CS 157 - Introduction to Programming and Modeling

Function Defintions

Dr. Stephen P. Carl

Quote of the Day

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Fowler

Abstraction and Decomposition

When solving large problems, we use the concepts of **abstraction** and **decomposition.**

Abstraction is used in many different ways. In the big picture, abstraction is the stripping away of irrelevant detail to get at the **core of the problem.**

For our purposes, it can also mean **naming a pattern.** Once we identify a pattern in code and give it a name, we can just use the name from then on.

Abstraction and Decomposition

Decomposition is the process of breaking down large, difficult problems into smaller problems that are easier to understand, and therefore to solve.

Our goal here is to find ways to break large coding problems into smaller pieces called *functions.*

Functions Define Operations

Like the built-in functions in Python, we can define and call our own functions to perform any operation we require.

Functions should be designed to be:

- short and easily understandable
- targeted to a specific task
- generalized, by being applicatable to a wide range of values

SEWANEE
The University of the South

Defining New Functions

A new function definition is introduced by the keyword **`def`** which defines the name of the new function. The definition is made up of the *header* and a *body.*

The function header has the following format:

def functionName(parameters):

where *parameters* is a parenthesized list containing zero or more parameter names.

SEWANEE
The University of the South

Function Header Examples

- A function with an empty parameter list

  def begin():

- A function with a single parameter

  def onMousePress(pressPoint):

- A function that takes two parameters to compute a number

  def distance(point1, point2):

SEWANEE
The University of the South

Parameter Lists

The *parameter list* is used to pass data to a function needed to perform its computation. As we've seen, a function can have zero, one, or more parameters.

Each name inside the parentheses is essentially a new variable name; the actual arguments used when the function is *called* are implicitly assigned to these variables.

The parameter names are separated by commas when there is more than one. A common error is to leave off the parens **`()`** when no parameters exist.

SEWANEE
The University of the South

Function Body

The function *body* contains the executable statements needed to do the intended computation. Statements are executed in order starting with the first statement after the function header.

Any kind of statement is valid in the function body, including those which alter flow of control, such as selection, iteration, or calls to other functions.

SEWANEE
The University of the South

Design Rules for Functions

- Every function will have a *signature comment*
- Every function will have a short description called a *docstring*
- Every function will have a *definition*
- Every function will have *examples* in the form of runnable *test cases*

SEWANEE
The University of the South

The Signature Comment

Start with a comment showing the *types* of the parameters and return value, if any. I use two #'s to make this stand out. For example:

```
## squaresum : list-of-int -> int
```

This says the function **`squaresum`** takes a list of integers and returns an integer (presumably the sum of their squares)

SEWANEE
The University of the South

Purpose and Definition

Next comes the implementation. After the function header, we add a *docstring,* a brief but complete description of the computation the function is supposed to do, followed by the code which implements it.

```
def squaresum(nums):
    """ computes the sum of squares of each value in nums """
    sum = 0
    for x in nums:
        sum = sum + x * x
    return sum
```

SEWANEE
The University of the South

Docstrings

By the way, docstrings allow multi-line comments and can also be used for the file header comment:

```
"""
sumsquares.py - functions on squared numbers in a list
       author - spc
"""
```

SEWANEE The University of the South

Test Cases

Test cases let us check the function's correctness right away. Write tests to exercise as many different cases as you can think of, remembering to consider *boundary conditions.* For example:

```
# use the pytest tool to run tests
def test_squaresum():
    assert squaresum([]) == 0
    assert squaresum([1, 2, 3]) == 14
```

By doing this, we ensure that we *really* understand what this function is supposed to do.

SEWANEE The University of the South

Support for Testing

The pytest tool supports testing by running any *test functions* it finds automatically. All test functions start with the **test_** prefix.

The **`assert`** statement *requires* the boolean expression which follows it to be **True.** Pytest runs each test function and flags any assertion that fails:

```
pytest squares.py
========== 2 passed in 0.03 seconds ==========
```

SEWANEE The University of the South

Support for Testing

Or, should there be an error in the function **`sumsquares`** you might see something like this:

```
pytest squares.py
========== FAILURES ==========
_______test_squaresum_______
  def test_squaresum():
>     assert squaresum([1, 2, 3]) == 14
E       assert 6 == 14
E        +  where 6 = squaresum([1, 2, 3])
```

SEWANEE The University of the South