# CONCEPTS OF PROGRAMMING

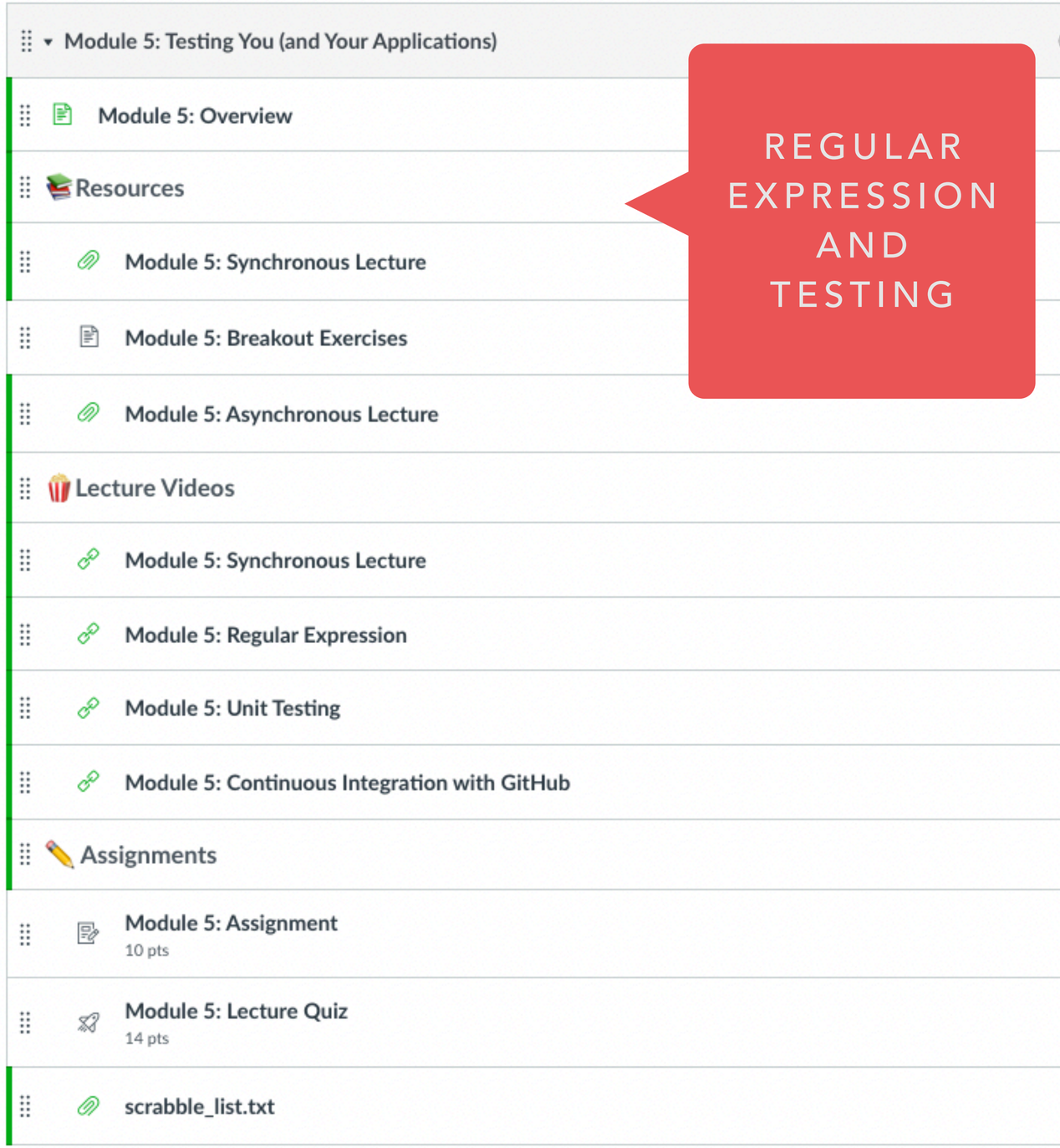


MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# OUTLINE

- Class News
- Review
  - Lecture Quiz
  - Unix Challenge
  - Homework Review
- Breakout
- Module Preview
- Assignment Preview



# CLASS NEWS

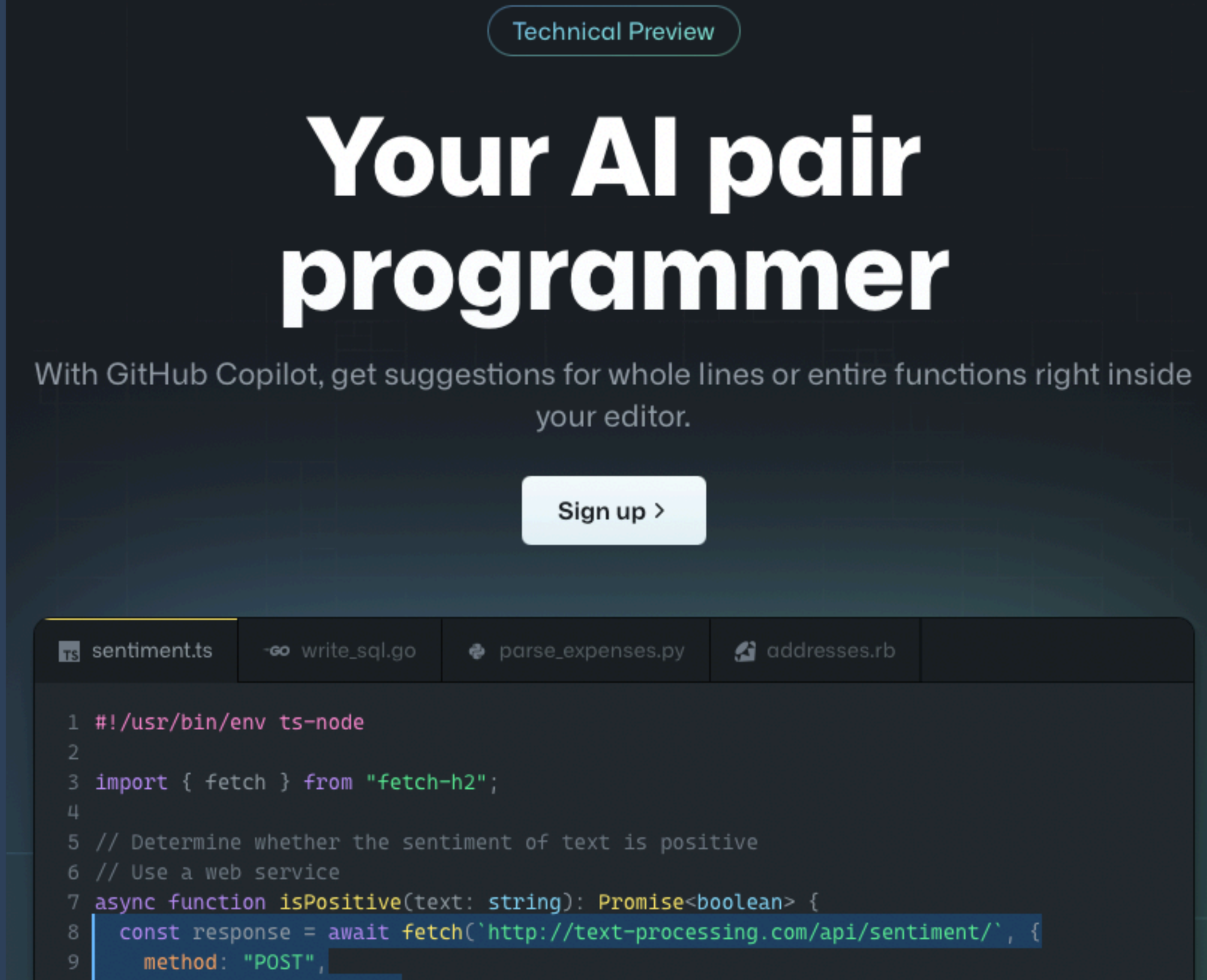


MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# COMPUTING NEWS

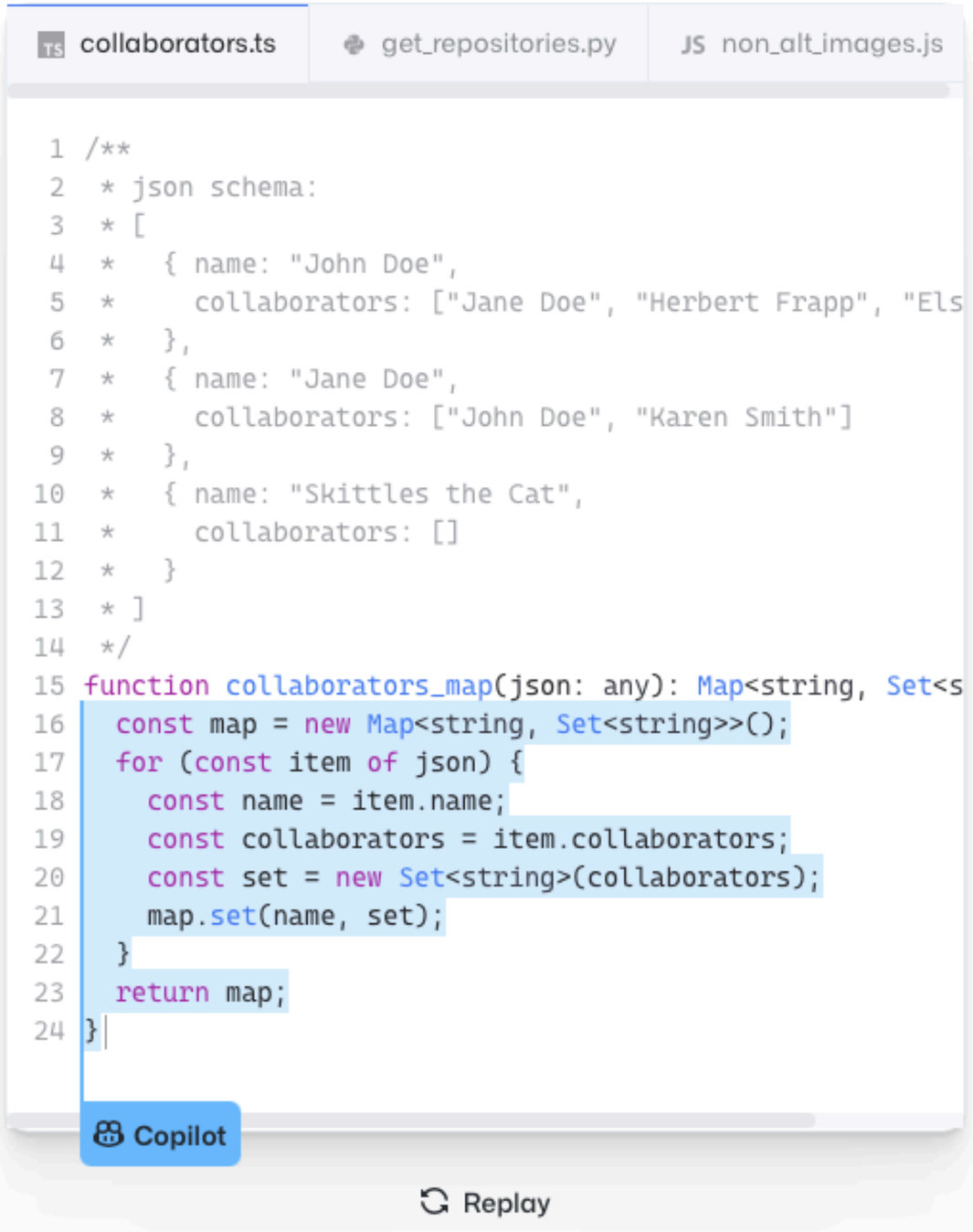
- Github technical preview



# COMPUTING NEWS

- Pseudocode wins the day!

**Convert comments to code.** Write a comment describing the logic you want, and let GitHub Copilot assemble the code for you.



# COMPUTING NEWS

- Inefficient code
- Loop off by 1
- Security failure
- Training failure

## Risk Assessment of GitHub Copilot

### 0xabad1dea, July 2021

*this is a rough draft and may be updated with more examples*

GitHub was kind enough to grant me swift access to the Copilot test phase despite me @'ing them several hundred times about ICE. I would like to examine it not in terms of productivity, but security. risky is it to allow an AI to write some or all of your code?

Ultimately, a human being must take responsibility for every line of code that is committed. AI shou be used for "responsibility washing." However, Copilot is a tool, and workers need their tools to be reliable. A carpenter doesn't have to worry about her hammer suddenly turning evil and creating structural weaknesses in a building. Programmers should be able to use their tools with confidence without worrying about proverbial foot-guns.

A follower of mine on Twitter joked that they can't wait to allow Copilot to write a function to validat JSON web tokens so they can commit the code without even looking. I prompted Copilot according and the result was truly comedic:

```
function validateUserJWT(jwt: string): boolean {
    return true;
}
```

# REVIEW



MODULE 4
MPCS 50101


THE UNIVERSITY OF CHICAGO

# Unix Challenge

# UNIX CHALLENGE

```
# Unix Challenge

tree anaconda3 > output.txt

more output.txt
```

# Lecture Quiz Review

# LECTURE QUIZ

## Question 1

A Python module is a file with the _____ file extension that contains valid Python code.

- .pymodule
- .pym
- .module
- .py

nswer

## Question 2

To use a module in another module, you must import it using an _____ statement.

nswers

import

# LECTURE QUIZ

## Question 3 — 1 pts

A package is a folder containing one or more Python modules. One of the modules in a package must be called _____.

Correct Answer

- __init__.py
- __main__.py
- init.py
- main.py
- __package__.py

## Question 4 — 1 pts

Which of the following is not true about main modules?

- When a python file is directly executed, it is considered main module of a program
- Main modules may import any number of modules
- Special name given to main modules is: __main__

Correct Answer

- Other main modules can import main modules

# LECTURE QUIZ

## Question 5 — 1 pts

What will be the output of the following Python code?

```
from math import factorial
print(math.factorial(5))
```

- 120
- Nothing is printed
- Error, method factorial doesn't exist in math module

Correct Answer

- Error, the statement should be: print(factorial(5))

## Question 6 — 1 pts

If the base condition is not defined in recursive programs what happens?

Correct Answer

- Program goes into an infinite loop
- Program runs once
- Program runs, but recursive function is not executed.
- An exception is thrown

# LECTURE QUIZ

How may times will the string "Hello" be printed?

```
def say_hello(n):
  if n >1:
      print('Hello')
      say_hello(n/2)

say_hello(32)
```

Correct Answers 5

## Question 8 1 pts

The `with` statement is used in exception handling to make the code cleaner and more readable. However, its use is limited to working with managing file resources.

Correct Answer

- True
- False

## Question 9 1 pts

There is no need to call `file.close()` when using `with` statement.

Correct Answer

- True

# LECTURE QUIZ

## Question 11

1 pt

In debugging, the "Step Over" and "Step Into" actions will behave the same in the following circumstance?

Correct Answer

- The next line does not invoke a function.
- The next line invokes a function.
- The next line writes to standard out.
- The next line does not contain a breakpoint.

## Question 12

1 pt

How many times will execution of a program with a two breakpoints will be halted?

- 0
- 1
- 2

Correct Answer

- Unknown

# LECTURE QUIZ

## Question 13 — 1 pts

A *docstring* is a string literal that occurs as the first statement in a function. A *modstring* is a string literal that occurs as the first statement in a function.

- True
- False

Correct Answer: False

## Question 15 — 1 pts

Given the nested `if-else` below, what will be the value `x` when the code executed successfully?

```
x = 0
a = 5
b = 5
if a > 0:
    if b < 0:
        x = x + 5
    elif a > 5:
        x = x + 4
    else:
        x = x + 3
else:
    x = x + 2
print(x)
```

- 0
- 4
- 2
- 3

Correct Answer: 3

# LECTURE QUIZ

**Question 14** 1 pts

The *docstring* can be accessed through the special `__doc__` attribute.

Correct Answer

- True
- False

# Homework Review

# HOMEWORK REVIEW

- There are many ways to solve every problem

```
# 3 tab iteration
for i in range(3):
    print("\t", end="")
```

```
# 3 tab string multiplication
print("\t" * 3)
```

# HOMEWORK REVIEW

- Bool Flags to control the flow of execution

```
def find_number(n):
    found = False

    for number in numbers:
        if number == n:
            found = True

    if found == True:
        print("Found it")
    else:
        print("Nope")
```

# HOMEWORK REVIEW

- Counter variables to keep track of iteration

```
def biggest_number(numbers):
    current_biggest = -1000
    number_count = 0

    for number in numbers:
        if number > current_biggest:
            current_biggest = number
        number_count = number_count + 1

    return current_biggest
```

# HOMEWORK REVIEW

- Pay attention to what goes into to iteration loops

```
def biggest_number(numbers):
    current_biggest = -1000
    seen = []
    found_number = False
    number_count = 0


    for number in numbers:
        seen = []
        if number > current_biggest:
            current_biggest = number
        seen.append(number)
        number_count = number_count + 1

    return current_biggest
```

# HOMEWORK REVIEW

- Reading tracebacks
  - Bottoms up
  - Pay attention to module

```
def greet(someone):
    print('Hello, ' + someon)

greet(‘World')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in greet
NameError: name 'someon' is not defined
```

# HOMEWORK REVIEW

- Reading tracebacks

THE ERROR STARTS WHERE GREET IS CALLED (LINE 2) BUT THE PROBLEM IS IN THE FUNCTION GREET IN MODULE ON LINE 2

```
def greet(someone):
    print('Hello, ' + someon)

greet(‘World')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in greet
NameError: name 'someon' is not defined
```

# HOMEWORK REVIEW

- File reading 3 ways

```
file = open("welcome.txt")
data = file.read()
file.close()
# do something with data
```

```
# Error handing
file = open('welcome.txt')
try:
    data = file.read()
    # do something with data
finally:
    file.close()
```

```
# With statement (autoclose and built-in
# error handling)
with open("welcome.txt") as file:
    data = file.read()
    # do something with data
```

BEST?

# HOMEWORK REVIEW

- Reading files in is “boilerplate” code
  - You’ve already written a function to do it
- Pay attention
  - How the data is written in the file
  - Extra blank lines
  - The very last line

1. Open file
2. Read in file as string
3. Clean up whitespace (\n)
4. Split (if needed)
5. Convert string to expected data type
6. Add to data structure (list)
7. Close file

# HOMEWORK REVIEW

- Reading files in is “boilerplate” code
  - You’ve already written a function to do it
- Pay attention
  - How the data is written in the file
  - Extra blank lines
  - The very last line

1. Open file
2. Read in file as string
3. Clean up whitespace (\n)
4. Split (if needed)
5. Convert string to expected data type
6. Add to data structure (list)
7. Close file

[1,2,3,4,5,6] IN A FILE IS NOT A LIST

# HOMEWORK REVIEW

ARGUMENTS FROM COMMAND LINE

```python
# argv.py
import sys

print("\n\n\n")
print("Command Line:", sys.argv)
print("Python script: ", sys.argv[0])
print("First argument: ", sys.argv[1])
print("Second argument: ", sys.argv[2])


i = 0
for arg in sys.argv:
    print("Argument ", i, ":", sys.argv[i])
    i = i + 1
```

```
(base) andrew@Andrews-MacBook-Pro module-5-demos % python
command-line-args.py cat dogfish bird 100



Command Line: ['command-line-args.py', 'cat', 'dogfish', '
bird', '100']
Python script:  command-line-args.py
First argument:  cat
Second argument:  dogfish
Argument  0 : command-line-args.py
Argument  1 : cat
Argument  2 : dogfish
Argument  3 : bird
Argument  4 : 100
(base) andrew@Andrews-MacBook-Pro module-5-demos %
```

```
python command-line-args.py cat dogfish bird 100
```

# HOMEWORK REVIEW

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print parser.parse_args(['-a', '-bval', '-c', '3'])
```

- Libraries for parsing command line arguments

# HOMEWORK REVIEW

- Can you use emojis as variables?

```
🤔 = "hello world"
print(🤔)
```

```
    File "command-line-args.py", line 17
    🤔 = "hello world"
    ^
SyntaxError: invalid character in identifier
```

# HOMEWORK REVIEW

- Can you use emojis as variables?

```
# emoji.swift
🤔 = "hello world"
print(🤔)
```

```
 >>> swift emoji.swift
🤔
```

# REFACTORING



MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# REFACTORING

- Rock, Paper Scissors,

```
import random
RPS = ["R", "P", "S"]
computer_object = random.choice(RPS)

# Function: player chooses object
print "Choose from Rock(R), Paper(P), or Scissors(S):",
player_object = raw_input()
def start_game():
    print "Choose from Rock(R), Paper(P), or Scissors(S):",
    player_object = raw_input()

# Function: replay
def replay():
    player_selection = raw_input()
    if player_selection is "Y": start_game()
    elif player_selection is "N": print "Thank you for playing!"
    else: print "You must choose either Yes(Y) or No(N)."

# Assess winner
if player_object is "R":
    if computer_object is "R":
        print "Computer chooses Rock - Draw!\nWould you like to play again (Y/N)?", replay()
    elif computer_object is "P":
        print "Computer chooses Paper - Computer wins!\nWould you like to play again (Y/N)?", replay()
    else:
        print "Computer chooses Scissors - You win!\nWould you like to play again (Y/N)?", replay()

elif player_object is "P":
    if computer_object is "P":
        print "Computer chooses Paper - Draw!\nWould you like to play again (Y/N)?", replay()
    elif computer_object is "S":
        print "Computer chooses Scissors - Computer wins!\nWould you like to play again (Y/N)?", replay()
    else:
        print "Computer chooses Rock - You win!\nWould you like to play again (Y/N)?", replay()

elif player_object is "S":
    if computer_object is "S":
```

# REFACTORING

- Functions
  - Seperation of responsibilities
  - Take data in, process, and return

```python
import random

def computer_object():
    choices = ["Rock", "Paper", "Scissors"]
    object = random.choice(choices)

    return object


def main():
    computer = computer_object()
    print(f"Computer: {computer}")



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

# REFACTORING

- Easier to test when functions do a single thing

```python
import random

def computer_object():
    choices = ["Rock", "Paper", "Scissors"]
    object = random.choice(choices)

    return object


def main():
    computer = computer_object()
    print(f"Computer: {computer}")



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

# REFACTORING

- Easier to test when functions do a single thing

NO WAY TO INDEPENDETLY TEST COMPUTER CHOICE

```
def computer_object():
    choices = ["Rock", "Paper", "Scissors"]
    object = random.choice(choices)

    return object
```

```
import random
RPS = ["R", "P", "S"]
computer_object = random.choice(RPS)

# Function: player chooses object
print "Choose f ock(R), Paper(P), or Scissors(S):",
player_object put()
def s

# Fu
def

# As
if p

(Y/N)?", replay()

lay again (Y/N)?", replay()
    else:
        print "Computer chooses Scissors - You win!\nWould you like to play again (Y/N)?", replay()

elif player_object is "P":
    if computer_object is "P":
        print "Computer chooses Paper - Draw!\nWould you like to play again (Y/N)?", replay()
    elif computer_object is "S":
        print "Computer chooses Scissors - Computer wins!\nWould you like to play again (Y/N)?", replay()
    else:
        print "Computer chooses Rock - You win!\nWould you like to play again (Y/N)?", replay()

elif player_object is "S":
```

# REFACTORING

- Flow of program should be driven by main block
- Allows for non-sequential development

```python
import random

def computer_object():
    choices = ["Rock", "Paper", "Scissors"]
    object = random.choice(choices)
    print(f"Computer: {computer}")
    return object


def main():
    computer = computer_object()



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

# REFACTORING

- Where to put the print (debug) statements
  - Keep with function?
  - Print on return?

```python
def main():
    computer = computer_object()
    print(f"Computer: {computer}")
```

```python
import random

def computer_object():
    choices = ["Rock", "Paper", "Scissors"]
    object = random.choice(choices)
    print(f"Computer: {computer}")
    return object


def main():
    computer = computer_object()



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

# REFACTORING

- Think about how we are going to assert our conditions
- Test driven development
  - Write code that satisfies the tests

```
def replace(original, replace, new):
    """Docstring here"""
    #
    # Your code here
    #

    return "something here"
```

```
# Test your function
greeting = "Today is Monday!"
revised = replace(greeting, "Monday", "Funday")

# Test for failure
assert revised != "Today is Monday!"

# Test for success
assert revised == "Today is Funday!"
assert replace("Hello World", "Hello", "Goodbye
```

# ANATOMY OF A FILE (MODULE)



MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# ANATOMY OF A FILE

- Best practices for Python files and modules

```python
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# ANATOMY OF A FILE

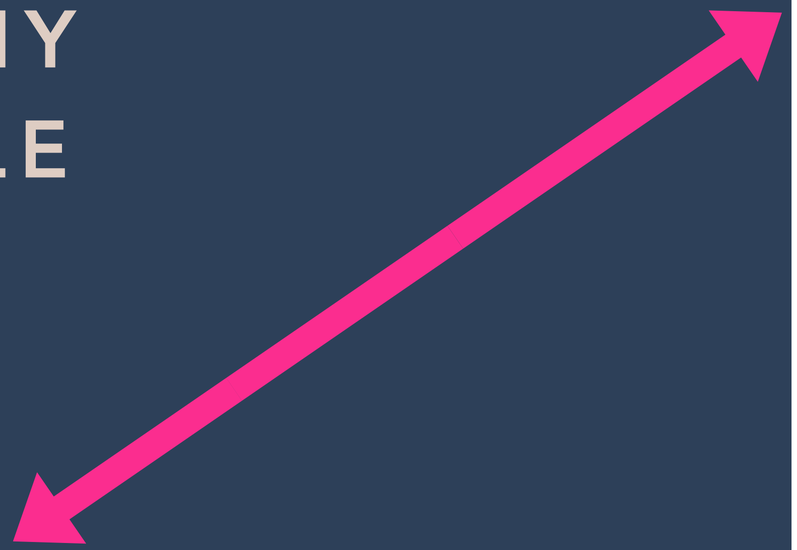
- Shebang #!
- Specify $PATH for Python interpreter
- Used in other scripting languages

#!/USR/BIN/PERL

```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.arg
    # sys.argv[0] is the script name itself and can
```

# ANATOMY OF A FILE

- Shebang
- Docstring
- Author information/metada
- Imports
- Constants - in CAPITALS
- Code



```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

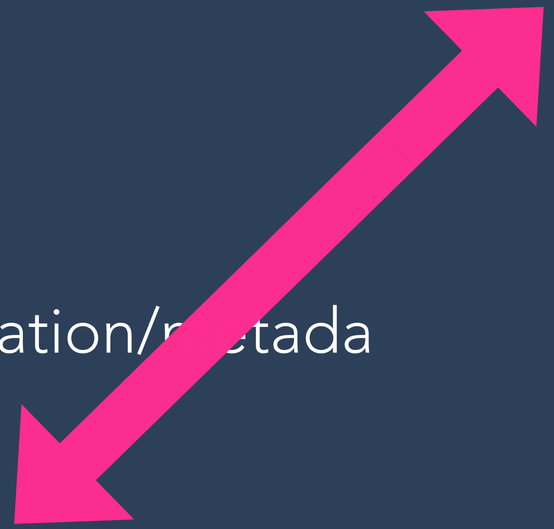
# ANATOMY OF A FILE

- Shebang
- Docstring
- Author information/metada
- Imports
- Constants - in CAPITALS
- Code

```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# Import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# ANATOMY OF A FILE

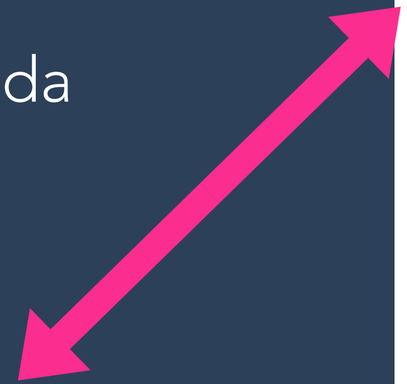
- Shebang
- Docstring
- Author information/metada
- Imports
- Constants - in CAPITALS
- Code



```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# ANATOMY OF A FILE

- Shebang
- Docstring
- Author information/metada
- Imports
- Constants - in CAPITALS
- Code



```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# ANATOMY OF A FILE

- main() function
- Functions
- Biolerplate for main() function

```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard bo
# the program
if __name__ == '__main__':
    main()
```

MAIN() IS A NAMING CONVENTION, NOT REQUIRED

# ANATOMY OF A FILE

- main() function
- Functions
- Biolerplate for main() function

```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in c
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

DOCSTRING AND COMMENTS

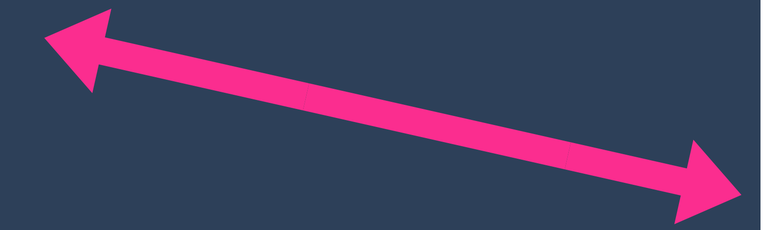
# ANATOMY OF A FILE

- main() function
- Functions
- Biolerplate for main() function

```
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# ANATOMY OF A FILE

- main() function
- Functions
- Biolerplate for main() function



```python
#!/usr/bin/env python
"""Anatomy.py: Description of what this file does."""

__author__      = "Jane Smith"
__copyright__   = "Copyright 2017, Company"
__license__     = "GPL"
__version__     = "1.0.1"
__email__       = "j.smith@co.com"

# import modules used here
import sys
import greeting

# Declare global constants (in caps)
PI = 3.1415
TAX_RATE = 8.75

# A main function if the module will be used as a
# standalone script
def main():
    """Print a greeting to the console using input"""
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

def func1():
    pass

def func2():
    pass

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

# BREAKOUT EXERCISES



MODULE 4
MPCS 50101


THE UNIVERSITY OF CHICAGO

# BREAKOUT EXERCISES

- Assertions provide internal checks for a condition
- Uses
  - Formal testing (pytest, unittest, nose)
  - Serious validation (NASA)
  - Self-checking

```
def alwaysTrue():
    return True

def add2numbers(x, y):
    return x + y


# When asserts fail there is an AssertionError
assert alwaysTrue() == True
assert alwaysTrue() != False
assert alwaysTrue() == False # AssertionError

assert add2numbers(1, 2) == 3 #
assert add2numbers(1, 2) == 5 # AssertionError
```

# BREAKOUT EXERCISES

## Module 5: Breakout Exercises

### Exercise 1

In preparation for you upcoming exam, try to  work these problems out manually. You may check your answers once you have agreed on a solution.

A. What is printed to the screen?

```
def func(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return func(n-1) + func(n-2)

for i in range(0,4):
    print(func(i),end=" ")
```

HTTPS://CLASSROOM.GITHUB.COM/A/ZBFDK_7Q

# MIDTERM EXAM REVIEW



MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# MIDTERM EXAM REVIEW

- Next Lecture
  - Midterm exam
  - Small homework assignment

# MIDTERM EXAM REVIEW

- Why do we have to take exams for computer science?
  - Checkpoint for instructors
  - Good practice for other CS classes
  - Coding interviews

# MIDTERM EXAM REVIEW

- Material
  - Everything through this week (except unit testing with pytest)
    - Asserts may be provided for testing (like in sample)
  - Familiar with Python standard library modules we have used
  - Looking for best coding practices (variable naming, file structure, error handling, data validation, etc.)

- Testing for logic and coding fluency (not esoteric Python features)

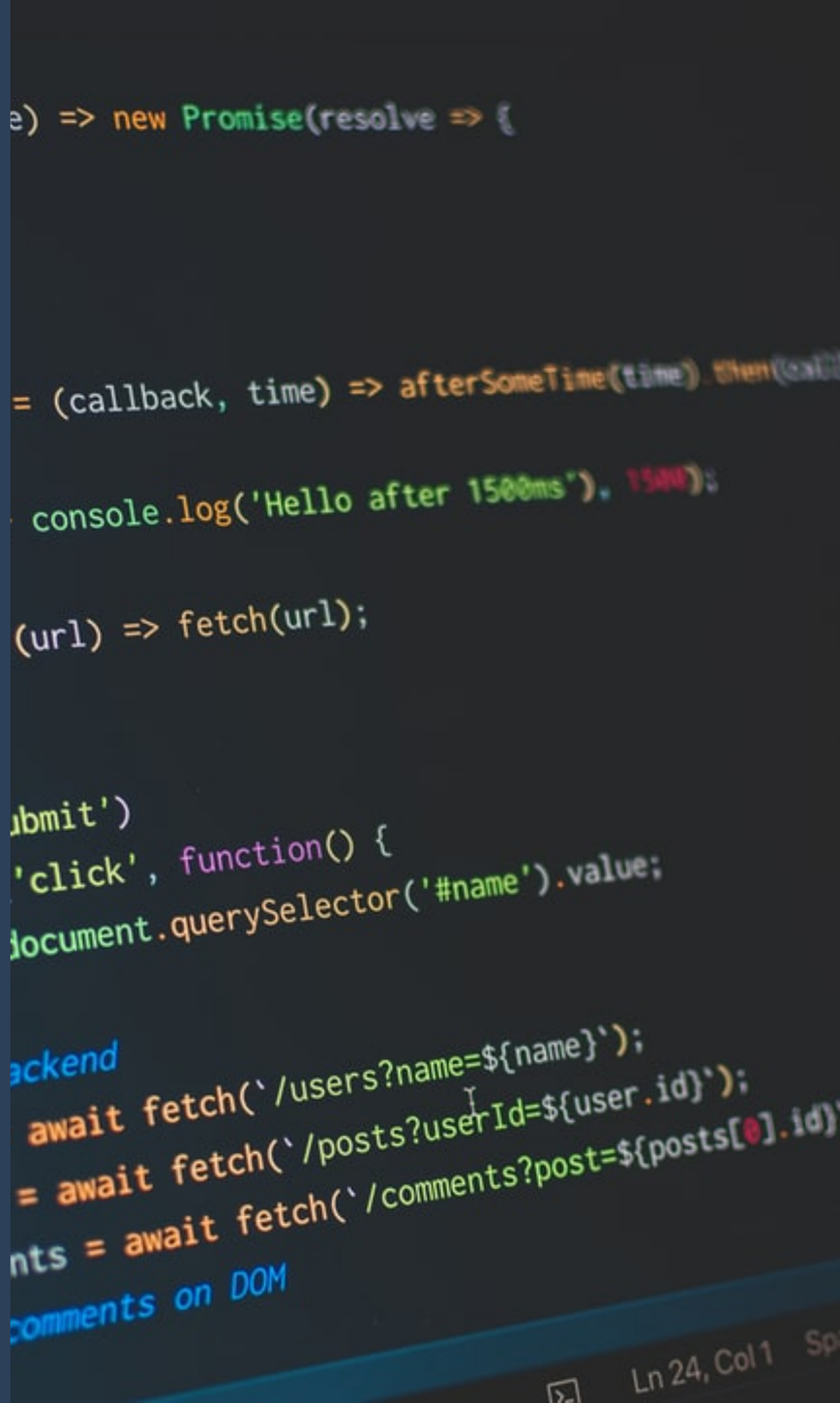
# MIDTERM EXAM REVIEW

- Open coursework
  - Notes, slides,  class resources, assignments 👍
  - Not open Internet (no Googling) 👎

YOU WON'T HAVE TIME TO LOOK EVERYTHING UP

# MIDTERM EXAM REVIEW

- Exam Format
  - Written answer (Canvas quiz)
    - Multiple Choice, Fill in the Blank, True/False
  - Programming (GitHub Classroom)
    - Download the classroom repository
    - Last commit before 7:15PM will be graded (90 minutes)



# MIDTERM EXAM REVIEW

- Logistics
  - 5:30 Zoom
    - Download repository
  - 5:45 Exam officially starts
  - 7:15 Exam is over
    - Last commit before 7:15 is graded.

# MIDTERM EXAM REVIEW

- How do I ask a question?
  - Slack DM
  - Zoom session will be active throughout exam. If you have a question send a message in chat/ Slack and you will be assigned a breakout room.
  - Instructors will be standing by

# MIDTERM EXAM REVIEW

- Tips
  - Read through all the problems first
  - Don’t spend too much time on one problem
  - Write psuedocode first, then fill in the details
  - You will spend more time thinking than typing

# MIDTERM EXAM REVIEW

- Biggest mistakes students make
  - Didn't read through all the problems first
    - Miss some key details (ie. what to return)
  - Spent too much time on one problem
  - No comments
  - Forgetting what's important (correct answer, best practices, etc.)

NOT TRUSTING THEMSELVES

# MIDTERM EXAM REVIEW

- The best way to prepare is to review the homework
  - Go back and pick a coding question and try to it without looking at your notes or Googling

# MIDTERM EXAM REVIEW

- Questions?

# SAMPLE EXAM

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
# Problem 1
# What do the following statements evaluate to?

a) 3+4 == 7
b) not(1==1)
c) not(1!=1)
d) ("ABc").rstrip('c').islower()
e) str(123).isdigit()
```

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
# Problem 2
# What do the following statements evaluate to?

a) 3+4 == 7                    # True
b) not(1==1)                   # False
c) not(1!=1)                   # True
d) str(123).isdigit()          # True
e) ("ABc").rstrip('c').islower() # False
```

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
# Problem 2
# What is the output printed to the screen?

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


countdown(5)
```

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
# Problem 2
# What is the output printed to the screen?

5
4
3
2
1
Blastoff!
```

PAY ATTENTION
N<=0

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
# Problem 5
# Write a program that calculates a 20% tip for a given dollar amount.
# Prompt the user to enter an amount. Using the input, calculate the tip and print out the
# inputed amount, tip and new total.  Use the exact format presented below when printing the screen.
#
#      Amount: $10.00
#      Tip: $2.00
#      Total: $12.00
#
# Validate the user input to ensure that the calculations can be completed.  If not, print a
# message to the user that indicates that there was a problem with the input and end the program.
#
# The program flow should resemble the following:
#
#    % python problem4.py
#    Enter a dollar amount? one hundred dollars
#    I was unable to calculate the tip.
#
#
#    % python problem4.py
#    Enter a dollar amount? 100.00
#    Amount: $100.00
#    Tip: $20.00
#    Total: $120.00
```

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

```
Problem 3: 10 Points

- [ ] 1pts Takes input from command line and validates it
- [ ] 1pts Exits (or re-prompt) with message on bad input
- [ ] 1pts Sets meaningful names for variable and constant values
- [ ] 1pts Does correct tip calculation
- [ ] 1pts Calculates the total by adding tip
- [ ] 1pts Prints in the meaningful format for data ($X.XX)
- [ ] 1pts Good coding practices (file structure, docstring,
           comments, functions, readability, etc.)
- [ ] 3pts Logic appears to be there whether or not code works
```

# MIDTERM EXAM REVIEW

## SAMPLE QUESTIONS

WHAT IF I BLANK?

```
# Get input from user and check that it is valid

# Convert to number and name 'total_bill'

# Multiply 'total_bill' by 20% and name result 'tip'

# Add 'total_bill' to 'tip' and name 'total_plus_tip'

# Print 'total_plus_tip' and format with a $ and 2 decimal places
```

# ASSIGNMENT



MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO

# ASSIGNMENT

- Assignment

## Module 5: Assignment

Published | Edit

For each problem, create a file name problem1.py, problem2.py, etc. Make that any unit test files are named appropriately and can be easily associated with their problem. Please make judicious use of comments and docstrings in your code. Use GitHub classroom to submit your work. We will only grade master branch.

GitHub Classroom Repository: https://classroom.github.com/a/9L4ba4lu

**Submit the GitHub Repository URL to Canvas.**

### Problem 1

Revisit your temperature converter and improve your code base by adding unit tests. This may require you to refactor your code.

In your unit tests, test for any type conversion errors that may occur from user input. Also, test that the formula you are using for the conversion is accurate under all possible valid input. Use accuracy to 2 decimal places while testing.

Write as many tests as necessary to exhaust all possible cases (that you can think of).

### Problem 2

Write a test suite for your solution to the following problem from Module 4:

> A number, `a`, is a power of `b` if it is divisible by `b` and `a/b` is a power of `b`. Write a function that takes parameters `a` and `b` and `returns True` if `a` is a power of `b`.

# THE END



MODULE 5
MPCS 50101


THE UNIVERSITY OF CHICAGO