Lab 7

- Lab 6 Review
- Review for Lab 7

Lab Musings

- As we learn more computer science, we're moving toward a much **higher ratio of thinking to coding**
  - Give yourself the time and room to think
- Going beyond simply correctness in solutions
  - Looking for understanding of good coding practices
    - Testing, readability, usability, documentation, organization, efficiency
      - (not necessarily in that order)

Lab Musings

- Lab benefit: access to other students, lab assistants, and instructor to help
- Lab limitation: may not be the best environment
  - Seems to cause a competitive atmosphere, increased anxiety for some students
  - You have until Friday to complete the lab
  - Work at your pace, **think clearly** and **deeply**

Compare Solutions

```
words = sentence.split()

shorthandList = []
for word in words:
    shorthandList.append(word[0])

shorthand = "".join(shorthandList)

shorthand = shorthand.lower()

print("Shorthand is:", shorthand)
```

```
words = sentence.split()

shorthand=""
for word in words:
    shorthand += word[0]

shorthand = shorthand.lower()

print("Shorthand is:", shorthand)
```

Compare Solutions

```
words = sentence.split()

shorthandList = []
for word in words:
    shorthandList.append(word[0])

shorthand = "".join(shorthandList)

shorthand = shorthand.lower()

print("Shorthand is:", shorthand)
```

Both are valid solutions. I'm not sure which is more efficient in practice.

However, the solution at left has more conceptual complexity (appending to a list and then converting to a string, as opposed to just creating the string).

In general, looking for less complex solutions.

Saw similar, more complex solutions for the password generation problem.

```
words = sentence.split()

shorthand=""
for word in words:
    shorthand += word[0]

shorthand = shorthand.lower()

print("Shorthand is:", shorthand)
```

Generating a Random Password

```
CHOOSE_NUM=0
CHOOSE_LOWER=1
CHOOSE_UPPER=2

password=""
len_password = randint(6,8)

for charPos in range(len_password):
    #determines if character is number, uppercase, or lowercase
    char_type = randint(0,2)
    #for each case, randomly assigns ASCII val
    if char_type == CHOOSE_NUM:
        asciival = randint(48,57)
    elif char_type == CHOOSE_LOWER:
        asciival = randint(97,122)
    elif char_type == CHOOSE_UPPER:
        asciival = randint(65,90)

    char = chr(asciival)
    password += char
```

Define outside of `for` loop

+ Good variable names

Even better to use constants for ASCII values. (I'm short on space)

Consider:
`MIN_NUM=ord('0')`

Review Caesar Cipher

- Consider the following solutions

```
for char in message:
    asciiVal = ord(char)
    if asciiVal == 32:
        …
    else:
        …
```

Which is easier to read and understand?

```
for char in message:
    if char == " ":
        …
    else:
        …
```

Review Caesar Cipher

- Consider the following solutions

```
for char in message:
    asciiVal = ord(char)
    if asciiVal == 32:
        …
    else:
        …
```

I know what " " means.
I don't immediately know what 32 means.
Lesson: prefer words over numbers.

```
for char in message:
    if char == " ":
        …
    else:
        …
```

Caesar Cipher with Files

- High-level description explaining what you're doing at the top of the program
- How to debug
  - Look at the input files
- Common issues
  - Not handling new lines ("\n") in the file
    - Similar to handling spaces
  - Close files as soon as possible

Review

- What is the keyword we use to create a new function?
- How do we get output from a function?
- What happens in the program execution when a function reaches a `return` statement?
- Why do we write functions?
- Why do we write functions?
- What makes a good function?
- How should you comment your functions?
- What is the name for the process for changing a program to improve readability/organization/readability without changing functionality?

Review: Functions

What does this program do?
What is the control flow/execution path?

```
def main():
    first = eval(input("Enter the first number: "))
    second = eval(input("Enter the second number: "))
    computedVal = myFunction(first, second)
    print("The answer is", computedVal)

def myFunction(x, y):
    result = x*x + y*y + 12
    return result

main()
```

What variables can function "see" here? What vars can't it see?

Review: Practice

- What is the output of this program?
  - Example: user enters 4

```
def main():
    num = eval(input("Enter a number to be squared: "))
    squared = square(num)
    print("The square is", squared)
    print("The original num was", n)

def square(n):
    return n * n

main()
```

practice2.py

Testing Functions

1. Create test cases
   - Input, expected output
2. Write a function that creates lists of the input and expected output and automatically tests your function
3. Call the function to test your function
4. Iterate
   - Add additional test cases if needed to help debug your function

Review: Testing Functions

```
def testBinaryToDecimal():
    """Test the binaryToDecimal function.
       Displays the correctness or incorrectness of the
       function.
       Nothing is returned."""

    paramInputs = ["0", "1", "10", "1001", "10000"]
    expectedResults = [ 0, 1, 2, 9, 16]
    for index in range(len(paramInputs)):
        paramInput = paramInputs[index]
        expectedResult = expectedResults[index]
        actualResult = binaryToDecimal(paramInput)
        if actualResult != expectedResult:
            print("**ERROR!**", paramInput, "should be", \
                  expectedResult)
            print("Instead, got", actualResult)
        else:
            print("Success on binary to decimal conversion for",\
                   paramInput, "-->", actualResult)
```

Call function to test: `testBinaryToDecimal()`

Review: Writing a "Good" Function

- Should be an "intuitive chunk"
  - Doesn't do too much or too little
  - If does too much, try to break into more functions
- Should be reusable
- Always have comment that tells what the function does

Writing Comments for Functions

- Good style: Each function ***must*** have a comment
  - Describes functionality at a high-level
  - Include the *precondition*, *postcondition*
  - Describe the parameters (their types) and the result of calling the function (precondition and postcondition may cover this)

Writing Comments for Functions

- Include the function's pre- and post- conditions
- **Precondition**: Things that must be true for function to work correctly
  - E.g., `num` must be even
- **Postcondition**: Things that will be true when function finishes (if precondition is true)
  - E.g., the returned value is the max

Example Comment

- Describes at high-level
- Describes parameters

```
def printVerse(animal, sound):
    """
    Prints a verse of Old MacDonald, plugging in the
    animal and sound parameters (which are strings),
    as appropriate.
    """
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a " + animal + EIEIO)
    …
```

Comment style: **Docstring** "documentation string"

Comments from docstrings show up when you use `help` function

Pre/Post Conditions

```
def binaryToDecimal( binary_string ):
    """
    pre: binary_string is a string that contains
    only 0s and 1s
    post: returns the decimal value for the binary
    string
    """
    dec_value = 0
    for pos in range( len( binNum ) ):
        exp = len(binNum) - pos - 1
        bit = int(binNum[pos])

        # compute the decimal value of this bit
        val = bit * 2 ** exp

        # add it to the decimal value
        decVal += val

    return dec_value
```

Function comments

```
def printHeadings():
    """displays table column headings"""
```

Good. Describes function at high level

```
def printHeadings():
    """defines the printHeader function"""
```

Not descriptive.
Says what *you're* doing, not what ***function*** does
Need to tell programmer how to use function

Getting Documentation

- `dir`: function that returns a list of methods and attributes in an object
  - `dir(<type>)`
- `help`: get documentation

- In the Python shell
  - `help(<type>)`
  - `import <modulename>`
  - `help(<modulename>)`

Where is Documentation Coming From?

- Comes from the code itself in "**doc strings**"
  - i.e., "documentation strings"
- Doc strings are simply strings *after* the function header
  - Typically use triple-quoted strings because documentation goes across several lines

```
def printVerse(animal, sound):
    """prints a verse of Old MacDonald,
    filling in the strings for animal
    and sound """
```

Summary "Good" Function

- Reusable functionality
- Good function name
- Good parameter names
- Good documentation
  - Well-described input, output

Review: Refactoring Converting Functionality into Functions

1. Identify functionality that should be put into a function
   - What is the function's input?
   - What is the function's output?
2. Define the function
   - Write comments
3. Call the function where appropriate
4. Create a `main` function that contains the "driver" for your program
   - Put at top of program
5. Call `main` at bottom of program

TOP-DOWN DESIGN

Designing Code

- 1st Approach: Bottom-up
  - Create functions
  - Call functions
- 2nd Approach: Refactoring
  - Write code
  - Refactor code to have functions
  - Call those functions
- 3rd approach: Top-down Design
  - Write code, calling functions
  - Write "stub" functions
  - Fill-in functions later

Top-Down Design: Alternative Approach to Development

1. Create overview, e.g., in `main`
2. Define functions later

```
def main():
    # get the binary number from the user, as a string
    binNum = input("Please enter a binary number: ")
    isBinary = checkBinary(binNum)
    if not isBinary : # equivalent to isBinary == False
        print(binNum, "is not a binary number.")
        sys.exit()

    decVal = binaryToDecimal(binNum)
    print(binNum, "is", decVal)
```

Benefits:
- Know what functions you need
- Know the requirements for your functions
  - What is each function's input, output

Problem: Create a Summary Report

- **Given**: a file containing students names and their years (first years, sophomore, junior, or senior) for this class
- **Problem**: create a report (in a file) that says the year and how many students from that year are in this class, on the same line.

writeSumReport.py

Problem: Create a Summary Report

- **Given**: a file containing students names and their years (first years, sophomore, junior, or senior) for this class
- **Problem**: create a report (in a file) that says the year and how many students from that year are in this class, on the same line.

Pseudocode for program

```
def main():
    # get name of data file
    # open output file
    for searchTerm in searchTerms:
        numFound = numOccurrences( searchTerm, dataFileName )
        outputFile.write("%s %d\n" % (searchTerm, numFound))
    # close output file
```

Example of top-down design:
- Can fill in details, e.g., the comments, the function numOccurrences

Development Advice

- Build up your program in steps
  - Always write small pieces of code
  - Test *function* separately from other code, using a test function
  - Test, debug. **Repeat**
- Development Options:
  - Refactor:
    - Write function body as part of `main`, test
    - Then, separate out into its own function
  - Top-down design
  - Bottom-up design

May use more than one approach in a program

Example: Could still refactor after using these options

Lab 7

- Function practice
- Defining functions (refactoring)
- File practice