

CS 134 Lecture 16:

Testing & Files

Announcements & Logistics

- No HW due tonight
- Lab 5 today/tomorrow (no prelab); due Friday noon (*after* midterm!)
- **Midterm reminders:**
 - **Review: Tonight 3/11** from 7-9pm
 - **Exam Thurs 3/14** from 6-7:30pm OR 8-9:30pm
 - Both exam and review are in Bronfman Auditorium
 - Exam only includes material up to last week (sets)
 - Sample Exam posted! Solutions give some POSSIBLE answers (other correct answers may exist)
- New Instructor Help Hours Schedule
 - Wednesday 1-4, Thursday 1-4

Do You Have Any Questions?

Last Time

- Explore another new Python type:
 - sets: *mutable unordered* collection
- Use tuples and sets in example functions

Lists, Tuples, Sets

Lists

- `list()`, `[]`
- **Mutable**
- Ordered
- Store any type, even a list
- Dynamic datasets
- Indexing, slicing, `len`, `in` & `not in`, iteration `in for loop`

Tuples

- `tuple()`, `()`
- **Immutable**
- Ordered
- Store any type, even a tuple
- Static ordered collections
- Indexing, slicing, `len`, `in` & `not in`, iteration `in for loop`

Sets

- `set()`
- **Mutable**
- Unordered
- Immutable types, no duplicates
- Unordered collections
- No indexing/slicing
- `len`, `in` & `not in`, iteration `in for loop`

Finish Set Example From Class:

Using set to implement `get_candidates()`

Today's Plan

- Discuss **testing** and **debugging** strategies (more on this in Lab 5)
- Start discussion on how to read from and iterate over **files**
 - Tuple example to solve Madlibs problem

Testing and Debugging

Testing vs Debugging

- **Testing:** Ensuring that your code meets the specifications
 - Has the correct (expected) behavior on all inputs
- **Debugging:** Process of finding and fixing bugs in your code
 - These can be syntax errors, runtime errors, logical errors
 - As part of testing overall



Types of Errors

- **Syntax errors:** Occurs when the code does not follow Python syntax rules
 - E.g., Missing colon after function definitions or if statements
 - Indentation issues
- **Runtime errors:** Occur when functions are invoked and variables are replaced by their values at runtime
 - E.g., divide by zero
- **Logical errors:** Occur due to mistakes in logic while implementing the functionality

How to Approach Debugging

- read the Python error messages
 - tell you line numbers and program trace
- print statements!
- rubber duck debugging

Example Program **Debugging**

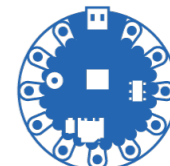
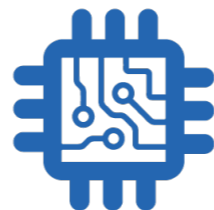
- `debugging.py` implements a "simple" sorting algorithm (it's actually complex code, but there are even more complex ways to sort a list).
- The sorting algorithm is correct, but it doesn't work!
- Sorting strategy:
 - Find the smallest element in the list
 - Swap that element with the first element in the list
 - Now the list has one element in its final location!
 - Repeat for `list[1:]`, then `list[2:]`, and so on...

`debugging.py`

Testing So Far In Lab

- We have already seen two ways to test a function
 - You can run your code 1) interactively or 2) as a script
- We've also seen how to import functions we wrote in one file, into a separate testing file (like **runtests.py**)
 - This is still running code as a script!
- There are also more automated ways to run these tests (which is what we do in this class for the 90+ weekly labs!)

runtests.py



runtests.py

- We always provide *some* tests for you in **runtests.py**
- The Lab Instructions often suggest additional tests to run in interactive python
- As a computer scientist, you should always be thinking of so-called **edge cases** that may break your code
 - Stress test!
 - ...but how might we do that?

runtests.py

- **runtests.py** typically has ~5 sections:
 - imports, datasets, our tests, your tests, and the test runner
- **Imports:** makes the functions we wrote in other files usable in this one
- **Datasets:** loads-in some handy datasets for testing the code
- **Our Tests:** Some basic tests we provide to see if your functions have minimal functionality
- **Your Tests:** A place where you should add your own tests, for ensuring maximal functionality
- **Test Runner:** Gathers command-line input (e.g., 'q2') and runs the tests specified in the body

runtests.py

- **To add your own tests:** copy a similar example from our tests into the function body, update the parameters and print statements.

```
def my_first_choice_votes_test():  
    # Replace the following line with your own test!  
    print("YOU HAVEN'T YET WRITTEN YOUR OWN TEST FOR first_choice_votes!")
```

```
def my_first_choice_votes_test():  
    result = first_choice_votes(aamir_beth_chris_ballots())  
    print("first_choice_votes(aamir_beth_chris_ballots())")  
    print("    should return: ['Aamir', 'Beth', 'Chris', 'Aamir']")  
    print("    yours returned: " + str(result))
```

- If you're using a function header already provided in the .py file, then this function will run!

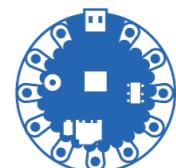
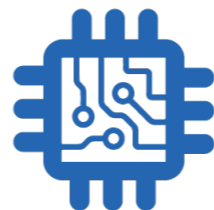
runtests.py

- If you've made a brand **new** function with a new name, you'll need to make sure that function is called in the **Test Runner**

```
if __name__ == "__main__":
    args = get_command_line_args()
    if len(args) == 0: # if there are no command-line arguments
        print("Please specify the question: q1, q2, q3, q4, q5, q6, q7, q8, q9")
    else:
        which_question = args[0] # reads the first command-line argument
        if which_question == "q1":
            first_choice_votes_test1()
            first_choice_votes_test2()
            my_first_choice_votes_test()
```

- Good individual (unit) tests often focus on one aspect of your code
- Comprehensive test suites should include many individual tests and combinations

Reading Data from Files



Working with Files in Python

- File I/O is a very common and important operation
- `open(filename)` is a built-in Python function for working with files
 - `filename` is a path to a file as a **string**
- Using `open()` within a `with ... as` code block, we can **iterate** over the **lines of a text file** just as we iterated over strings and lists in previous lectures

Opening Files: `with ... as`

Path to file on computer as a string

```
with open(filename) as input_file:
```

```
# do something with file
```

Variable name for your file

Note. (syntax) Indentation defines the body of the with block where the file is open. File automatically closed after with...as block.

files-plotting.ipynb

Iterating over Lines in a File

- Within a `with open(filename) as input_file:` block, we can iterate over the lines in the file just as we would iterate over any sequence such as lists, strings, or ranges
- The end of a line in the text file is determined by the special newline character `'\n'`
- Example: We have a text file `mountains.txt` within a directory `data`. We can iterate and print each line as follows:

```
with open("data/mountains.txt") as book:  
    for line in book:  
        print(line)
```

Variable name for your file

0, proudly rise the monarchs of our
Path to file on computer as a string

With their kingly forest robes, to the sky,

Where Alma Mater dwelleth with her chosen bar
'\n' between each line

And the peaceful river floweth gently by.

Iterating over Lines in a File

- Because the end of the line in a file is a newline character `'\n'` and when we `print(a_string)` a newline character is added to the end...we end up with an empty newline between each printed line!

```
with open("data/mountains.txt") as book:  
    result = []  
    for line in book:  
        result += [line]  
    print(result)
```

```
['O, proudly rise the monarchs of our mountain land,\n',  
'With their kingly forest robes, to the sky,\n',  
'Where Alma Mater dwelleth with her chosen band,\n',  
'And the peaceful river floweth gently by.\n']
```

Removing Leading/Trailing Whitespace from a String

- Because the end of the line in a file is a newline character `'\n'` and when we `print(a_string)` a newline character is added to the end...we end up with an empty newline between each printed line!
- Let's write a function that will remove leading and trailing whitespaces.

```
s = '\n \t String with\t different\nspaces.\r\n\t'
```

`'\n'` newline

`'\t'` newline

`'\r'` return

```
spaces = ['\n', '\t', '\r', ' ']
```

```
>>> len('\n')
```

```
1
```

`'\n'` is one character! The backslash *escapes* the character.

Removing Leading/Trailing Whitespace from a String

- Let's write a function that will remove leading and trailing whitespaces.

```
def strip(line):  
    # handle empty line somehow  
    # return line?  
    spaces = ['\n', '\t', '\r', ' ']  
  
    # find where the words start  
    # look at each character  
    # if it's a space...keep looking  
    # keep track of indices looked at  
  
    # find where the word ends  
    # look at each character in reverse  
    # if it's a space...keep looking  
    # keep track of indices looked at  
  
    # return the string between the start and end index
```


Removing Leading/Trailing Whitespace from a String

- Let's write a function that will remove leading and trailing whitespaces.

```
def strip(line):  
    if not line: # handle empty line  
        return line  
    spaces = ['\n', '\t', '\r', ' ']  
  
    # find the first not-space  
    start_index = 0  
    while start_index < len(line) and line[start_index] in spaces:  
        start_index += 1  
  
    # find the last not-space  
    end_index = len(line) - 1  
    while end_index > 0 and line[end_index] in spaces:  
        end_index -= 1  
  
    return line[start_index:end_index+1]
```

```
>>> s = '\n \t String with\t different\nspaces.\r\n\t'  
>>> strip(s)  
'String with\t different\nspaces.'
```

Useful String and List Functions in File Reading

- When reading files, we may need to use some common string and list operations to work with the data.
- We'll learn about the built-in features python has for these later in the semester, but we can write our own with iterating over strings and accumulator variables!
 - `strip(line)`: Remove any leading/trailing white space or “\n”
 - `split(line, ',')`: Separate a **comma-separated** sequence of words and create a list of strings
 - `join(' ', lines)`: Create a single “big” string with words separated by spaces instead of commas
 - `count_appearances(ele, let)`: Count the occurrence of various elements
 - ...and so on!