

## Lab 4

- Review Lab 3
  - Run Animations!
- Function review

## Lab 3

- Iterative Fibonacci Sequence was a question on several students' interviews

## 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...
```

## Lab 2 Feedback: Common Issues

Which solution is more efficient (does less “work”)?

```
operand1=6
for operand2 in range(1, 14):
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

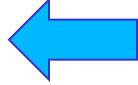
vs

```
for operand2 in range(1, 14):
    operand1=6
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

## Lab 2 Feedback: Common Issues

Which solution is more efficient (does less “work”)?

```
operand1=6
for operand2 in range(1, 14):
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```



vs

```
for operand2 in range(1, 14):
    operand1=6
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

← Additional assignment each time through loop

## Lab 2 Feedback: Common Issues

Which solution is simpler?

```
operand1=6
for operand2 in range(1, 14):
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

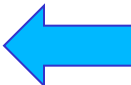
vs

```
operand1=6
operand2=0
for x in range(13):
    operand2 = x + 1
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

## Lab 2 Feedback: Common Issues

Which solution is simpler?

```
operand1=6
for operand2 in range(1, 14):
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```



vs

```
operand1=6
operand2=0
for x in range(13):
    operand2 = x + 1
    result = operand1 % operand2
    print(operand1, "%", operand2, "=", result)
```

More code makes  
solution more difficult  
to understand

## Animation Feedback

- If moving multiple objects together
  - Move *all* the objects, then sleep
  - Otherwise, animation looks choppy
- Could use a list with the **for** loop, as discussed for several sections in the textbook
  - Simplifies and reduces code

```
for object in [ myObj1, myObj2, myObj3 ]:
    object.move()
    sleep(.001)
```

## Run Animations

## Review

- 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

## 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. Test the function programmatically
4. Call the function where appropriate
5. Create a `main` function that contains the "driver" for your program
  - Put at top of program
6. Call `main` at bottom of program



## Review

- How can we programmatically test functions?

## test module's `testEqual` function

- Example from yesterday

```
def testWinPercentage():  
    test.testEqual( calculateWinPercentage(0, 1), 0 )  
    test.testEqual( calculateWinPercentage(2, 2), .5 )  
    test.testEqual( calculateWinPercentage(3, 7), .3 )  
    test.testEqual( calculateWinPercentage(1, 0), 1 )  
  
testWinPercentage()
```

After confirming that the function works...

## test module's testEqual function

- Example from yesterday

```
def testWinPercentage():  
    test.testEqual( calculateWinPercentage(0, 1), 0 )  
    test.testEqual( calculateWinPercentage(2, 2), .5 )  
    test.testEqual( calculateWinPercentage(3, 7), .3 )  
    test.testEqual( calculateWinPercentage(1, 0), 1 )  
  
# testWinPercentage()  
main()
```

Comment out call to test function.  
Call main.

## Lab 4 Overview

- Calling functions defined in the same program
- Refactoring code
- Modifying function definitions
- Testing functions
- Creating a module
- Writing a program with a function from scratch