23.17 While Loops

- for loops = "definite iteration" = loop's body is run a predefined number of times. while loop = "indefinite iteration" = loop repeats an unknown number of times and ends when some condition is met.

```
card_deck = [4, 11, 8, 5, 13, 2, 8, 10]
hand = []
while sum(hand)  < 17:
   hand.append(card_deck.pop())
```

-> features 2 new fns. sum returns the sum of the elements in a list, and pop is a list method that removes the last element from a list and returns it.

23.10 For Loops — Using the `range()` Function with `for` Loops

range() = built-in fn used to create an iterable sequence of number. use range() w/ a for loop to repeat an action a certain number of times.

```
for i in range(3):
    print("Hello!")
```

23.22 Break, Continue = more control over when a loop should end, or skip an iteration, which can be used in both for and while loops.

--> break terminates a loop

--> continue skips one iteration of a loop

Below, you'll find two methods to solve the cargo loading program from the video. The first one is simply the one found in the video, which breaks from the loop when the weight reaches the limit. However, we found several problems with this. The second method addresses these issues by modifying the conditional statement and adding continue.

23.25 Zip and Enumerate useful built-in functions that can come in handy when dealing with loops.

- Zip returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables.
- Like we did for range() we need to convert it to a list or iterate through it with a loop to see the elements.

unpack each tuple in a for loop

```
letters = ['a', 'b', 'c']
nums = [1, 2, 3]

for letter, num in zip(letters, nums):
    print("{}: {}".format(letter, num))
```

* Enumerate = a built in fn, returns an iterator of tuples contain indices and values of a list.

```
letters = ['a', 'b', 'c', 'd', 'e']
for i, letter in enumerate(letters):
// for i, letter in zip(range(len(letters)), letters):
    print(i, letter)
```

indices

0 a
1 b
2 c
3 d
4 e

unzip a list into tuples using an asterisk

```
some_list = [('a', 1), ('b', 2), ('c', 3)]
letters, nums = zip(*some_list)
```

23.28 List Comprehensions = create lists really quickly and concisely with list comprehensions.

```
capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())
```

can be reduced to:

allow us to create a list use for loop in 1 step.

```
capitalized_cities = [city.title() for city in cities]
```

- Create a list comprehension with brackets [ ], including an expression to evaluate for each element in an iterable.

--> **Conditionals in List Comprehensions** (listcomps)

Can add conditionals to listcomps, after the iterable, use if keyword to check a condition in each iteration.

```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

>> squares = to the list [0, 4, 16, 36, 64], as x to the power of 2 is only evaluated if x is even.

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

To add else, move conditionals to the beginning of the listcomp, right after the expression.

LESSON 24 FUNCTIONS

Learn how to use functions to improve and reuse your code! Learn about functions, variable scope, documentation, lambda expressions, iterators, and generators.

24.1 Intro - We'll learn about:

- Defining Functions
- Variable Scope
- Documentation
- Lambda Expressions
- Iterators and Generators

Functions = take what we have already learned how to do, and put it in a holder that allows you to use it over and over again in an easy to use container.

Bertelsmann Tech Scholarship - Data Track

24.2 ---------- Defining Functions

function Header
function Body

```
def cylinder_volume(height, radius):
    pi = 3.14159
    return height * pi * radius ** 2
```

// Arguments, parameters = values, passed in as inputs
//refer to arg, var and define new var, used within body
//return statement, used to send back an output value from fn to the statement that called fn. return followed by an expression, evaluated to get output value for fn

call the function

```
cylinder_volume(10, 3)
```

- print = output to console
- return = provides value to store/ work w/ later

ARGUMENTS
VALUES PASSED IN AS INPUTS TO A FUNCTION

24.3 Quiz: Defining Functions

Quiz: Population Density Function

Write a function: population_density that takes 2 arguments, population & land_area, & returns a population density calculated from those values.

```
# write your function here

def population_density(population, land_area):
    return population/land_area


# test cases for your function
test1 = population_density(10, 1)
expected_result1 = 10
print("expected result: {}, actual result: {}".format(expected_result1, test1))

test2 = population_density(864816, 121.4)
expected_result2 = 7123.6902801
print("expected result: {}, actual result: {}".format(expected_result2, test2))
```

Quiz: readable_timedelta

function named readable_timedelta. take 1 argument, an integer days, & return a string = how many weeks and days that is.

```
def readable_timedelta(days):
    # use integer division to get the number of weeks
    weeks = days // 7
    # use % to get the number of days that remain
    remainder = days % 7
    return "{} week(s) and {} day(s).".format(weeks, remainder)

# test your function
print(readable_timedelta(10))
```

24.5 Variable Scope

- refers to which parts of a program a var can be referenced, or used, from. Var created inside a fn, be used within that fn.
- Var defined outside fns, can be accessed within a fn, have a global scope, but value of a global var can not be modified inside fn.
- Scope is how information is passed throughout programs

24.8 Documentation = make code easier to understand and use. Fns readable bec' use documentation strings, or docstrings, surrounded by triple quotes = comment to explain the purpose of fn, and how it should be used.

```
def population_density(population, land_area):
    """Calculate the population density of an area. """
    return population / land_area
```

24.11 Lambda Expressions = Used to create anonymous fns, don't have a name, creating quick fns that aren't needed later in your code. Useful for higher order fns, or fns that take in other fns as arguments.

- lambda keyword indicate that this is a lambda expression, following are one or more arguments. Last = expression that is evaluated and returned in this fn.
- With this structure, lambda expressions aren't ideal for complex fns, but can be very useful for short, simple fns.

```
def multiply(x, y):
    return x * y
```

can be reduced to:

```
multiply = lambda x, y: x * y
```

Both fns used in same way, we can call multiply:

```
multiply(4, 7)
```

This returns 28.

24.14 [Optional] Iterators and Generators

- Iterables = objects return 1 of their elements at a time, such as a list. built-in functions = 'enumerate,' return an iterator.
- Iterator = object, represents a stream of data, different from a list, also an iterable, but is not an iterator bec' it is not a stream of data.
- Generators are a simple way to create iterators using fns. Can also define iterators using classes. uses yield to return values one at a time
- We can convert it to a list or iterate through it in a loop to view its contents --->>>

```
def my_range(x):
    i = 0
    while i < x:
        yield i
        i += 1
for x in my_range(5):
    print(x)
```

```
0
1
2
3
4
```

```
lessons = ["Why Python Programming", "Data Types and Operators",

def my_enumerate(iterable, start=0):
    # Implement your generator function here
    count = start
    for element in iterable:
        yield count, element
        count += 1

for i, lesson in my_enumerate(lessons, 1):
    print("Lesson {}: {}".format(i, lesson))
```

```
def chunker(iterable, size):
    # Implement function here
    for i in range(0, len(iterable), size):
        yield iterable[i:i + size]
for chunk in chunker(range(25), 6):
    print(list(chunk))
```

Bertelsmann Tech Scholarship - Data Track

24.17 [Optional] Generator Expressions = combines generators and list comprehensions! You can actually create a generator in the same way you'd normally write a list comprehension, except with parentheses instead of square brackets.

```
sq_list = [x**2 for x in range(10)]
# this produces a list of squares
sq_iterator = (x**2 for x in range(10))
# this produces an iterator of squares
```

24.18 Conclusion > writing functions, check out this talk from PyCon by Jack Diederich
https://www.youtube.com/watch?v=rrBJVMyD-Gs&feature=youtu.be
- blog post about yield and generators from Jeff Knupp.
https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/

LESSON 25 SCRIPTING

Setup your own programming environment to write and run Python scripts locally! Learn good scripting practices, interact with different inputs, and discover awesome tools.

25.01 Scripting
- Python Installation and Environment Setup
- Running and Editing Python Scripts
- Interacting with User Input
- Handling Exceptions
- Reading and Writing Files
- Importing Local, Standard, and Third-Party Modules
- Experimenting with an Interpreter

25.02 Python Installation
- Check = $ python --version > Python 3.7.4

25.03 Install Python Using Anaconda
> https://www.anaconda.com/distribution/#windows
Udacity > Learn Anaconda and Jupyter Notebooks
> https://classroom.udacity.com/courses/ud1111
- Check = $ conda --version > conda 4.7.12

25.05 Running a Python Script > $ python
> exit() or control + D (Mac) or control + Z (WD)
$ python first_script.py

25.06 Programming Env Setup > Atom

25.07 Editing a Python Script

25.08 Scripting with Raw Input
use built-in function input

```
name = input("Enter your name: ")
print("Hello there, {}!".format(name.title()))
```

prompts user enter a name, then uses input in a greeting
- input fn takes data and stores = string, to interpret input to integer, wrap result w/ new type to convert it

```
num = int(input("Enter an integer"))
print("hello" * num)
```

- use built-in fn eval, evaluates a string as a line of Py

```
result = eval(input("Enter an expression: "))
print(result)
```

user inputs `2 * 3`, this outputs `6`

25.10 Scripting with Raw Input Solution: Generate Messages
>> 09_raw_input.py

```
names = input("Enter names separated by commas: ").title().split(",")
assignments = input("Enter assignment counts separated by commas: ").split(",")
grades = input("Enter grades separated by commas: ").split(",")

message = "Hi {},\n\nThis is a reminder that you have {} assignments left to \
submit before you can graduate. You're current grade is {} and can increase \
to {} if you submit all assignments before the due date.\n\n"

for name, assignment, grade in zip(names, assignments, grades):
    print(message.format(name, assignment, grade, int(grade) + int(assignment)*2))
```

25.11 Errors and Exceptions
Syntax errors occur when Py can't interpret our code. EOL end of line.
Exceptions occur when unexpected things happen during execution of a program, even code is syntactically correct, different types of built-in exceptions in Py, will see which exception is thrown in the error message
- ValueError = built-in exception > error when built-in operation/ fn when arg get uncorrect value.
- NameError = var not defined.

25.13 Handling Errors
- Use try state to handle exceptions. 4 cls + 1 more in video can used.
try: mandatory clause in a try state. 1st block code in try state.
except: exception while run try block, jump to except block, handles that exception.
else: If Python runs into no exceptions while running the try block, it will run the code in this block after running the try block.
finally: Before end try state, will run code in finally block under any conditions, before stopping the program.
* Specifying Exceptions = specify which error we want to handle: ValueError exception, KeyboardInterrupt. We can have multiple except blocks.

```
while True:
    try:
        x = int(input('Enter a number: '))
        break
    except:
        print('That\'s not a valid number!')
    finally:
        print('\nAttempted Input\n')
```

```
try:
    # some code
except (ValueError, KeyboardInterrupt):
    # some code
```

Bertelsmann Tech Scholarship - Data Track

25.14 Practice: Handling Input Errors

> program env w/ a Terminal + code editor

25.16 Accessing Error Messages ---------->>>>>

When you handle an exception, you can still access its error message & print something like this ---------->>>

We still access error messages, even if you handle them to keep your program from crashing!

```
try:
    # some code
except ZeroDivisionError as e:
   # some code
   print("ZeroDivisionError occurred: {}".format(e))
```

```
ZeroDivisionError occurred: integer division or modulo by zero
```

25.17 Reading and Writing Files

- Open all files in Py give program interface & automate tasks.
- How to **read a file into Py**.

f = open(path + para.(mode)) = py obj

read() method

close() = free sys resources (ex. file handles)

open a file obj = open a WD to look into a file

- create a file = some_file.txt

```
f = open('some_file.txt', 'r')
file_data = f.read()
f.close()

print(file_data)
```

To see how many files can we open in a OS

```
files = []
for i in range(10000):
    files.append(open('some_file.txt', 'r'))
    print(i)
```

Writing to a File

- open an existing file in 'w' mode, any content that it had contained previously 'll be deleted. If you're interested in adding to an existing file, w/out deleting its content, should use the append ('a') mode.

```
f = open('another_file.txt', 'w')
f.write('Hello World!')
f.close()
```

With = syntax that auto-closes a file after finished using it.

with keyword > open a file, do operations on it, auto close it after the indented code is executed > don't have to call f.close()! --->>

```
with open('another_file.txt', 'r') as f:
    file_data = f.read()

print(file_data)
```

25.20 Importing Local Scripts

`import useful_functions` = file = obj = module

import state creates a module obj = useful_functions.

Modules = Py files, contain definitions and statements. To access objs from an imported module, use dot notation. Can add an alias to imported module to reference it w/ a different name.

```
import useful_functions as uf
uf.add_five([1, 2, 3, 4])
```

```
def create_cast_list(filename):
    cast_list = []
    with open(filename) as f:
        for line in f:
            name = line.split(",")[0]
            cast_list.append(name)

    return cast_list

cast_list = create_cast_list('flying_circus_cast.txt')
for actor in cast_list:
    print(actor)
```

other_script.py

```
num = 2 + 3
```

demo.py

```
import other_script
print(other_script.num)
```

Using a main block

Bertelsmann Tech Scholarship - Data Track

25.21 The Standard Library

import state runs code in the module

demo.py

```
import math

print(math.factorial(4))
```

25.22 Quiz: The Standard Library ---->>>>>>

Quiz: Password Generator

Write a function generate_password that selects 3 random words from the list of words word_list and concatenates them into a single string. Function should not accept any arguments and should reference the global variable word_list to build the password.

```
# Use an import statement at the top
import random

word_file = "words.txt"
word_list = []

#fill up the word_list
with open(word_file,'r') as words:
    for line in words:
        # remove white space and make everything lowercase
        word = line.strip().lower()
        # don't include words that are too long or too short
        if 3 < len(word) < 8:
            word_list.append(word)

# Add your function generate_password here
# It should return a string consisting of three random words
# concatenated together without spaces
def generate_password():

    return random.choice(word_list) + random.choice(word_list)
    + random.choice(word_list)


# test your function
print(generate_password())
```

25.24 Techniques for Importing Modules

- import a fn/ class from a module:

```
from collections import defaultdict
```

- import multiple objects from a module:
- rename a module:

```
import multiprocessing as mp
```

- import an obj from a module and rename it

Modules, Packages, and Names

Modules are split down into sub-modules, contained within a package. A package = module that contains sub-modules. A sub-module specified w/ dot notation.

Modules that are submodules are specified by the package name and then the submodule name separated by a dot.

```
import os.path
os.path.isdir('my_path')
```

FINISH PYTHON