Chapter 7: Functions

CSCI-UA 0002 – Introduction to Computer Programming
Mr. Joel Kemp
Agenda

- What are Functions?
- Function basics
- Returning Data
- Recursion
- Variable Scope
- Pass by reference
- Global Variables
- Modules
Functions

• What is a function?
  – A group of code that can be called by name and solves a particular task.

• Example:

```python
from math import sqrt
result = sqrt(9)  # sqrt takes an input and provides an output
```

• Why do we use them?
  – Code reuse
    • Without `sqrt()`, we’d have to write the code to compute the square root ourselves.
    • We’d also have to copy and paste it every time we wanted to compute the square root.
  – Cleaner code
    • We can break a complex problem into sub-problems
      – Each function could solve a sub-problem
Our first function!

• Functions in Python are defined using the `def` keyword.

• Example:
  ```python
def squared(num):
    result = num * num
    print(result)
  ```

• Note:
  – The `squared()` function takes a single `parameter` and prints the answer.
    • **Parameter**: a variable name that represents an input to a function.
  – A colon is used to prepare the interpreter for a body of code.
    • Similar to a branch and loop!
    • Indentation is also used to indicate the function body
Calling Functions

• We can call our custom functions as we have been doing:
  – As stand-alone statements
    ```python
    squared(2)
squared(3)
    ```
  – Within branches
    ```python
    if(x > 0):
        squared(x)
    ```
  – Within loops
    ```python
    while(True):
        squared(x)
    # Assume this breaks
    ```
But wait...

• The `squared()` function took in a value and printed the result to the console.
  – What if I don’t want to print yet?
  – What if I want to use the result in another computation?

• Returning Data:
  – You can return data from a function using the `return` keyword.
    
    ```python
    def squared(num):
        result = num * num
        return result
    ```
    
    – You can now use a variable to catch the returned value.
    ```python
    num_squared = squared(3)  # num_squared now holds 9
    ```
  
  • The evaluation of the function call involves passing the value 3 into the body of `squared()`.
  • `squared()` takes 3 and assigns it to the local variable, `num`.
    – The result, 9, is then returned to wherever `squared()` was invoked.
  • `num_squared`
How does it work?

• Function definition:
  
  ```python
  def squared(num):
      result = num * num
      return result
  ```

• Function call:
  
  ```python
  num_squared = squared(3)  #num_squared = 9
  ```
  
  – The evaluation of the function call involves passing the value 3 into the body of `squared()`.
  
  – `squared()` takes 3 and assigns it to the local variable, `num`.
    
    • The result, 9, is then returned to wherever `squared()` was invoked.
  
  – The variable, `num_squared`, is set to the result of the evaluation of `squared(3)`, which is 9.
Recursion

• What is it?
  – A method of nesting function calls where a function calls itself within its own definition!

• Example:
  ```python
  def factorial(num):
      if(num == 0):
          return 1
      else:
          return num * factorial(num - 1)
  ```

• Recursive programs are elegant, but decrease readability, use more resources, and can be solved iteratively.
  – Still, that’s a sexy function! 😊
Recursion Diagram

def factorial(num):
    if(num == 0):
        return 1
    else:
        return num * factorial(num - 1)

factorial(5) = 5 * factorial(4)
    -> 4 * factorial(3)
        -> 3 * factorial(2)
            -> 2 * factorial(1)
                -> 1 * factorial(0)

Refer to: factorial.cpp
Exercises

1. Write a function `my_sum(N)` that accepts a number, N, and recursively calculates the sum of numbers from 1 to N.
   - Examples:
     ```python
     my_sum(4)  # returns 10 (4+3+2+1)
     my_sum(5)  # returns 15 (5+4+3+2+1)
     ```

2. Write a function `harmonic_sum(N)` that accepts a number, N, and recursively calculates the sum `1 + 1/2 + 1/3 + ... + 1/n`

Refer to: recursiveExercises.py
Scope

• The *scope* of a variable is the section of code that has access to that variable.

```python
def squared(num):
    result = num * num
    return result
```

– Both variables `num` and `result` are local to the `squared()` function.

• Think of functions as self-contained, mini programs.
  – Functions should use their own variables and not rely on other parts of the program.
    • Variables declared within a function are called *local variables*.
    • Function parameters are also considered local variables.
    • Local variables cannot be accessed (i.e., do not exist) outside of the function.

Refer to: scope.py
Pass by Reference

• Let’s say that a function takes in a list (or hash)...
  – If you modify the list by adding or removing data, then that list gets permanently modified!
    • i.e., the changes exist outside that function.
  – Methods called on that reference permanently modify the passed data.

• However, assigning a new list to the local variable doesn’t affect the original list.

Refer to:
  reference.py
Exercises

1. Write a function, `average()`, that accepts a list of numbers, `numbers_list`, and returns the average of the numbers in the list.
   
   Example:
   
   ```python
   average([1,2,3])  # returns 2.0
   ```

2. Write a function, `euclidean_instance()`, that accepts 4 numbers `x1, y1, x2, y2` and returns the Euclidean distance of those points.

   \[ dist = \sqrt{(x2 - x1)^2 + (y2 - y1)^2} \]

Refer to: functionExercises.py