

Objectives

- Wrap-up: Using **str** methods
- Introduction to Functions

Feb 25, 2008

Sprenkle - CS111

1

str Methods

- **str** is a class or a type
- **Methods**: available operations to perform on **str** objects
 - Used slightly differently than functions
 - Provide common functionality
- To see all the methods available for the **str** class
 - **help(str)**

Feb 25, 2008

Sprenkle - CS111

2

Common str Methods

Method	Operation
center(width)	Returns a copy of string centered within the given number of columns
count(sub[, start [, end]])	Return # of non-overlapping occurrences of substring sub in the string.
endswith(sub), startswith(sub)	Return True iff string ends with/starts with sub
find(sub[, start [, end]])	Return first index where substring sub is found
isalpha(), isdigit(), isspace()	Returns True iff string contains letters/digits/whitespace only
lower(), upper()	Return a copy of string converted to lowercase/lowercase

Feb 25, 2008

Sprenkle - CS111

string_methods.py

3

Common str Methods

Method	Operation
replace(old, new[, count])	Returns a copy of string with all occurrences of substring old replaced by substring new . If count given, only replaces first count instances.
split([sep])	Return a list of the words in the string, using sep as the delimiter string. If sep is not specified or is None, any whitespace string is a separator.
strip()	Return a copy of the string with the leading and trailing whitespace removed
join(<sequence>)	Return a string which is the concatenation of the strings in the sequence with the string this is called on as the separator
swapcase()	Return a copy of the string with uppercase characters converted to lowercase and vice versa.

Feb 25, 2008

Sprenkle - CS111

string_methods.py

4

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
 - Reports number of the guess in the puzzle
 - Displays puzzle with guesses filled in
- Think about ...
 - User input robustness?
 - Any special cases?

Feb 25, 2008

Sprenkle - CS111

wheeloffortune.py

5

Implementing Wheel of Fortune

- Differences between real and simulated game
 - Players say letter rather than type it in
 - Case matters
- Colin's suggestion to change the user's guess to uppercase -- OK
 - Emulates real game better
 - All uppercase letters in puzzle
- Keeping my **swapcase** solution
 - Allows user to have lowercase letters in original phrase

Feb 25, 2008

Sprenkle - CS111

6

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 += " "
```
- Practice: update puzzle with the user's guess
 - User's guess is named **guess**

Feb 25, 2008

Sprengle - CS111

7

Functions

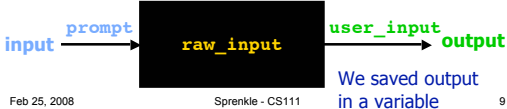
- We've used functions
 - Built-in functions: **len**, **input**, **raw_input**
 - Functions from modules, e.g., **math** and **random**
- Today, we'll learn how to define our own functions!

Feb 25, 2008

Sprengle - CS111

8

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)**
- 

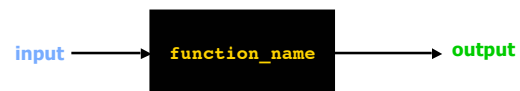
Feb 25, 2008

Sprengle - CS111

9

Functions

- In general, 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**



Feb 25, 2008

Sprengle - CS111

10

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 classes in OO Programming

Feb 25, 2008

Sprengle - CS111

11

Comparison of Code Using Functions

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

Feb 25, 2008

Sprengle - CS111

12

Example Program

- Lab 2, Problem 4
 - Any place to make a function?
 - Any place that has some useful code that we may want to reuse?

Feb 25, 2008

Sprengle - CS111

13

Convert meters to miles



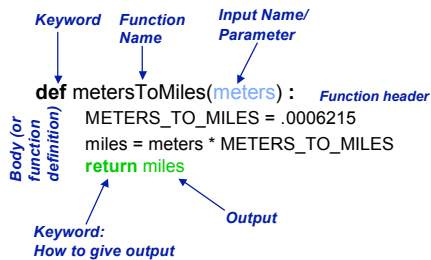
- **Input:** meters
- **Output:** miles

Feb 25, 2008

Sprengle - CS111

14

Syntax of Function Definition



Feb 25, 2008

Sprengle - CS111

15

Functions: Similarity to Math

- In math, 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

Feb 25, 2008

Sprengle - CS111

16

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 integer of decimal places to round to

Feb 25, 2008

Sprengle - CS111

17

Parameters

- **Formal Parameters** are the variables named in the 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):` **Formal**
 Use: `roundCelc = round(celc, 2)` **Actual**

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

Feb 25, 2008

Sprengle - CS111

18

Function Output

- When the code reaches a statement like **return x**
x is the **output returned** to the place where function called and the function stops
 - 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 (see `menu.py`)

Feb 25, 2008

Sprenkle - CS111

19

Calling your own functions

```
miles = metersToMiles(100)
miles2 = metersToMiles(200)
miles3 = metersToMiles(400)
miles4 = metersToMiles(800)
```

Output is assigned to miles4

Function Name

Input

Feb 25, 2008

Sprenkle - CS111

20

Flow of Control

- When you call the function, the computer jumps to the function and executes it
- When it is done executing the function, the computer returns to the same place in the first code where it left off

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

meters is assigned dist1, which is 100

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

Feb 25, 2008

Sprenkle - CS111

21

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

Feb 25, 2008

Sprenkle - CS111

22

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
```

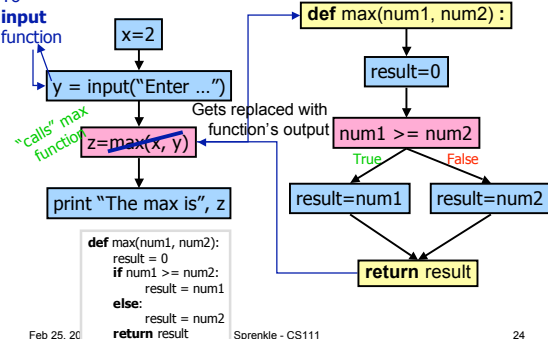
Program starts executing here

Feb 25, 2008

Sprenkle - CS111

23

Flow of Control



Feb 25, 2008

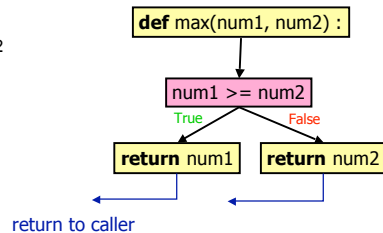
Sprenkle - CS111

24

Flow of Control: Using **return**

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

```
x=2
y=6
z = max( x, y )
```



Feb 25, 2008

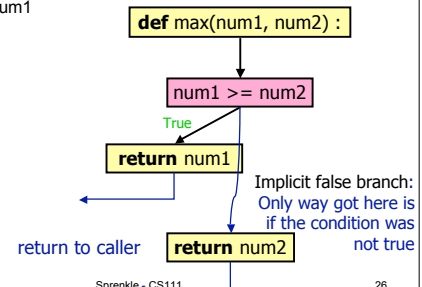
Sprenkle - CS111

25

Flow of Control: Using **return**

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

```
x=2
y=6
z = max( x, y )
```



Feb 25, 2008

Sprenkle - CS111

26

Passing Parameters

- Only **copies** of the actual parameters are given to the function
 - for **immutable** data types (which are what we've talked about so far)
- The actual parameters in the calling code do not change.
- Swap example:
 - Swap two values in script
 - Then, put into a function

Feb 25, 2008

Sprenkle - CS111

27

Where are Functions Defined?

- Functions can go inside of program script
 - Defined before use/called (if no **main()** function)
- Functions can go inside a separate **module**
 - Reduces code in main script
 - Easier to reuse by importing from a module
 - Maintains the "black box"
 - Isolates testing of function
 - Write "test driver" scripts to test functions separately from use in script

Feb 25, 2008

Sprenkle - CS111

menu.py

28

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 what program does
- main** usually takes no arguments
 - Example: `def main():`

Feb 25, 2008

Sprenkle - CS111

29

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

Feb 25, 2008

Sprenkle - CS111

30

Program With `main()` & Functions

```
def main():  
    print  
    print "This program converts binary numbers 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: ")  
  
    print "The decimal value is", binaryToDecimal(binary_string)
```

Presents overview of what program does (hides details)

Feb 25, 2008

Sprenkle - CS111

31

Example program with a `main()`

- oldmac.py

Feb 25, 2008

Sprenkle - CS111

32

Broader Issue Reading

- Two articles about Microsoft Excel 2007 Bug

Feb 25, 2008

Sprenkle - CS111

33