Lab 4

- Review Lab 3
  - Run Animations!
- Function review

Lab 3

- Iterative Fibonacci Sequence was a question on one student's interview

Lab 3 Feedback

- Continuing to get tougher in grading
  - Paying more attention to style (e.g., variable names), efficiency, readability, good output
  - High-level descriptions
  - More strict on adhering to problem specification
  - Constants
  - Demonstrate program **more than once** if gets input from user or outcome changes when run again
    - Find errors before I do!

Program Organization

```
# high-level description
# author name

import statements

CONSTANT_DEFNS = …

program_statements ...
program_statements ...
program_statements …
```

Program Organization

```
# high-level description
# author name

import statements

CONSTANT_DEFNS = …

def main():
    statements…
    statements...

def otherfunction():
    statement...
```


Discussion

- Why link from your Lab 2 page to your home page?

Run Animations

Refactoring:
Converting Functionality into Functions

1. Identify functionality that should be put into a function
   - What should the function do?
   - What is the function's input?
   - What is the function's output (i.e., what is returned)?
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

Animate Circle Shift

- What it does: circle is animated, moving to a new position
- Input: circle, new center point
- Output: nothing returned

WHAT MAKES A FUNCTION GOOD?

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 sumList(listOfNumbers):
    """
    Pre: listOfNumbers is a list of numbers.
    Post: returns the sum of the numbers in the list
    """
    …
```


TESTING FUNCTIONS

Testing Functions

- Functions make it easier for us to test our code
- We can write code to test the functions
  - Test Case:
    - Input: parameters
    - Expected Output: what we expect to be returned
  - We can verify the function programmatically
    - "programmatically" – automatically execute test cases and verify that the actual returned result is what we expected
    - No user input required!

Example: Testing sumEvens

```
import test

def main():
    actual = sumEvens( 10 )
    expected = 20
    test.testEqual( actual, expected )

def sumEvens(limit):
    total = 0
    for x in range(0, limit, 2):
        total += x
    return total
```

This is the actual result from our function

This is what we expect the result to be

testSumEvens.py

Evening Help Student Assistants

- 7 – 9 p.m., in Parmly 405, Sun - Thurs

Lab 4 Overview

- Filling in a function, testing functions
- Refactoring
- Writing a program with a function from scratch