Objectives

- String review
- Introduction to Functions

Implementing Wheel of Fortune

- Simplifications: no money, no buying vowels, no keeping track of previous guesses, one player
- Functionality
  - Displaying puzzle appropriately
  - Gets guesses from user
    - Either letters or solve the puzzle
  - Keep track of the number of guesses
  - Displays puzzle with guesses filled in
- Think about …
  - User input robustness?
  - Any special cases?

wheeloffortune.py

Implementing Wheel of Fortune

- Differences between real and simulated game
  - Players type in letter rather than say it
    - Case matters
    - What if user enters more than one letter?

Implementing Wheel of Fortune

- User input verification
  - How can we ensure that the user entered only one letter?
  - How can we ensure that the user entered a *letter*?
- Checking the guess
  - How can we tell if the guessed letter is in the puzzle?
  - How can we report the number of times the guessed letter occurs in the puzzle?

Implementing Wheel of Fortune

- How many times should we prompt the user for a guess?
- How can we display the current puzzle?
  - What does the puzzle look like when we start the game?
  - What does it look like after we correctly guess a letter?

Wheel of Fortune

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

```
displayedpuzzle = ""
for char in PHRASE:
    if char != " ":
       displayedpuzzle += "_"
    else:
       displayedpuzzle += " "
```

puzzle

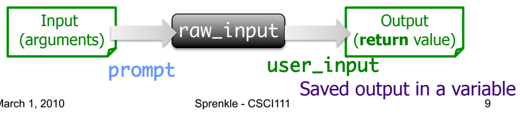
DEFINING FUNCTIONS

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**!

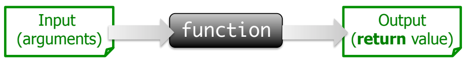
Review: Functions

- 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?

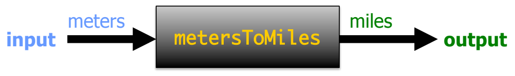
- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  -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

Similar to benefits of OO Programming

Example Program: Lab 2, Problem 4

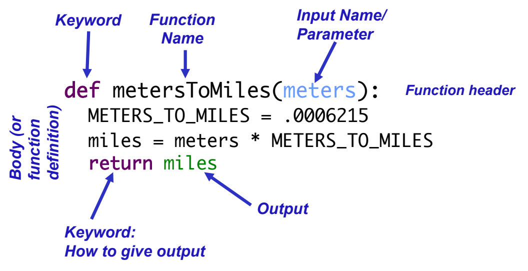
- Any place to make a function?
  - Duplicated code is often a "symptom" of when we should make a function
- Any place that has some useful code that we may want to reuse?

Convert meters to miles



- Input: meters
- Output: miles

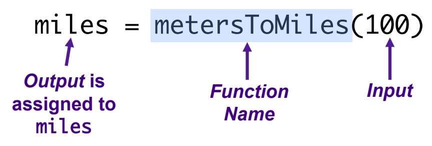
Syntax of Function Definition



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

Calling your own functions

Same as calling someone else's functions …



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

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***
- 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.

Formal *Actual*

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_example.py`

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
```

What does this function do?

Function definitions: Save functions for later use

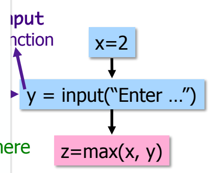
Program starts executing here

Flow of Control

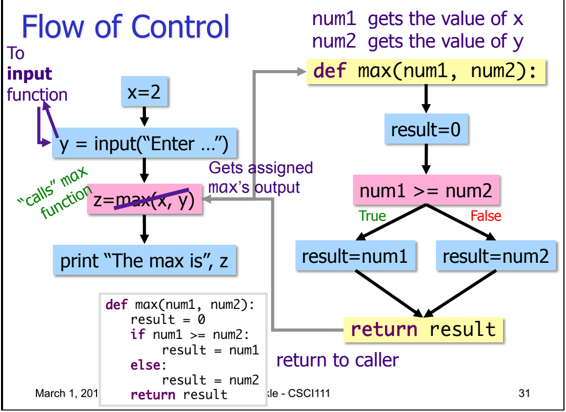
```
def max(num1, num2):
    result = 0
    if num1 >= num2:
      result = num1
    else:
      result = num2
    return result

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

Program starts executing here



Flow of Control

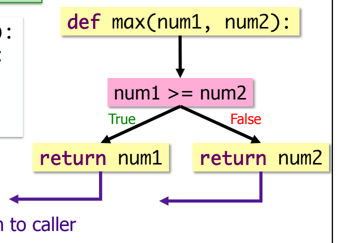


```
def max(num1, num2):
    result = 0
    if num1 >= num2:
        result = num1
    else:
        result = num2
    return result
```

Flow of Control: Using return

Is this implementation of the function correct?

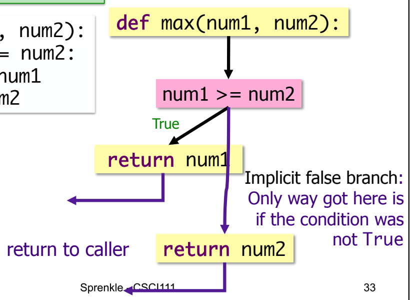
```
def max(num1, num2):
    if num1 >= num2:
        return num1
    else:
        return num2
```



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

- Identify 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

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
```

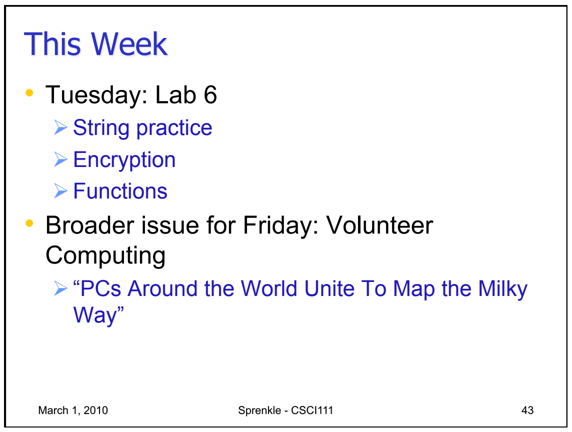
Presents overview of what program does (hides details)

Example program with a main()

- oldmac.py

Converting functionality into functions

- `binaryToDecimal.py`
  - Converting from binary to decimal
  - Checking if a string contains only binary numbers
- Write comments for the functions



This Week

- Tuesday: Lab 6
  - String practice
  - Encryption
  - Functions
- Broader issue for Friday: Volunteer Computing
  - “PCs Around the World Unite To Map the Milky Way”