I. Announcements

Quiz 1 is coming up!
- **Regular exam**: Monday, Sept. 29 7:30pm - 9:00pm (34-101 for last names A-K, 32-123 for last names L-Z)
- **Conflict exam**: Thursday, Oct. 2 8am - 9:30am (34-301)

II. Functions

Why use functions?
- Functions help us decompose our code and make abstractions
  - **Decomposition**: breaks code into self-contained modules/reusable pieces
  - **Abstraction**: helps us suppress detail by turning useful chunks of code into black boxes that have a defined function and “just work”
- Makes code shorter and more readable
- Improve maintainability: can change functionality by only changing the function
- You have already been using functions
  - `len`, `raw_input`

How to write functions in python

```python
def function_name(formal_param1, formal_param2):
    """Specification. Include description of what function does, and any assumptions about parameters (e.g. type, acceptable values)""

    # Function body (indented)
    # Return statement
```

- **def**
  - stands for define
  - is a Python keyword, used only for defining functions
- **function name**
  - should be meaningful (like variable names)
  - used to invoke the function
- **formal parameters**
the parameters that are used within the function body

- can have default parameter values

- specification
  - is not required for your code to run, but is a very good practice!
  - state assumptions, which must be met by the client (users of the function), usually about parameter types and acceptable values
    - e.g. expects 1 argument of type string
  - state guarantees, which must be met by the function and guarantees that it runs in an expected way
    - e.g. returns integer representing length of string

- function body

- return statement
  - Emphasize: every function has a return type! Even built-in functions like print (returns None)
  - If the return value is not explicitly stated, then the default is return None
  - Ways to write return statements in Python (the last 3 are equivalent):
    - return <value>
    - return None
    - return
    - <nothing>
  - return vs. print
    - print works like a function that returns None (**according to Python documentation, it’s not normally available as a function since it’s recognized as a statement, but you can import it as a function to disable the statement**), can be used anywhere to output a value for the user to see
    - return only works when used within a function, will terminate the function

- function invocation
  - invoke the function by calling the function name, along with the actual parameters a.k.a. arguments
  - Can invoke with arguments out of order using keyword arguments

```
fuction_name(arg_1, arg_2)
OR
function_name(formal_param2 = arg_2, formal_param1 = arg_1)
```

Examples

1. `len`
   - assumes 1 argument of type string
   - return type: int

2. `raw_input`
• assumes 1 argument of type string
• return type: string

3. type
• assumes 1 input
• return type: type

What happens when a function is called
1. Actual parameters are evaluated, and then formal parameters are bound to the resulting values
2. Point of execution moves from point of invocation to the first line in the function body
3. Code in the body of the function executes until a return statement is hit or there are no more statements to execute
4. Value of the invocation becomes the returned value (None if there is no return statement)
5. Resume execution of code after the function invocation

Example
1. Compute your restaurant bill

```python
def add_tax(bill, tax_rate = 0.08):
    """Adds tax (8% by default) to a restaurant bill.""
    bill *= 1.0 + tax_rate
    print "With tax:", bill
    return bill

def add_tip(bill, tip_percent = 0.15):
    """Adds tip (15% by default) to a restaurant bill.""
    bill *= 1 + tip_percent
    print "With tip:", bill
    return bill

meal_cost = float(raw_input('Enter meal cost: '))
print "Base cost of meal: ", meal_cost
print "Computing tax..."
meal_with_tax = add_tax(meal_cost)
print "Computing tip..."
meal_with_tip = add_tip(meal_with_tax)
print "Done!"
```
Scoping

- Each function defines a new name space or scope
- Things defined within a scope (e.g. functions, variables) only exist in that scope

Example

```
1. def make_soup(veg, spice, broth):
   def mix_ingredients(veg, spice):
       print "Mixbot: you told me use veg = " + veg
       print "Mixbot: I hate " + veg
       veg = "carrots"
       print "Mixbot: I will make soup with veg = " + veg
       return spice + "ed " + veg

       print "Soupbot: you told me to use veg = " + veg
       print "Soupbot: I hate " + veg
       veg = "broccoli"
       print "Soupbot: I will tell Mixbot to use veg = " + veg
       spiced_veg = mix_ingredients(veg, spice)
       print "Soupbot: silly Mixbot, I wanted to use veg = " + veg
       return spiced_veg + " in " + broth + " broth"

   veg = "artichoke"
   spice = "pepper"
   broth = "chicken"

   print "Me: I want to make soup with veg = " + veg
   print "Me: We made " + make_soup(veg, spice, broth)
   print "Me: Silly bots, I wanted to make soup with veg = " + veg
   print mix_ingredients(veg, spice) # Why is this undefined?
```
III. Recursion

What is recursion?
- The technique of breaking a problem down into smaller and smaller problems until you can solve it trivially
- A recursive definition has 2 parts
  - at least one base case that directly defines the result for a special case
  - at least one recursive (inductive) case that defines the answer in terms of the question using a different input, which is usually a simpler problem
- In programming, a recursive function is a function that calls itself (the recursive case)

- You can write any recursive function as a for loop (but it can look a lot more complicated)
- You can write any for loop as a recursive function (but you usually don’t want to)
- Recursion may be slower in many languages, because there is overhead associated with calling many functions repeatedly
- So, why do we use recursion if we could just use a loop?
  - Looks simpler and more readable for certain problems
  - Easier to understand and maintain code

Easy examples
1. Fibonacci (covered in textbook)

```python
def fibonacci(x):
    """Assumes n an int >= 0""
    if x == 0 or x == 1:
        return 1
    return fibonacci(x - 1) + fibonacci(x - 2)
```

2. Factorial (covered in lecture)

```python
def factorial(x):
    if x == 0:
        return 1
    return x * factorial(x - 1)
```

3. Sum of numbers from 1 to n (very similar to factorial)

```python
# recursive solution
def sum_to_n(n):
    if n==0:
        return 0
```
else:
    return sum_to_n(n-1) + n

# iterative solution
def sum_to_n(n):
    total = 0
    for i in range(n+1): # can ask about why we need n+1 instead of n
        total += i
    return total

Harder examples
1. String reversal (non-numerical example)

def reverse_word(word):
    if word == "":
        return word
    else:
        return reverse_word(word[1:len(word)]) + word[0:1]

print reverse_word("goldfish")

2. Find max in list (similar to search algorithms)

def find_max(L):
    if len(L) <= 1:
        return L[0]
    else:
        possible_max_1 = L[0]
        possible_max_2 = find_max(L[1:len(L)]);

        if possible_max_1 > possible_max_2:
            return possible_max_1
        else:
            return possible_max_2

print find_max([5, 2, 3, 10, 5, 12])

3. Remove all vowels in word (multiple base cases)

def remove_vowels(word):
if len(word)==1:
    if word[0] == "a" or word[0] == "e" or word[0] == "i" or word[0] == "o" or word[0] == "u":
        return ""
    else:
        return word[0]
else:
    return remove_vowels(word[0]) + remove_vowels(word[1:])

print remove_vowels("goldfish")