Objectives

- Wrap up functions
  - Testing functions
  - Default parameters
- Modules
- Exception Handling

Lab Review: Functions

- #1 Call/use functions that were already defined
  - Create 4 bugs, put in list
- #2 Define function, where I give the function header and one example use
- #3 Take code that you wrote and refactor it to use functions
  - Very common practice
- #5 Start from scratch, creating functions, as appropriate

Lab Review

http://retrievalpractice.org/

- "… we have excellent evidence that students remember material better when they test themselves and try to retrieve information from their own minds. And yet most students still study by reviewing their notes over and over again — probably the least-effective study strategy they can employ. "

http://chronicle.com/article/Small-Changes-in-Teaching-The/235583

Labs

- Main skill you're learning: given a problem, ***find*** and ***apply*** appropriate solutions
  - Often involves reviewing ***examples*** and ***slides***
  - Includes reading/understanding documentation
- Learning good organization, documentation, problem-solving techniques
  - Good to consider why those are best practices
- Make sure you understand what is happening in examples
  - Copy and execute the code
  - Modify the code and execute it to see how your modifications change the execute

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

Lab Review: Design Decisions

- My suggestions for how to design your code (e.g., how to design your functions) are based on some more advanced rules about code organization
  - Reusability
  - Separation of concerns

PARAMETER DEFAULTS

Defaults for Parameters

- Can assign a default value to a parameter
  - In general, in function header, default parameter(s) should come *after* all the parameters that *need* to be defined
- Example: **`range`** function
  - Didn't have to specify `start` or `increment` when calling the function
  - Default `start` = 0
  - Default `increment` = 1

Using Default Parameters

- By default, the `genWinningNum` function could assume that there are 4 numbers

Assigns a value to `numNums` **ONLY IF** not passed a parameter

```
def genWinningNum(numNums=4):
   winNum = ""
   for i in range(numNums):
      winNum += str(randint(MIN_VALUE,MAX_VALUE))
   return winNum
```

Examples of calling function:
```
genWinningNum(6)
genWinningNum()
genWinningNum(4)
```


Immutable vs Mutable Parameters

PASSING PARAMETERS

Passing Parameters

- Only ***copies*** of the actual parameters are given to the function
  - For **immutable** data types — Which are?
- The *actual* parameters in the calling code do not change
- **Swap example:**
  - Swap two values in script
  - Then, put into a function

Immutable Data is Passed by Value

```
def swap(a, b):
    tmp = a
    a = b
    b = tmp
    print(a, b)

x = 5
y = 7

swap(x, y)

print("x =", x)
print("y =", y)
```

swap.py

Immutable Data is Passed by Value

```
def swap(a, b):
    tmp = a
    a = b
    b = tmp
    print(a, b)

x = 5
y = 7

swap(x, y)

# at the end, x is still 5 and y is still 7
print("x =", x)
print("y =", y)
```

This code does not have the desired swapping effect.

Because the integer variables are passed *by value*, the value of x and y are not changed by the call to the swap function.

Lists as Parameters to Functions

If a list that is passed as a parameter into a function is **modified *in* the function**, the list **is modified *outside* the function**

- Lists are **not** passed-by-value/copied
- Different from immutable types (e.g., numbers, strings)

- Parameter is actually a **pointer** to the list in memory

Problem: Sort a list of 3 numbers, in descending order

```
def descendSortNums( listOfNumbers ):
    """
    Sort the given list of numbers in descending order
    """
```

Called as:

```
myList = …
descendSortNums(myList)
print(myList)
```


Descend Sort a List w/ 3 elements

```
def descendSortNums(listOfNumbers):
    listOfNumbers.sort()
    listOfNumbers.reverse()
```

Function does **not** *return* anything. Simply modifies the `listOfNumbers` parameter.

```
def main():
    myList = [1,2,3]
    descendSortNums(myList)
    print(myList)
```

Takeaway

- When you pass lists into functions as actual parameters, the actual parameter can be changed within the function because lists are ***mutable***
- When you pass immutable data types (e.g., strings, ints, floats, …) as actual parameters into functions, the actual parameters cannot be changed.

PROGRAMMATICALLY TESTING FUNCTIONS

Testing Functions

- Functions make it easier for us to test our code
- We can write code to test the functions
  - Input: parameters
  - Output: what is returned
    - We can verify programmatically

What are good tests for `binaryToDecimal(binnum)` and `isBinary(candidate)`?

`binaryToDecimal.test.py`

Testing Functions Discussion

- How does this way of testing compare to what you'd normally do?
- What are the tradeoffs?

Testing at Different Levels

Why is a function that automatically tests one function ***not sufficient/complete*** testing?

Testing at Different Levels

- Why is a function that automatically tests one function not sufficient/complete testing?
  - The test function does not test the user input/the message that is displayed

Testing Summary

- Testing is VERY important
- Want to think about what results you expect first and then compare that to what you actually get
- Want to try a variety of different cases to reveal errors in your code

CREATING MODULES

Looking Ahead

- For Friday
  - Lab 7
  - Broader Issue: CS Education