Objectives

- More on functions
- Prep for lab

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?

Review: Functions

```
CONSTANT = 12

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 + CONSTANT
    return result

main()
```

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

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

Review: Why Functions?

- Organize code
- Easier to read
- Easier to change
- Easier to reuse

Practice

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

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

def square(n):
    return n * n

main()
```

practice1.py

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

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()
```

Error! n does not have a value in function main()

Variable Scope

- Know "lifetime" of variable
  - Only during execution of function
  - Related to idea of "scope"
- What about variables outside of functions?
  - Example: non_function_vars.py

Variable Scope

```
non_func = 2
non_func_string = "aardvark"

def main():
    func()
    print(non_func)
    print(non_func_string)

def func():
    print("In func: nf =", non_func)
    print("In func: nfs =", non_func_string)

main()
non_func = 6
non_func_string = "dog"
print(non_func)
print(non_func_string)
```

non_function_vars.py

WHAT MAKES A GOOD FUNCTION?

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
```

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 """
```

REFACTORING

Refactoring

- After you've written some code and it passes all your test cases, the code is probably still not perfect
- ***Refactoring*** is the process of improving your code *without* changing its functionality
  - Organization
  - Abstraction
    - Example: Easier to read, change
  - Easier to test
- Part of iterative design/development process
- Where to refactor with functions
  - Duplicated code
    - "Code smell"
  - Reusable code
  - Multiple lines of code for one purpose

Refactoring: Converting Functionality into Functions

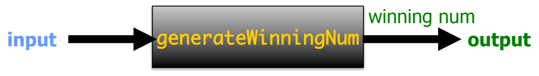
1. Identify functionality that should be put into a function
   - What is the function's input?
   - What is the function's output?
2. Define the function
   - Write comments
3. Call the function where appropriate
4. Create a `main` function that contains the "driver" for your program
   - Put at top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

Refactoring Practice

- `pick4num.py`

- Where are places that we can refactor and add functions?

Generate Winning Number



- Input:
  - Options: none; number of digits; range on random numbers
  - Tradeoffs: more general (more parameters), more difficult to use
- Output: winning number

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`

Debugging Advice

- Build up your program in steps
  - Always write small pieces of code
  - Test, debug. **Repeat**
- Write function body as part of `main`, test
  - Then, separate out into its own function
  - Similar to process using in lab probs
- Test function separately from other code

TOP DOWN DESIGN

Designing Code

- 1st Approach
  - Create functions
  - Call functions
- 2nd Approach
  - Write code
  - Refactor code to have functions
  - Call those functions
- Time for 3rd approach…
  - Write code, calling functions
  - Write “stub” functions
  - Fill-in functions later

Top-Down Design: Alternative Approach to Development

1. Create overview
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)
```

DEAL OR NO DEAL

Lab 7: *Deal or No Deal* Overview

- Have 26 cases with various amounts of money
  - Amounts are known
- Player selects a case (hope has the big jackpot)
- In each round, player opens up cases
  - Reveals amounts that are not in the case they chose
- Banker makes an offer to buy the case
- Player decides if want to take the deal
  - Is the offer more than what is in the case?
  - Make decision based on amounts that haven’t been opened yet
- Game ends when only one more case to open (two amounts on board) or player takes the deal.

Implementing *Deal or No Deal*

- Given: partial solution in code
  - Complete `main()` function, some additional functions
- Your job:
  - Read, understand given code
  - Fill in the functions for a complete solution

Modeling *Deal or No Deal*

How can we represent that a case has been opened?

- Cases, numbered 0 to 25
  - Have dollar amounts in them

1000000	1000	5	…	750000	value
0	1	2		25	case/ position

- Board
  - Which dollar amounts have been chosen, which are still in play

.01	1	5	…	1000000	value
0	1	2		25	position

Modeling *Deal or No De*

`CHOSEN = -1`
means case opened:
Don't display on board, Don't allow user to select again

- Cases, numbered 0 to 25
  - Have dollar amounts in them

1000000	1000	5	…	**CHOSEN**	value
0	1	2		25	case/ position

- Board
  - Which dollar amounts have been chosen, which are still in play

.01	**CHOSEN**	5	…	1000000	value
0	1	2		25	position

Functionality

- Read in values contained in cases from a file
  - What data type should these numbers be?
- Have user select from remaining cases
  - Make sure choice is valid
- Display remaining cases
  - Print four to a row
- Display remaining amounts on board
  - Left column is smaller amounts

How to print remaining cases?

- Cases, numbered 0 to 25
  - Have dollar amounts in them

1000000	1000	5	…	CHOSEN	value
0	1	2		25	case/ position

- Board
  - Which dollar amounts have been chosen, which are still in play

.01	CHOSEN	1000	…	-1	value
0	1	2		25	position

This Week

- Lab 7
  - Functions
    - Refactoring, testing
  - Deal or no deal
    - Lists, top-down design
- Broader Issue: Digital Humanities
- Monday, March 12
  - Katherine Crowley talk at 7:30 p.m. in Stackhouse
  - 10 pts extra credit for write up on Sakai

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

```
x = 5        x = 7
y = 7   →    y = 5
```

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

```
# order list such that list3[0] >= list3[1] >= list3[2]
def descendSort3Nums( list3 ):
```

Called as:

```
list = …
descendSort3Nums(list)
print(list)
```

How implemented with list methods?
Can we do this using only 3 comparisons?

Descend Sort a List w/ 3 elements

```
def descendSort3Nums(list3):
   if list3[1] > list3[0]:
       # swap 'em
       tmp = list3[0]
       list3[0] = list3[1]
       list3[1] = tmp

   if list3[2] > list3[1]:
       tmp = list3[1]
       list3[1] = list3[2]
       list3[2] = tmp

   if list3[1] > list3[0]:
       tmp = list3[0]
       list3[0] = list3[1]
       list3[1] = tmp
```

```
def main():
     list = [1,2,3]
     descendSort3Nums(list)
     print(list)
```

Function does **not** *return* anything. Simply modifies the `list3` parameter.