time so forgive me if I missed a few things - thanks for watching - Pete :)

HARVARD CS50 Intro to Python Programming (On FreeCodeCamp.org youtube)
Episode 0: Functions, Variables (0:04:48 - 1:50:24)

They use Visual Studio. They start with print("hello, world"). They talk about auto complete. Discuss command line interface (terminal window) instead of icons for running programs. Python is a language to write code AND an interpreter which allows .py files to be converted translated into 0s and 1s aka machine code/

Arguments are inputs to a function. A function is a pre-built operation you may want to run multiple times with different arguments. A side-effect is a visual, audible, or any other kind of measurable output from a program or function. Bugs are mistakes. Errors provide information on bugs and often provide hints at fixing bugs. Do not dive into practical examples much, just discuss the terms.

Gives an example of command line input by prompting users for their name. Then stores that value into a variable aka a return value. Describes single equals sign as assignment operator (sets a value from right to left). Shows how to print a variable and combine printing variable with printing string.

Discusses and shows comments. # pound sign for one line. Discusses using pseudo code and comments as a todo list. Also shows using triple quotes to do big block comments. Shows using the plus sign to concatenate strings, and a comma to add print arguments.

Discusses viewing documents at python.org/docs. Parameters when you write the function, arguments when you pass them in. Says \n is newline, sep is short for separator, Gives some guidance for how to read python documentation and how to change default parameters.

Positional parameters execute in specific order. Named parameters must have 'name' and are optional. Shows how you can use different single or double quotes for strings, and can use backslash inside strings for literals (escape character). Shows using an f string with {curly brackets} for variables and an f at the beginning.

Discuss methods (built in functions that come with data types). Strip removes spaces from either side of the variable. Capitalize makes the first letter of string capital. Title makes the first letter of each word capital. You can chain multiple functions together, so you can strip and capitalize together in one line, and even add methods to input lines.

Split can break a larger string into smaller substrings by passing in what character to break it up with. It returns a list of strings that can be stored in a list or separate variables.

Integers, whole positive or negative numbers. +,-,*,/,%(modulo - remainder). Interactive mode by using python commands live in the terminal window rather than writing in text editor. Build a simple calculator. Uses two inputs to get two numbers, and shows how to transform strings (default) into the data type we want (int) either on input line or after the fact.

Spends a fair bit of time showing that you can combine lots of things onto one line, but that there is always the discussion of are you making it too hard to read when you could just use a few more lines. Then discusses floats which are numbers with decimals (floating point value).

While working with floats and integers, he talks about built in rounding mechanisms. Starts with round, which is the nearest integer up or down by default. Then shows adding a colon in an f string to add a separator to big values, so either a comma or period to break up every three digits of big numbers. Then shows how to use round to the nearest number of decimal points instead of integer (default), or add specific (.2f) formatting to an f string to print a partial string.

Starts discussing how to make functions to get called and do repeated things we would want done a lot of times. Uses making a program that says hello to the user as an example. Introduction to def, to define a function. Discusses the importance of keeping code inside a function one tab/four spaces indented. Shows how to parameterize a function to pass the users name in as an input. Basically walks through the same steps of optimizing the hello world he did in the beginning but now using the input passed in to a function, and adds having a default value for your parameters in case the programmer doesn't provide an argument to the function.

Shows defining functions before calling them is required, or you can put your main code in a function called main, so you can just make sure the last line of your code is main. Discusses 'scope' which is defining variables either available to full project, or just within a single function. You hand off variables by passing and returning information from functions. Discusses using return to get info back from functions that the rest of your program can use.

Finishes with an example of building a function that squares an input value and returns the result as an output. Touches on ** being the raise to power operator, and the pow(x,exp) function.

Episode 1: Conditionals (1:50:24 - 2:46:23)

>, >=, <, <=, ==, != All these are examples of numerical conditionals that can be used with the 'if' keyword. Starts by creating a compare function to make decisions based on user inputs of two numbers. First example is if x is less than y, greater than y or equal to in three separate if conditions. Talks about yes/no questions are boolean expressions.

Introduces elif which will take into account whether or not the initial if statement was true, and only execute if it wasn't. Shows how these can reduce the number of steps a program has to take in making decisions. Adds in else, which removes the need for checking the final scenario.

Explains that these make minor speed differences in these simple examples, but these optimizations make a big difference on big data. Introduces the or statement which allows chaining multiple conditionals. Replaces if x<y or x>y with x!=y, and shows how you could use == in the inverse case. Creates a function to assign grades based on input grade. Initially elif's through > and < cases, then combines those in one check, and then removes the upper bounds. Then they create a parity (odd/even) checker using the modulo operator. Then build it into a function. Then explains 'pythonic' stylistic things, and collapses the whole parity checker into just 'return True if n% 2 == 0 else False'. Then collapses even further to 'return n % 2 == 0'.

Now shows using match instead of if elif stacks, comparable to 'switch' in C languages. Using match and case, and else is basically case _ . Separate many inputs for same case with vertical bar | .

Episode 2: Loops (2:46:23 - 4:07:10)

Makes a function to print meow a bunch of times. While loops run FOREVER until specified condition goes away. Danger of infinite loops with no way of getting out of it. Update your index value when you run the loop somehow. Shows you can use i + 1 or i - 1 to do singular increments for your loop. Discusses starting to count from 0 in programming and computer science rather than one. They even called the first lecture of this zero lecture 0. Show i += 1 is the same as i = i + 1. Similar to other languages ++ or - -.

Then they dive into for loops and list variables at the same time. Lists are a series of values separated by commas and in square brackets. A for loop just goes through a list or value and runs that many times. You can replace the list of values in a for loop with the command 'range(3)' and it says it will run up to but not including that value. Pythonic improvement is if you need a variable but don't use it ever, replace i with _. Super pythonic shortcut is print("meow\n" * 3, end="").

Shows using while True: (intentional infinite loop) to permanently and continuously repeat functionality until 'break' is used. Can use this to make sure you get a positive input from the user and combo with a for loop. Break exits the most recently begun while loop. Shows doing every small individual operation as its own function (get a number, make sure its positive, print something, etc). Return works like break but also sends back a value.

Starts doing some practice problems using lists. Uses a loop for printing every name or item in a list by using for student in students: print(students). Then shows how to use for i in range(len(students)) - length len gets how many items are in a list, or characters in a string, etc. Can then use [i] to address specific items in lists.

Starts showing dictionaries and explaining key-value pairs. Indicated by curly brackets {}, keys are separated from values with a colon, key-value pairs are separated by commas. You can address a value using its key in square brackets[]. Combos with a list with for student in

students print(students[student]).Combines a lot of list, print and loop concepts here. Shows how to make a list of dictionaries so you can put many values for each item inside a list. Also introduces None, which is the absence of a value.

Finishes with fun example printing game levels procedurally. Combines tools, nests for loops.

Episode 3: Exceptions (4:07:10 - 4:51:45)

Exceptions are things going wrong. Shows different types of errors. Starts with syntax error for unended string. Pivots to numbers and writes a program that gets a value from a user, and converts to an integer. Shows ValueError when an invalid input is entered (string, float, etc.).

Introduces try - ‘try to do the following’, except with specific errors (value error), and print x is not an integer if that error occurs. Talks about how using a generic catch-all except that doesn’t specify what type of error is bad practice but possible. Shows NameError which means a variable name doesn’t exist or is not defined properly.

Shows adding in else, which is code that will run if no exception is raised. Adds the try code into a while loop until the user successfully follows the prompt, sending an integer and breaking out of the while loop. Then moves it into a function called get int., and uses past setting up the main function space. Adds that you can use pass to essentially execute but do nothing.

Then he adds a parameter to the get int function ‘prompt’, and uses the prompt to tell the user what they’re inputting. Mentions that you can manually raise exceptions using raise but does not show how in this episode.

Episode 4: Libraries (4:51:45 - 6:09:15)

Libraries are generally files of code that you or others have previously written to be used. Modules are another word for. Talk about random first. docs.python.org/3/library/’modulename’.html. For information on modules. Introduces ‘import’ as the command to get libraries into the program.

First programming example is a coin flipping program. Imports random, and uses random.choice([list of choices]). Equal probability of each item in list. Introduces from, keyword that lets you grab specific functions from libraries rather than full libraries. From random import choice. Now can use it as choice instead of random.choice.

Next uses random.randint(a,b) to get a random value between 1 and 10. Next uses random.shuffle(list) - randomizes a list of values. Uses a list of sample cards as an example. Works on strings or numbers or anything else, and actually shuffles source lists rather than returning a new value.

Next he dives into the statistics library. Statistics.mean([list of values]). Starts to discuss command line arguments. Introduces the sys library. Sys.argv - argument vector (list of all words that human typed in to prompt before enter). Print("hello, my name is", sys.argv[1]). 0 index here is the name of the program being run. Not entering any text in the command line would cause IndexError.

Uses try and except to handle not getting enough arguments. The swaps for an if, elif, else statement. Space by default separates command line arguments unless you enclose the whole thing in double quotes. He then introduces sys.exit which is an immediate end to the program that is running, similar to break from a while loop but this is the whole program.

Then shows how to take a slice of a list to omit index 0 from a for loop. For arg in sys.argv[1:] From start # to end #, if either side of the colon is blank it will start at the beginning or run til the end. Negatives will count backwards from end of the list.

He then talks about packages, which is a 3rd party library that a user can install - PyPI is the python package index (pypi.org). He uses cowsay as an example. Uses this to explain and introduce pip which is python's included package installer/manager. 'Pip install cowsay', import cowsay, cowsay.cow('string'). Silly ascii art cow saying your entered string. Cowsay.trex - same deal but a trex says it.

Introduces APIs, application programming interface - typically third party code packages that you can query against or use by connecting through an API. Python's popular API package is 'requests'. pypi.org/project/requests. Itunes.py, talks about the importance of reading API documentation.

JSON java script object notation - typically used to exchange data between computers. Writes python code to use sys.argv and if elses to request itunes data using requests.get('url{args}.{args}'), and store the response in a variable. Then imports the json library, and uses json.dumps(response.json(), indent=2) - shows a dictionary of key-value pairs.

Then starts parsing the response to just get useful pieces, for result in obj["results"]: print(result["trackName"]). Then he shows how to use a .py file that you have already created as a 'library' by importing it in the beginning just like any other module.

Shows importance of using If __name__ == "__main__": main() in creating your own 'library'.py files to make sure that if you call your own files as libraries they don't run as if they were called on their own.

Episode 5: Unit Tests (6:09:15 - 7:00:22)

Returns to calculator example from before to show how to create a unit test. Gets an input from the user and returns that valued squared. Adds in the if name = main (from above) so that if this

function was imported it wouldn't run the whole thing, just whatever programs/pieces were wanted.

Starts creating a function that the sole purpose of is to test his squared function. Calls it test_calculator.py . Starts by from calculator import square. Def test_square(). Says if square(2) != 4: print error msg. Same for 3-9, and the first time running gets nothing. Then changes from squaring to adding to show getting an error. Talks about importance of writing optimal test code to avoid complicated and lengthy tests. Introduces the assert command.

Assert is 'boldly claiming something is true'. Assert square(2) == 4, 3==9, no ifs or formatting. Shows that this causes AssertionError. Adds try and except assertion error. Shows that sometimes you get lucky even with bad code and the program seems to work, so it's very important to test for a wide range or representative scenarios.

Introduces pytest library. Unit tests are tests for functions that you have written. Uses asserts and removes try and excepts, then instead of running python test_calulator.py, writes pytest test_calculator.py . Does try and except type info for you and prints out results. Not the easiest to read but still useful.

Then breaks test file into testing positive, negative and zero cases, so if an error is thrown using pytest for one case, the other cases will still be attempted so you get more info when testing. Talks about the responsibility of catching all possible cases being on the programmer.

Then adds a test to check if the wrong format of input was given, with pytest.raises(TypeError): square("cat"). Pivots to a new type of test based on original hello world program. Identifies that you need to return a value from the function you're testing to be able to use the assert and pytest stuff.

Then talks about packages - multiple .py files inside a folder. Adding a __init__.py file is what tells python it's a package and not just a file or module alone. So you can run pytest test, pytest will search through the folder, look for tests to run and automatically test.

Episode 6: File I/O (7:00:22 - 8:32:32)

File I/O Is all about writing code that can read information from or write information to external files to 'save' things that happen while the program is running. We do that in this lesson using lists. Uses example of getting user name as input, but gets 3 names, and stores value into a new names list. Again, if not using i inside of for loop use _ instead. Prints all names with sorted(names) to show in console window in alphabetical order.

Introduction to open function, just need to provide name of file to open, and (optionally) how you want to open it. open('names.txt', 'w') (write opens wanting to send data, and will create if it doesn't exist). file.write(name), file.close(). Shows that running this over and over by default overwrites data, does not default to append. open('names.txt', 'a') is appending instead of

overwriting (clobbering), but this will smush new data together if you do not include a newline character at the end of each entry.

Talks about the risk of forgetting to file.close() at the end, more python option is use keyword ‘with’. This is used by replacing file = open by with open() as file: then indent writing into the context of with. This automates closing.
Now they pivot to reading. With open(‘names.txt’, ‘r’) as file: lines = file.readlines(). For line in lines: print(‘hello, ‘, line). This shows all data one at a time, but the newlines in the text file make everything skip a line. Fixes by adding line.rstrip to the print initially. Then replaces with for line in file: print(‘hello, ‘, line.rstrip()). Same effect but much cleaner and more pythonic.

Then they show we want the data to be read back sorted alphabetically. To do this we would need to backpedal to storing in a list before printing, or you can say for line in sorted(file): if you don't want to make any changes and just read the data. Also shows you can sort in reverse order using reverse=True. Starts adding more information to the file, and changes the file from .txt to .csv (comma separated values). Now information grouped together is separated by commas, and individual entries are represented by rows.

To read from a csv, still use with open(‘students.csv’) as file: (read is default so don’t need to specify) and for line in file: . Now each entry will be read in as a full line, but need to split on the commas. This is done using row = line.rstrip().split(“,”), which will return a list of individual values inside that row. If you know what values you’re getting back use variables like name, house = line.rstrip().split(“,”). Then adds each line and their values to a list to sort. Shows it would be best as a list of dictionaries. Student = { }, student[”name”] = name, student[“house”] = house. Then students.append(student).

Then optimizes student = {“name:name, “house”:house}. Creates a function called get_name() to get the name of the student from the dict. Shows you can sort a dictionary by using this function in the way of sorted(students, key=get_name). This allows you to change how to sort your dictionary by any key. He then introduces single line ‘lambda’ functions, which are single line, single use ‘anon’ functions, good when you don’t need the function anywhere else in the code.

For student in sorted(students, key=lambda student: student[“name”]): lambda param1,param2, etc: operation to return. Then shows issues of having commas inside of data that should only be one column. Turns out there is a csv library! Still with open(“students.csv”) as file: but now add reader = csv.reader(file). For row in reader still returns a list of all items. Shows you can use the first row of data as the column names. Now can use csv.DictReader(file), for row in reader: and each row is a dictionary of info paired up with column names as keys.

Then writes using file append like before and writer=csv.writer(file) and writer.writerow([name, home]). Then uses csv.DictWriter(file, fieldnames=[“name”, “home”]) and writer.writerow({“name”: name, “home”: home}). Now talk about using the pillow library to process image files.with example of gifs.

He starts with two images.gif files. Import sys, from PIL import Image, images = [ ], for arg in sys.argv[1:]: image = Image.open(arg). images.append(image). images[0].save('name of file.gif', save_all=True, append_images=[images[1]], duration=200, loop=0). Runs the command line, passes in the two images, and it gives a looping gif. Says this is the basics, doesn't do a deep dive on audio, video, etc. Just an introduction.

Episode 7: Regular Expressions (8:32:32 - 10:37:35)

'RegExes' give you the ability to find patterns in your code. Starts by writing an example to prompt the user for their email address as a user input string. Creates an if @ in email: valid, else invalid. Adds a check to make sure a dot and @ are in the email, now valid.

He then splits the input on the @ sign into username first half and domain name second half. Now he checks that there is some text before the username, and the domain.endswith(".edu"). Shows that to make a good and robust email checker, it would be very tedious, and introduces the regular re library.

re.search(pattern, string, flags=0) - if re.search("@", email): still not great but more succinct. Same limitations as before. Introduces some specific helpful characters (dot - any character except newline, * 0 or more repetitions, + 1 or more reps, ? means 0 or 1 reps, {m} m reps, {m,n} m-n reps.

Email checker turns into re.search(r".+@.+\.edu") - (r is raw string, passes \ literally).
Shows a few more new symbols ^matches the start of a string, $matches the end of a string. Email checker now turns into re.search(r"^.+@.+\.edu$", email): Now introduces [ ] for a set of brackets, and [^] complement the set (you cannot match these characters).

Email checker now turns into re.search(r"^[^@]+@[^@]+\.edu$", email): he then introduces a lot more specific characters about \d decimal digit, \D not a decimal, \s whitespace, \S not whitespace, \w word \W not a word.

Then he dives into how to handle lowercase/uppercase. You could use .lower or .upper to fix the original input, but also re.search supports optional flags. re.IGNORECASE (case insensitive), re.MULTILINE (self explanatory), re.DOTALL (configuring the dot to be any character including newlines).

The specifics get pretty complicated and have a lot of iterating and trying different things and seeing what breaks and what works. Ultimately, re.search and other re functions rely on users reading documentation to understand what tools are available to them.

He then shows what most browsers use to check for valid emails and it is incredibly long and challenging to read. Then he starts talking about re.match and re.fullmatch which automatically start from beginning and beginning and end.

Then we switch into cleaning up data. Start by getting user input ‘name’. We add .strip() to the input. Then split on a comma and store as last, first. Then redefine name as {first} {last}. This is manual, now uses matches = re.search(r”^.+, .+$”, name) - A|B a or b, (...) a group, (?:...) non-capturing version.
If matches (something exists), last, first = matches.groups(), name = f”{first} {last}”. Then cleans it up and optimizes. He then introduces := (walrus operator) to allow you to assign a value and ask a boolean question/evaluation in the same line of code.

Now goes into extracting data from an entry. Prompts user for a URL. username = url.replace(“https://twitter.com”, “”). Programmer version of find and replace. Then uses removeprefix to only remove text if it is at the beginning.

Then he starts fresh, imports the re function, and still gets url from user. Now introduces re.sub(pattern, replacement, string, count=0, flags=0). Automatically substitutes the pattern for the replacement string inside the string. re.sub(r”^(https?://)?(www\.)?twitter\.com”, “”, url).

He then shows you can use search instead of sub to make sure it exists, and only get the username portion. Then he uses the walrus operator to check for a username and set variables matches equal to the result if so. Also shows if you want something optional in the string but not captured as a group use the (?:...) like with the www.

Finishes by talking about how you can use [a-z0-9_] to allow groups and ranges of characters.

Episode 8: Object-Oriented Programming (10:37:35 - 13:28:47)

Discusses that most stuff has been procedural (top to bottom), and dabbling in functional (passing functions around), and that python gives you flexibility. OOP is a solution to problems that come from huge and complicated programs.

Starts by making a procedural program that gets name and house from the user and prints that out. He then makes an ‘abstraction’ of this by making functions get name and get house and puts them inside the main function. Adds the if __name__ == __main__ : run main line to the bottom. Then changes to getting both inputs in one function and returning multiple values from it. Introduces the tuple (pronounced tupple), a collection of data that you can’t change the values of. You can address items inside a tuple same as items inside a list [#].

He then changes the get student function to returning a dictionary with the same info instead of a list or tuple. Now introduces classes - creating reusable code to use for multiple instances.

class Student: … - attributes are values specified inside them. Student = Student(), student.name = input(“Name: “), student.house = input(“House: “), return student. A specific instance of a class in an object.

Methods are functions inside a class that behave in a special way that you can define. Shows def __init__ (self, name, house): double underscore is called dunder lol. This is an Instance method, self.name = name, self.house=house. This is a real start of object oriented programming. Specifying self is weird but is used to create a local memory of stuff passed in.

Introduces raise to manually create an error message. If someone doesn't enter a value, raise ValueError('missing name'). Or if house not in valid options raise ValueError('invalid house').

Shows using def __str__(self): can be used to return info about the object as a string rather than memory location. Now he dives into defining custom functions inside of classes.

Implements a function to match a spell to an emoji to print out. Calls by doing student.charm() and def charm(self) inside student class. Now introduces properties - attributes with more defense mechanisms in place to avoid mess ups. Decorators are functions that modify how other functions work.

Uses getter and setter functions with the same name, intended to prevent users from avoiding built in error checking. Move error checking into a setter function. Student.house = will automatically call a def house(self, house) setter function. Getter requires @property before the function, setter requires @house.setter before it. This allows you to remove all error checking anywhere else for a value. Anytime you try to do .'name' = it will automatically look for that setter. However, to use getters and setters you can't have self.house and def house so change one to _house - he uses self._house = house and def house():

Talks about python relying on conventions and best practices sometimes, but a programmer could break code by intentionally using the _name directly to avoid getter and setter. Now he shows that int(x, base=10) is actually a class. And any variable that is an integer is an object of the int class. Same with str, and that things like str.lower() or str.strip() is an object method being called. Same with list and every other data type.

Now he talks about class methods. He makes a 'sorting hat' class Hat:... then adds hat = Hat(). hat.sort("Harry"). Def sort(self, name). Initializes list of houses in init, randomly chooses one using random.choice(self.houses). Then he discusses that using classes for something that is only used once is a little bit overkill because its benefit is in being used repeatedly by like objects.

He then introduces @classmethods and @classvariables. @classmethod gets added right before the method inside a class, and self gets replaced with cls as first parameter. Removes self.houses from init function, and random.choice(cls.houses) inside def sort. Now you can just call Hat.sort rather than making an object of it.

Now goes back to original student example. Starts cleaning up by getting student info inside class. @classmethod Def get(cls): name = input('Name: '), house same, return cls(name,

house). This means we can call this method without instantiating an object first. This way we can use get to make a student which is a class method inside the class.

Now inside main function student = Student.get(). Now mentions there are @staticmethod but does not dive into it. Now talks about inheritance. This gives classes a chance to get or inherit, methods from parent classes.
Makes a class called student like before with name and house. Now makes another class called professor which has name and subject. Shows that when there is overlap between classes with alot of common parameters, find a way to combine. So he makes a third class called wizard and just gives it name with error checking for name.

Now makes class Student(Wizard) and Professor(Wizard): def __init__(self, name, house): super().__init__(name), self.house=house and same for professor but with subject instead of house.

Uses exceptions as an example of a common parent class - inherited child class hierarchy. Then starts talking about operator overloading. Like using + for concatenation and addition. Makes a vault program holding types of 'coins'. Makes a class called vault. def__init__(self, galleons=0, sickles=0, knuts=0). Stores all coins in self.galleons = galleons, etc.

potter = Vault(100, 50, 25). Weasley = Vault(25, 50, 100). Shows what if we want to combine the contents of two vaults? You could reference each individual variable in each class, combine them and make another class, but that is tedious. Instead what if we could do potter + weasley.

Talks about special methods like object.__add__(self, other) now goes into vault class and makes def __add__(self, other): galleons = self.galleons + other.galleons and same for sickles and knuts. return Vault(galleons, sickles, knuts). Now using the + operator just makes a new vault that is the total combined values.

Episode 9: Et Cetera (13:28:47 - 15:57:47)

Starts by talking about the miscellaneous things we didn't dive into in depth yet, but are useful to cover in this course. Sets are like lists that can not have duplicates, shows this is useful for getting all unique values in a dataset, houses = set(). Then talks about global variables which allows you to reference variables you defined outside of function and change their values inside functions if necessary. global variable_name is all you need to do to be able to write to it.

He then does the exact same code as he did with globals with a class. Talks about using globals sparingly, generally makes it harder to know where information is coming from in debugging. He then talks about making constants in all-caps variables at the top of the program to be clear where fixed values are coming from. For _ in range(CONST): CONST = 3 etc.

Then talks about type hints. Tells python what type a variable is going to be rather than letting it figure it out by default - done using (variable: type). Uses a package called mypy. It will flag an

incompatible type error ahead of time. Tool for programmers, not something to roll out as part of the package.

Then he talks about docstrings, which are recommendations for how to document your code. Formally documenting a function should be done with the triple quote ‘’’comment and description’’’ right after the line of def. Also shows standard conventions of defining what parameters, types, return values and more should be inside the triple quotes with :param, :type etc. Then he shows how to make optional command line arguments using sys.argv and -n .

Introduces the argparse library. Create a parser instance of the argparse.ArgParser(), defines what optional arguments you want to be able to handle, and tell it what to do for each. This is a good tool but you should definitely read the documentation if you want to use it since they didn’t do a deep dive.

Then they talk about unpacking values. Instead of function(list[0], list[1], list[2]), you can use function(*list). Then unpacks dicts. **dict passes in one at a time key=value. Also the syntax used for *args and **kwargs. Arguments and keyword arguments. Shows print uses this asterisk format because print take (*objects) and can show as many inputs as it is given.

Now he introduces map. This is used for ‘functional’ programming (not oop). It allows you to apply some function to every element of a list. map(function, iterable). map(str.upper, words).

Next moves onto being able to do a list comprehension to construct a list on the fly. list = [word.upper for word in words] . or new_list = [list1[‘name’] for student in students if student[“house] == ‘whatever’]

He then introduces filter. New_list = filter(filter_function, list1). This allows an entire list to be given true/false using the passed function that will then create a new list of the trues from the function. This is another list comprehension replacement in a way.

He then dives into dictionary comprehensions. Similar to list comprehensions, this creates dicts on the fly. Students = {key: ‘value’ for item in list1}. Basically same rules as list comprehensions but for dicts.

He then touches on enumerate, which allows you to return the index and value of an item in a list all at once. For i, student in enumerate(students): print(i + 1, student). Then introduces generators. These help when processing massive amounts of data or running huge programs that can use a ton of computer memory. They can return a small chunk of the massive datasets at a time. He says the documentation is important, but the most important is the keyword yield.

Yield is a replacement for return inside functions that essentially means return one value at a time, allows the overarching for loop to keep running, and start returning values even if the program hasn’t finished. Yield is returning an ‘iterator’.

Ctrl + c will interrupt a long running program if you need to escape. He then summarizes very quickly the whole course in the last five minutes of the video. He ultimately finishes by using the pyttsx3 python text to speech module and cowsay to print out the ascii cow and the spoken words ‘this was cs50’.