CS 5001

LECTURE 6

FILES, EXCEPTION HANDLING

KEITH BAGLEY

FALL 2023

Northeastern University
Khoury College of Computer Sciences

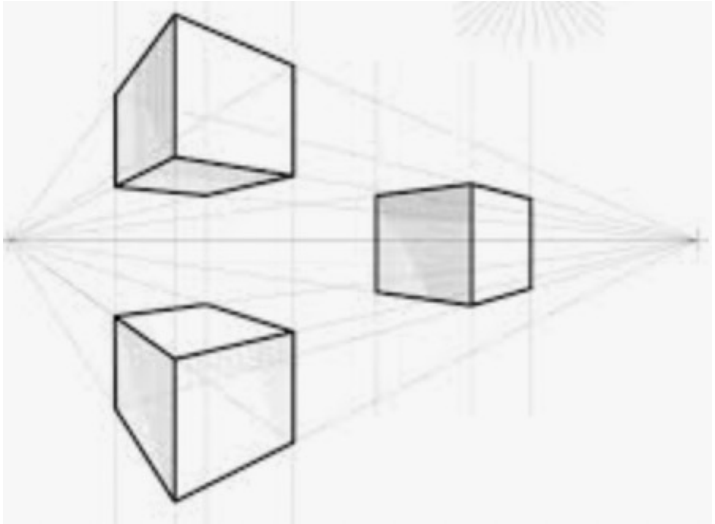
440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

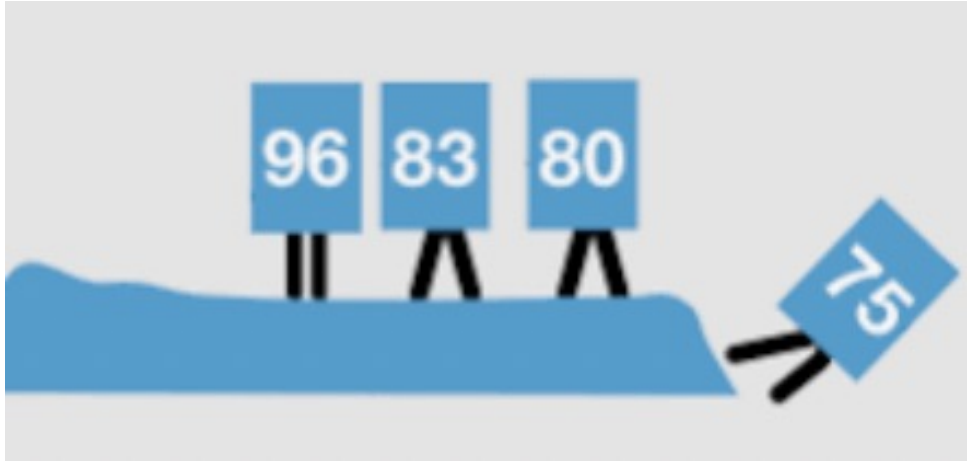
AGENDA

- Course Check-in
- Sequences Review
- Files & Persistent Data
  - Why files
  - Files are not Databases
- Exception Handling
- Remember to complete Quiz 2 this week!
- Midterm Review (Midterm is next week)
- Q&A

CHECK-IN AND PERSPECTIVE

- I structure the course so that no ONE element will “sink” your grade
- Grades are not “curved” so don’t worry about the mean or median
- I also drop your lowest homework grade, so there’s that too… ☺





QUOTE OF THE WEEK

""If you are going to achieve excellence in big things, you develop the habit in little matters. Excellence is not an exception, it is a prevailing attitude."
— Colin Powell

SEQUENCES

- Strings, Lists, and Tuples are all Sequences
- They share a common protocol
  - i.e. there are common functions and operators that can be applied to them all – the Sequenceable protocol
  - This allows for uniformity in how we work with every sequence

SEQUENCES

```
 8
 9 def binary(digits):
10     digits = digits[-1::-1] # start at the last bit & work to the front
11     value = 0
12     for i in range(len(digits)):
13         value = value + int(digits[i]) * 2**i
14     return value
15
16
17
18 def main():
19     print("001", binary("001")) # 1
20     print("100", binary("100")) # 4
21     print("101", binary("101")) # 5
22     print("111", binary("111")) # 7
23
24     print(['1', '1', '1'], binary(['1', '1', '1']))
25
26 if __name__ == "__main__":
27     main()
```

- From last week:
  - Line 24 works because our code was written for sequences, not specific strings, lists or tuples

SEQUENCES: THOUGHT QUESTION

```
10 def main():
11     print("001", binary("001")) # 1
12     print("100", binary("100")) # 4
13     print("101", binary("101")) # 5
14     print("111", binary("111")) # 7
15
16     print(['1', '1', '1'], binary(['1', '1', '1']))
17
18     bin_tuple = ("1", "1", "1")
19     print(bin_tuple, binary(bin_tuple))
20
21
22
23 if __name__ == "__main__":
24     main()
25
```

- Think about this:
  - What if we modify our code and add lines 18 and 19?
  - Will this still work?

SEQUENCES: THOUGHT QUESTION

```
>>>
= RESTART: /Users/kei
DE-for-semester/lectu
001 1
100 4
101 5
111 7
['1', '1', '1'] 7
('1', '1', '1') 7
>>>
```

- Yes!
- Again, Tuples are sequences too. Nothing “breaks” if we pass in a sequence that conforms to the protocol.

ENUMERATE REVIEW

```
>>> p = [1, 2, 3, 4]
>>> p
[1, 2, 3, 4]
>>> for index, value in enumerate(p):
        print(index, value)


0 1
1 2
2 3
3 4
>>> for each in enumerate(p):
        print(each)


(0, 1)
(1, 2)
(2, 3)
(3, 4)
```

- Enumerations give us a way to traverse sequences by index and value, using tuples to package each intermediate pair

FILES

WHAT ARE FILES?

- Files are a form of “persistent” data
  - “Non-volatile” memory. The data remains even if you turn off the power on your computer
  - Often stored to disk
    - But…tape, USB, non-volatile RAM, etc. also options

PERSISTENCE

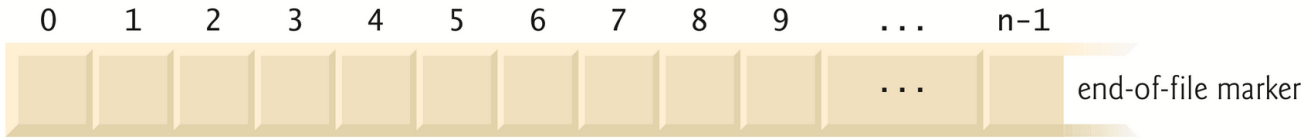
- Persistence is more than just files
- Persistence => the data “persists” over time/space/devices
  - Could be files, databases, non-volatile memory, Cloud storage, continuous transmission
  - For this course, we’ll focus on files

TYPES OF FILES

- There are different kinds of files, and different file formats
  - Text files
  - Music files
  - Video
  - Word Processing
  - Etc.
- Broadly speaking, files can be segmented into 2 major types
  - Text files
    - Sequence of text characters
  - Binary files
    - Sequence of bytes

FILES

- We'll focus on text files, but realize that there are variations even here regarding the schema (representation) of the data
  - several popular formats
    - Plain text
    - JSON (JavaScript Object Notation)
    - CSV (comma-separated values)
    - Others
- For each file you open, Python creates a file object that you'll use to interact with the file
- First character in a text file or byte in a binary file is located at position 0
  - In a file of n characters or bytes, the highest position number is n – 1



OPENING & READING FILES

TABLE 5.1 Opening and Closing Files in Python

Method Name	Use	Explanation
`open`	`open(filename, mode)`	Built-in function. Opens a file called `filename` and returns a reference to a file object. If the file does not exist, raises an `OSError`. The mode can be: `'r'`: open the file for reading `'w'`: open the file for writing. If the file exists, it will be overwritten. `'a'`: open the file for writing. If the file exists, the new data will be appended to the end.
`close`	`fileVariable.close()`	Signals that file use is complete and releases any memory or other resources associated with the file.

OPENING & READING TEXT FILES – OPTION 1

```
# variable name is infile, which is a file object
# 'grades.txt' is the name of the file
# 'r' specifies the mode, we're opening the file for reading
infile = open('grades.txt', 'r')
```

```
# option one: read all the lines of the file into a single string
# data type of grades: string
grades = infile.read()
print(grades)

    # grades contains...
    '85.5\n90.1\n93.7\n98.3\n88.4'

    # terminal output
    85.5
    90.1
    93.7
    98.3
    88.4
```

OPENING & READING TEXT FILES – OPTION 2

```
# variable name is infile, which is a file object
# 'grades.txt' is the name of the file
# 'r' specifies the mode, we're opening the file for reading
infile = open('grades.txt', 'r')
```

```
# option two: read into a list, one file line is one element
# data type of grades: list
# In this example, we strip the linebreak we've read from the file

grades = infile.readlines()
for grade in grades:
    print(grade.strip('\n'), ' ')

    # terminal output
    85.5 90.1 93.7 98.3 88.4
```

Notice the uniformity in how we can access the data? Similar to the other sequences we've covered

FILES: RULES OF THE ROAD

- If you use the plain/basic open function, you are required to close the file
  - Otherwise errors may occur
  - Resource sink
    - Sort of like leaving the door open in the winter with the heat blasting
- ”Modern” Python better practice to use the “with open” approach

```
def open_with():
    with open('accounts.txt', mode='r') as accounts:
        for record in accounts: # read line by line
            acct_num, name, balance = record.split()
            print(acct_num, name, balance)
```

WITH KEYWORD (WITH FILES)

- Many applications acquire resources
  - files, network connections, database connections and more
  - Should release resources as soon as they're no longer needed
    - Ensures that other applications can use the resources
- with statement
  - Acquires a resource and assigns its corresponding object to a variable
  - Allows the application to use the resource via that variable
  - Calls the resource object's close method to release the resource

No explicit close() needed!

```
with open('accounts.txt', mode='r') as accounts:
```

With keyword to acquire resources (file in this case)

Open file

Variable associated

SPECIFYING A "READ POSITION"

- Files are sequential
- We can move the read position by using seek()
- To process a file sequentially from the beginning several times during a program's execution, you must reposition the file-position pointer to the beginning of the file
  - Can do this by closing and reopening the file, or
  - by calling the file object's seek method, as in

```
file_object.seek(0)
```

WRITING TO A FILE

- Similar to reading, but will open the file using the “w” mode
  - Mode 'w' opens the file for writing, creating the file if it does not exist
  - If a path is not specified, Python creates it in the current folder
- Be careful—opening a file for writing deletes all the existing data in the file
- By convention, the .txt file extension indicates a plain text file
- Use the write function to write to the file. The write() function takes one argument and it should be a string value for text files

```
with open('accounts.txt', mode='w') as accounts:
    accounts.write('100 Jones 24.98\n')
    accounts.write('200 Doe 345.67\n')
    accounts.write('300 White 0.00\n')
    accounts.write('400 Stone -42.16\n')
    accounts.write('500 Rich 224.62\n')
```

UPDATING FILES

- Formatted data written to a text file cannot be modified without the risk of destroying other data
  - If the name 'White' needs to be changed to 'Williams' in accounts.txt, the old name cannot simply be overwritten
  - The original record for White is stored as: 300 White 0.00
  - If we overwrite the name 'White' with the name 'Williams', the record becomes: 300 Williams00
- The problem is that records and their fields can vary in size
  - So, in most cases, we must create a temp file to do the modifications & replace the original

UPDATING FILES

- Table of file-open modes for text files
  - *Reading* modes raise a `FileNotFoundError` if the file does not exist
  - Each text-file mode has a corresponding binary-file mode specified with `b`, as in `'rb'` or `'wb+'`

Mode	Description
`'r'`	Open a text file for reading. This is the default if you do not specify the file-open mode when you call open.
`'w'`	Open a text file for writing. Existing file contents are *deleted*.
`'a'`	Open a text file for appending at the end, creating the file if it does not exist. New data is written at the end of the file.
`'r+'`	Open a text file reading and writing.
`'w+'`	Open a text file reading and writing. Existing file contents are *deleted*.
`'a+'`	Open a text file reading and appending at the end. New data is written at the end of the file. If the file does not exist, it is created.

Other File Object Methods

- **`read`**
  - For a text file, returns a string containing the number of characters specified by the method's integer argument
  - For a binary file, returns the specified number of bytes
  - If no argument is specified, the method returns the entire contents of the file
- **`readline`**
  - Returns one line of text as a string, including the newline character if there is one
  - Returns an empty string when it encounters the end of the file
- **`writelines`**
  - Receives a list of strings and writes its contents to a file

EXERCISE: PERSISTENT GRADE KEEPER

- We wrote a few weeks ago
  - We used loops & lists to compute our average?
- We'll revisit that and make our work persistent
  - Download `grade_keeper.py` from Canvas
  - Fill in the details for the functions `load_grades()` & `save_grades()`
  - Uncomment the calls to those functions in main()
  - Run the program. Examine the output
  - Using a text editor, open the grades.txt file and see what's there

FILE THIS AWAY!

- Never ever “hardcode” file paths!
- Python’s “native language” is Linux/Unix path separators (“/”). Do NOT use Windows-style (“\”) since doing so will limit portability
  - `# An example of both is:`
    `open('C:\\Users\\BruceWayne\\tax_return.txt')`
  - Avoid hardcoding file names too, if possible
- Instead, import os, and use the path.join() function:

```
>>> import os
>>> path = os.path.join("taxes", "brucewayne", "2023.tax")
>>> path
    'taxes/brucewayne/2023.tax'
>>>
```

OTHER USEFUL FUNCTIONS FROM OS

- Getting current working directory, Determining if a directory (or file) exists, and other useful operating-system actions can be achieved using the os module

```
>>> os.path.sep
'/'
>>> os.path.split(path)
('taxes/brucewayne', '2023.tax')
>>> os.path.exists(path)
False
>>> os.getcwd()
'/Users/keithbagley/Documents'
>>> os.path.exists('/Users/keithbagley/Documents')
True
>>>
```

EXCEPTION HANDLING

BUT WHAT IS SOMETHING GOES WRONG?

- with open(‘doesNotExist.txt', mode='w') as accounts:

EXCEPTIONS

- Exceptions are error conditions that arise from
  - Logic errors (e.g. wrong calculations that cause the program to crash – e.g. divide by zero)
  - Runtime errors (e.g. accessing non-existent elements in a list)
  - Type errors (particularly for dynamic languages like Python, this is a special kind of runtime error)
  - etc

APPROACHES TO HANDLING EXCEPTIONS

- Defensive Programming
  - Plan for possible “error conditions”
  - Write code to handle or mitigate problems

APPROACHES TO HANDLING EXCEPTIONS

- Use Exception Handling Mechanism
  - Many languages (Python, Java, C++, etc.) have built-in facilities to ASSIST with handling exceptions

APPROACHES TO HANDLING EXCEPTIONS

- Combination of both

+

DEFENSIVE PROGRAMMING

- Handle the problems as you encounter them.
- Need to think about what could possibly go wrong
- Use conditional code to manage the situation

DEFENSIVE PROGRAMMING

- Handle the problems as you encounter them.
- Need to think about what could possibly go wrong
- Use conditional code to manage the situation

```
def defensive_divide():
    while True:
        numerator = input("Enter the numerator ")
        denominator = input("Enter the denominator ")
        if not numerator.isdigit() or not denominator.isdigit():
            print("Enter only numbers please!")
            continue
```

EXCEPTION HANDLING

- Python's try/except block to handles errors in general and are useful for file access error.
- An exception is something unusual that happens; the System doesn't know what to do next, and Python responds by giving us an error message on terminal.
- A try/except block allows us, the programmer, to find these kinds of exceptions, and write code to "handle" them -- give them a better response than the one here, especially a user-facing response

EXCEPTION HANDLING

Division By Zero

Recall that attempting to divide by `0` results in a `ZeroDivisionError`:

```
10 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-a243dfbf119d> in <module>
----> 1 10 / 0

ZeroDivisionError: division by zero
```

EXCEPTION HANDLING

- Use the exception handling mechanisms to “sense” when an exception occurs and then let the framework help you manage the exception

```
def divide():
    while True:
        try:
            numerator = int(input("Enter the numerator "))
            denominator = int(input("Enter the denominator "))
            result = numerator / denominator
        except ZeroDivisionError:
            print("You must enter two non-zero integers")
```

EXCEPTION HANDLING

- Various types of exceptions can occur when you work with files
  - FileNotFoundError
  - Attempt to open a non-existent file for reading with the 'r' or 'r+' modes
  - PermissionsError
    - Attempt an operation for which you do not have permission
    - Try to open a file that your account is not allowed to access, etc.
- Attempt to write to a file that has already been closed
- Plus, other non-file errors (Index Error, etc.)

EXCEPTION HANDLING

- A try/except block tries to execute a piece of code that might cause an exception. If an exception does happen, then we fall to the except and do that instead. Essentially, our code will do the following:
- Try to open a file for reading
- Except, if there's an error, do something nicer than the error message

EXCEPTION HANDLING

```
# Python code to handle a file that can’t be read
try:
    infile = open('gradess.txt', 'r’) #extra s file does not exist
    grades = infile.read()
    infile.close()

except OSError:
    print('Error reading file')
    return
```

OR

```
try:
    with open('gradez.txt', 'r') as accounts:
```

EXCEPTION HANDLING

```
def try_it(value):
    try:
        x = int(value)
    except ValueError:
        print('{} could not be converted to an integer'.format(value))
    else:
        print('int({}) is {}'.format(value, x))

def main():
    try_it(10.7)

    try_it('Python')

main()
```

BOTH-AND

```
def get_numeric_input(prompt):
    while True:
        value = input(prompt)
        if not value.isdigit():
            print("Enter only numbers please!")
            continue
        return int(value)

def both_and_divide():
    try:
        numerator = get_numeric_input("Enter the numerator ")
```

RAISING EXCEPTIONS

- Note that you can also raise your own exceptions
  - We use the raise keyword to do this
- Why would you ever want to do this?

RAISING EXCEPTIONS

- Using raise
- A silly example:

```
def peppers():
    word = input("Enter a secret word")
    if word.lower() == "peppers":
        raise ValueError
    print(f"I like {word}")
```

RAISING EXCEPTIONS

- Using raise
- A more realistic example:

```
def validate_email():
    email_address = input("Enter your email address: ")
    if email_address.count("@") == 1:
        return email_address
    raise ValueError # else

def main():
    try:
        my_email = validate_email()
        print(f"{my_email} is a valid email address")
    except ValueError:
        print("Entered an invalid email")
    print("Thanks for using my validator")
```

EXERCISE: YOU TRY – SAFE DIVIDE

- Remember our divide() function? Rewrite that function, but instead of using try-catch to handle an error when division-by-zero actually happens, use raise to signal an exception if the denominator is 0
  - You may use a ValueError for this exercise.

MIDTERM REVIEW – BIG IDEAS AND TOPICS

MIDTERM: FORMAT

- Where: Remote. Open book. Open IDLE
- Duration: Approximately 4 hours
- Types of Questions:
  - Some multiple choice
  - Trace the flowchart(s)
  - Write the code
    - These will be very small functions – NOT entire programs!

      Similar to practice problems in Canvas

    - Your function(s) must fulfill the specification. Read spec carefully!

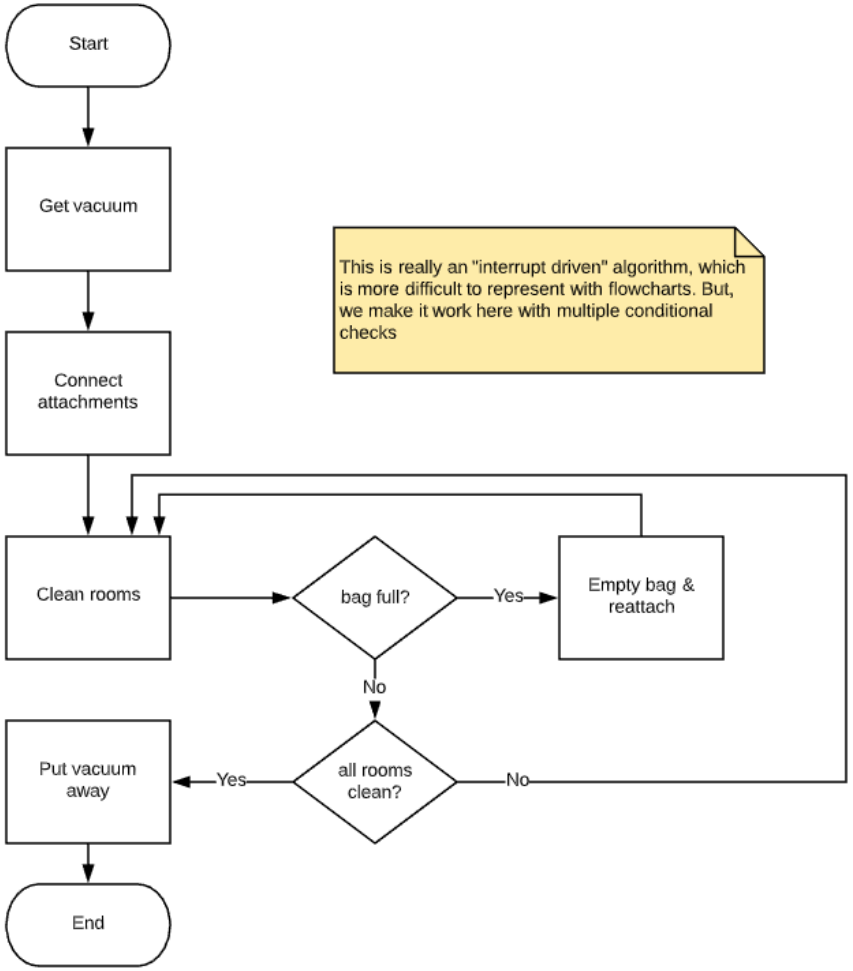
REVIEW: COMPUTATION & ALGORITHMS

- Algorithm (simple definition): An ordered set of unambiguous steps that solves a problem in a finite amount of time
- The definition allows us to talk computationally & implement using machines
- The definition also allows us to explore algorithms manually. Algorithms may or may not be automated using computers

REVIEW: FLOWCHARTS

- Graphical Representation of an algorithm
  - **Think first**: Flowcharts can help us visualize our algorithms before “putting pen to paper” or starting to develop code
  - **Verify**: Flowcharts are at a high enough level (abstraction) that non-programmers can get a sense of what’s going on. So, we can test and verify our ideas with other people, even if they are not technically savvy

REVIEW: FLOWCHARTS



As humans, we can work with some ambiguity

Computers need more precision & less ambiguity

ARITHMETIC OPERATORS

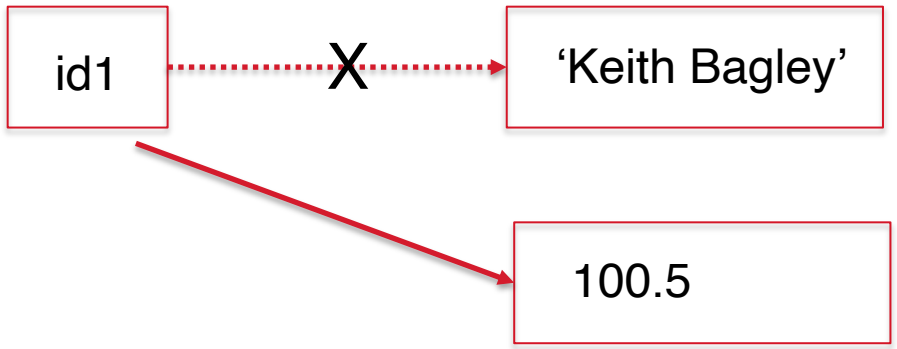
- Last week we discussed some of the operators we use in Python to perform calculations. Here's a list of the rest

Symbol	Operator	Example	Result
-	Negation	-5	-5
+	Addition	11 + 3.1	14.1
-	Subtraction	5 - 19	-14
*	Multiplication	8.5 * 4	34.0
/	Division	11 / 2	5.5
//	Integer Division	11 // 2	5
%	Remainder	8.5 % 3.5	1.5
**	Exponentiation	2 ** 5	32

VARIABLES HAVE TYPES

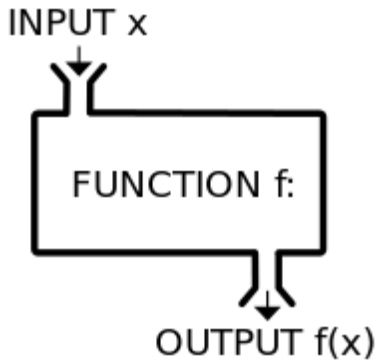
- Python variables hold data of a certain type
- The value held by the variable may change
- Python is “dynamically typed” so the type of the data may change

```
>>> name = "Keith Bagley"
>>> name
'Keith Bagley'
>>> print(name)
Keith Bagley
>>> "Dr. " + name
'Dr. Keith Bagley'
>>> name
'Keith Bagley'
>>> name = "Dr. " + name
>>> name
'Dr. Keith Bagley'
>>> name = 100.5 # not a good variable name but...
>>> name
100.5
>>>
```



FUNCTIONS

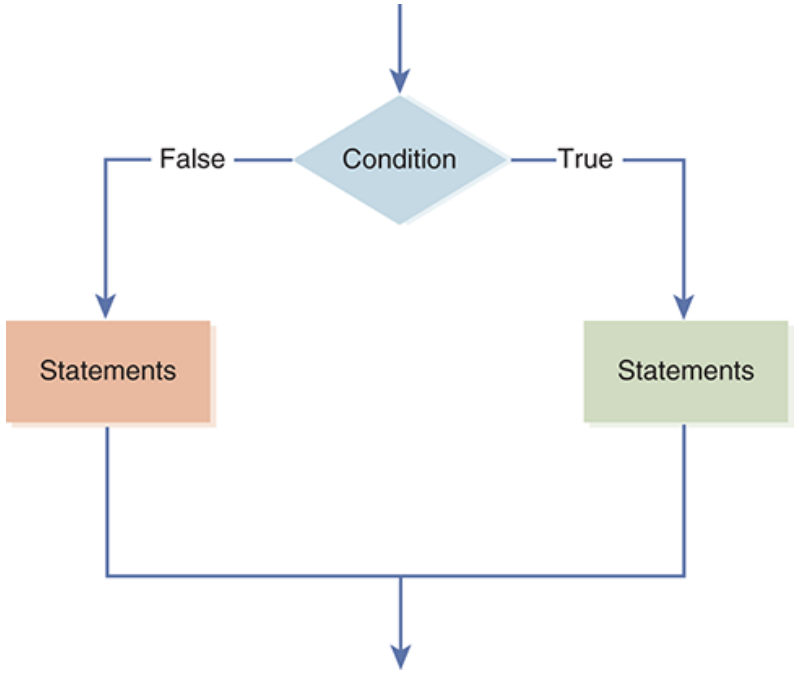
This is a function definition. It is the “recipe” for what to do (the algorithm)
It does not “start to cook” until we actually invoke (call) it




```
def divide(x,y):
    ''' Function: divide, takes 2 numbers x, y and returns result of x/y'''
    result = x / y
    return result
```

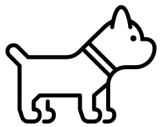
IF-ELSE

- If allows us to change the program flow & behavior based on a condition




```
if <condition>:
   <statements>    # execute if condition is true
else:
   <statements>    # execute if condition is false

# else is optional!
```



If you like ice cream,
then let's go to Cold Stone Creamery

otherwise let's go home

SEQUENCES: LISTS, STRINGS, TUPLES

- A List is a named collection of data
  - Similar to the to-do and shopping lists you might use
  - Allows us to refer to a group of data values by ONE name
- Strings and Tuples are immutable sequences, similar to lists


```
>>> list_1 = [10, 20, 30, 40]     # a list of 4 integers
>>> list_2 = ['yummy', 'rummy', 'tummy']  # a list of 3 strings
>>> tupperware = (1,2,3) # a tuple of 3
```

LISTS & LOOPS: BY INDEX/POSITION OR VALUE

```
food = ["grapes", "apples", "snickers"]
i = 0
while i < len(food):
    food[i] = "good" + food[i]
    i = i + 1
print(food)

food = ["grapes", "apples", "snickers"]
for i in range(len(food)):
    food[i] = "good" + food[i]
print(food)

food = ["grapes", "apples", "snickers"]
for each in food:
    each = "good" + each
print(food)
```

By index/position. The actual value can be modified

By index/position. The actual value can be modified

By value: a copy is made. Immutable

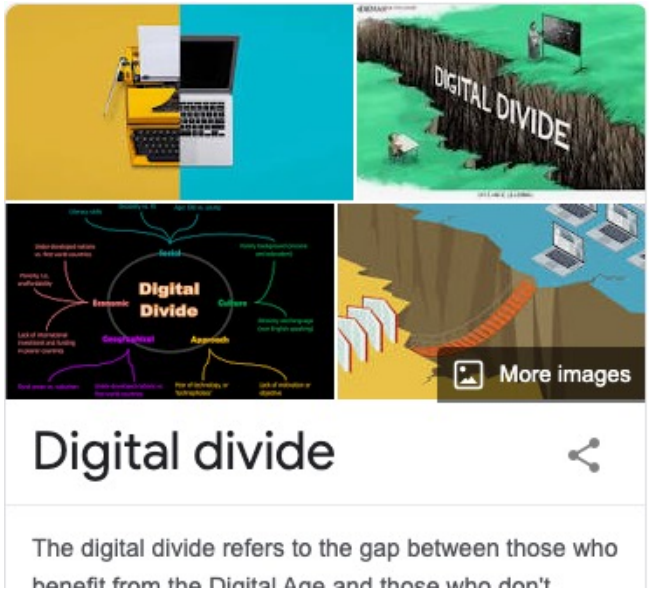
RECITATION THIS WEEK: MIDTERM REVIEW

FACES OF C.S.: LYNDSEY SCOTT

American model, **software developer**, and actress. She was the first African American model to sign an exclusive runway contract with Calvin Klein. Between modeling assignments for prestigious fashion houses like Gucci, Prada, and Victoria's Secret, she writes mobile apps for iOS devices. She has been credited for challenging the stereotypes about models and computer programmers, and for inspiring young women to code

https://en.wikipedia.org/wiki/Lyndsey_Scott

DIGITAL DIVIDE?



The idea of the "digital divide" refers to **the growing gap between the underprivileged members of society**, especially the poor, rural, elderly, and handicapped portion of the population who do not have access to computers or the internet; and the wealthy, middle-class, and young Americans living in urban and suburban ...

https://cs.stanford.edu › projects › digital-divide › start

Q: Should basic technology (e.g. broadband access) be as expected for “thriving” as clean water is?

Q & A

- Questions?

THANKS!

- Stay safe, be encouraged, & see you next week!