Objectives

- Defining Functions

Functions

- We've used functions
  - Built-in functions: `len`, `input`, `raw_input`
  - Functions from modules, e.g., `math` and `random`
- Today, we'll learn how to **define our own functions**!

Review: Functions

- Function is a **black box**
  - Implementation doesn't matter
  - Only care that function generates appropriate output, given appropriate input
- Example:
  - Didn't care how `raw_input` function was implemented
  - Use: `user_input = raw_input(prompt)`

Creating Functions

- A function can have
  - 0 or more inputs
  - 0 or 1 outputs
- When we define a function, we know its inputs and if it has output

Writing a Function

- I want a function that averages two numbers

• What is the input to this function?
• What is the output to this function?

Writing a Function

- I want a function that averages two numbers
- What is the input to this function?
  - The two numbers
- What is the output to this function?
  - The average of those two numbers, as a float

These are key questions to ask yourself when designing your own functions.
• Inputs: parameters
• Output: what is getting returned

Comparison of Code Using Functions

- Without functions: `menu_withoutfunc.py`
- With functions: `menu_withfunctions.py`

How do the two programs compare in terms of
- Length? (all code and just the "main" code)
- Readability?

Why Write Functions?

- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function (*maintainability*)

Similar to benefits of OO Programming

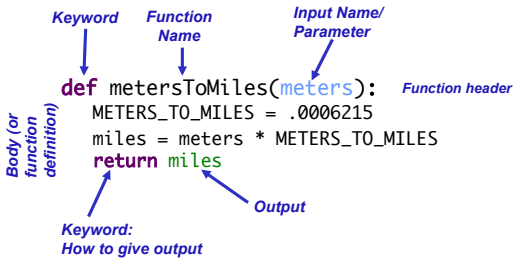
Refactoring

- Duplicated code is often a symptom of when we should make a function
  - Called a "code smell"
- Example code – lab2, prob4
  - Any place to make a function?
  - Any place that has some useful code that we may want to reuse?

Convert meters to miles

- Input: meters
- Output: miles

Syntax of Function Definition



Calling your own functions

Same as calling someone else's functions …

```
miles = metersToMiles(100)
```

Output is assigned to `miles`

Function Name

Input

Functions: Similarity to Math

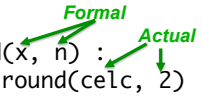
- In math, a function definition looks like:
  - $f(x) = x^2 + 2$
- Plug values in for x
  - $f(3) = 3^2 + 2 = 11$
  - 3 is your input, assigned to x
  - 11 is output

Parameters

- The inputs to a function are called ***parameters*** or ***arguments***, depending on the context
- When ***calling***/using functions, arguments must appear in same order as in the function header
  - Example: `round(x, n)`
    - **x** is the `float` to round
    - **n** is `int` of decimal places to round **x** to

Parameters

- **Formal Parameters** are the variables named in the function definition
- **Actual Parameters** or **Arguments** are the variables or literals that really get used when the function is called.



Defined: `def round(x, n) :`
Use: `roundCelc = round(celc, 2)`

Formal & actual parameters must match in ***order***, ***number***, and ***type***!

Passing Parameters

- Only **copies** of the actual parameters are given to the function for **immutable** data types
  - Immutable types: what we've talked about so far
    - Strings, integers, floats
- The actual parameters in the *calling* code **do *not*** change

Function Output

- When the code reaches a statement like

```
return x
```

  - The function stops executing
  - x is the **output *returned*** to the place where the function was called
- For functions that don't have explicit output, `return` does not have a value with it, e.g.,

```
return
```

  - Optional: don't *need* to have `return`
    - Function *automatically* returns at the end

Example Functions

- `userPBPref(username)`
  - For the given user, returns the amount of PB they want on their sandwich
  - Input: ?
  - Output: ?
- `spread(condiment, amount_in_TB, sandwich)`
  - Spreads given amount of condiment on sandwich
  - Input: ?
  - Output: ?

Example Functions

- `userPBPref(username)`
  - For the given user, returns the amount of PB they want on their sandwich
  - Input: `username`
  - Output: the user's PB preference
- `spread(condiment, amount_in_TB, sandwich)`
  - Spreads given amount of condiment on sandwich
  - Input: `condiment, amount_in_TB, sandwich`
  - Output: no output
    - State of sandwich changes → now has condiment on it

CONTROL FLOW WITH FUNCTIONS

Flow of Control

- When code calls a function, the program jumps to the function and executes it
- After executing the function, the computer returns to the same place in the *calling code* where it left off

Calling code:

```
# Make conversions
dist1 = 100
miles1 = metersToMiles(dist1)
```

`dist1` (100) is assigned to `meters`

```
def metersToMiles(meters) :
    M2MI=.0006215
    miles = meters * M2MI
    return miles
```


Flow of Control

```
def max(num1, num2):
    result = 0
    if num1 >= num2:
        result = num1
    else:
        result = num2
    return result

x = 2
y = input("Enter a number: ")
z = max(x, y)
print "The max is", z
```


Flow of Control

```
def max(num1, num2):
    result = 0
    if num1 >= num2:
        result = num1
    else:
        result = num2
    return result

x = 2 ← Program starts executing here
y = input("Enter a number: ")
z = max(x, y)
print "The max is", z
```

What does this function do?

Function definitions: Save functions for later use, nothing executed

Flow of Control

```
def max(num1, num2):
    result = 0
    if num1 >= num2:
        result = num1
    else:
        result = num2
    return result

x = 2 ← Program starts executing here
y = input("Enter a number: ")
z = max(x, y)
print "The max is", z
```

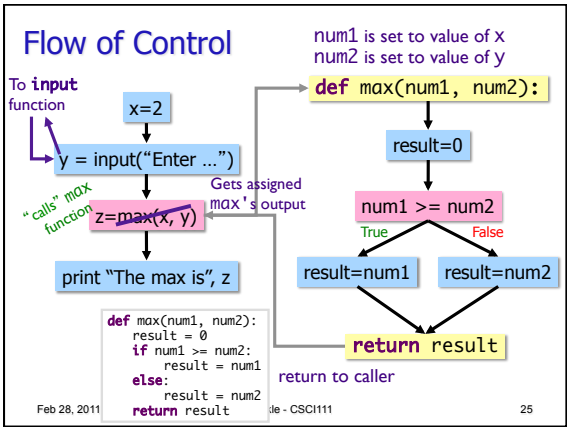
To **input** function

x=2

y = input("Enter ...")

z=max(x, y)

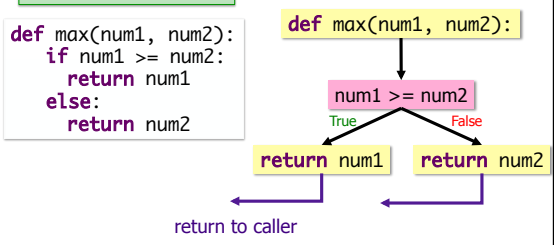
Flow of Control



Flow of Control: Using `return`

Is this implementation of the function correct?

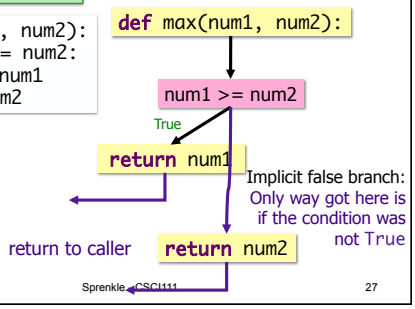
```
def max(num1, num2):
    if num1 >= num2:
        return num1
    else:
        return num2
```



Flow of Control: Using `return`

Is this implementation of the function correct?

```
def max(num1, num2):
    if num1 >= num2:
        return num1
    return num2
```



Function Input and Output

- What does this function do?
- Identify function's input and output

```
def printVerse(animal, sound):
    print BEGIN_END + EIEIO
    print "And on that farm he had a " + animal + EIEIO
    print "With a " + sound + ", " + sound + " here"
    print "And a " + sound + ", " + sound + " there"
    print "Here a", sound
    print "There a", sound
    print "Everywhere a " + sound + ", " + sound
    print BEGIN_END + EIEIO
    print
```


Function Input and Output

- 2 inputs: **`animal`** and **`sound`**
- 0 outputs
  - *Displays* something but does not **`return`** anything

```
def printVerse(animal, sound):
    print BEGIN_END + EIEIO
    print "And on that farm he had a " + animal + EIEIO
    print "With a " + sound + ", " + sound + " here"
    print "And a " + sound + ", " + sound + " there"
    print "Here a", sound
    print "There a", sound
    print "Everywhere a " + sound + ", " + sound
    print BEGIN_END + EIEIO
    print
```

Function exits here

PROGRAM ORGANIZATION

Where are Functions Defined?

- Functions can go inside of program script
  - If no **`main`**() function, defined ***before*** use/called
    - Example from lab2.4.py
  - If `main()` function, defined anywhere in script
    - More in a bit…
- Functions can go inside a separate **module**
  - Example: `menu.py`
  - More on Wednesday

Program Organization: `main` function

- In many languages, you put the "driver" for your program in a **`main`** function
  - You can (and should) do this in Python as well
- Typically **`main`** functions are defined at the top of your program
  - Readers can quickly see overview of what program does
- **`main`** usually takes no arguments
  - Example: `def main():`

Using a `main` Function

- Call **`main()`** at the bottom of your program
- Side effects:
  - Do not need to define functions before **`main`** function
  - **`main`** can "see" other functions
  - Note that **`main`** is a function that calls other functions
    - *Any* function can call other functions

Example program with a main()

- oldmac.py

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

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 (more in a bit)
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

`binaryToDecimal.py`

Refactoring: Converting Functionality into Functions

- `binaryToDecimal.py` Functionality
  1. Converting from binary to decimal
  2. Checking if a string contains only binary numbers

Summary: Why Write Functions?

- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function (*maintainability*)

Similar to benefits of OO Programming

This Week

- Tuesday: Lab 6
  - More String practice: methods and more
  - Encryption – Caesar Ciphers
  - Functions
- No class Friday: SSA
  - No Broader Issue
  - Opportunities for extra credit

Alternative Approach to Development

- Create overview, define functions later
  - Top-down approach

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

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

- More later…

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

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


Pre/Post Conditions

```
# pre: binary_string is a string that contains only
# 0s and 1s
# post: returns the decimal value for the binary
# string
def binaryToDecimal( binary_string ):
    dec_value = 0
    for pos in xrange( 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
```


Review: Wheel of Fortune

- Practice: Modify displayed puzzle to handle punctuation
  - Include punctuation in displayed puzzle
  - Original code:

```
displayedPuzzle = ""
for char in puzzle:
    if char.isalpha():
        displayedPuzzle += "_"
    else:
        displayedPuzzle += char
    return displayedPuzzle
```


Program with `main()` and Functions

```
def main(): ← Program's driver goes at top
  print
  print "This program converts from binary to decimal numbers."
  print

  binary_string = raw_input("Enter a number in binary: ")

  while not isBinary(binary_string) :
      print "Sorry, that is not a binary string"
      binary_string = raw_input("Enter a number in binary: ")

  decValue = binaryToDecimal(binary_string)
  print "The decimal value is", decValue
```

Presents overview of what program does (hides details)