CPSC 301: Computing in the Life Sciences  
Lecture Notes 5:  
Python Functions

Pre-class slides

Jessica Dawson  
jq Dawson@cs.ubc.ca  
http://www.cs.ubc.ca/~jq Dawson

University of British Columbia  
Department of Computer Science

2016 W2
At the end of this unit, you will be able to:

- call a previously defined Python function
- define Python functions to perform certain tasks
- describe how a function is executed
- identify local variables and function parameters
- trace a function call and identify what value the function returns
- sketch out the state of memory before, during and after a function call
- design a function according to the suggested recipe
Functions

• A large program (procedure) for a task is usually broken down to subprograms each of which computes a subtask
  – easier to read understand and change the program
  – subtasks can be used in other programs
• A subtask is like a smaller task. It has its input values, its procedure and its results (output)
• In Python we use functions to define simple tasks which can be the subtasks of a problem
• That is, a complex program is split into a number of functions, each computing a subtask of the problem
**Function Definition**

- A function definition looks like:
  ```python
  def function-name( parameters ) :
      """ docstring for the function """
      block-of-statements
  
  where:
  
  - **def** is a Python keyword starting a function definition
  - **function-name** is the name we give to the function, like a variable name
  - **parameters** are variables that will provide the input values for the function
  - A **docstring** is a (usually multiline) string that appears immediately after the header line ("**def ...**") of a function
  - the block-of-statements defines the tasks for the function; all these statements are indented by a tab stroke

- A return statement of the form
  ```python
  return some-value
  ```
  is used in the function statements to return the result
• Example 1: a function that converts miles to kilometres

```python
def kilometers(miles):
    """Returns the kilometers that correspond to the given miles """
    return miles * 1.609
```

• To use the function we use its name and provide values for its parameters (called *arguments*).

• Example of function calls:
  – kilometers(100) will return 160.9
  – distance_in_miles = 10
    
    ```
    distance_in_kilometers = kilometers(distance_in_miles)
    then distance_in_kilometers  16.09
    ```

• Example 2: A function that calculates the income tax given a income and a tax rate:

```python
def income_tax( income, rate ) :
    """Returns the income tax for the given income and tax rate"""
    return (income - 9000.00) * rate
```

• Example call: income_tax( 50000, 0.30 )
Local Variables

• A function may need to declare its own variables to simplify the calculations. These variables
  – are only accessible from the statements inside the function's body
  – are called *local variables*

• Example: An income_tax function:
  ```python
def income_tax( income, rate ) :
    """Returns the income tax for the given income and tax rate""
    exemption = 9000.0
    net_income = income - exemption
    return net_income * rate
  ```

• Note: parameters are only accessible inside the function body as well; they are local variables as well
  – in the above example: income, rate, exemption and net_income are only known inside the income_tax function
  – we say that the *scope* of these variables is the code from their declaration to the end of the function
Function Call

• Each function call is executed in its own name space or frame: an environment with its local variables and parameters.
• When a function returns its environment is destroyed and the value is returned to the environment that called the function.
• Example:

```python
def income_tax(income, rate):
    exemption = 9000.0
    net_income = income - exemption
    return net_income * rate
```

# Main Program

```python
my_income = 10000.0
my_rate = 0.2
my_tax = income_tax(my_income, my_rate)
```

income_tax(...) call's frame:
- income → 10000.0
- rate → 0.2
- exemption → 9000.0
- net_income → 1000.0

Main's frame:
- my_income → 10000.0
- my_rate → 0.20
- my_tax → __________
The return statement

• Determines flow of control
  – When return is encountered in a function, the function call is done and control is returned to the caller
  – Any further statements in the function are ignored
• A function call is an expression which always has a value
  – That value can be specified in the return statement which completes the function call
  – If no value is specified in the return statement, the value of the function call is None
  – If no return statement is encountered before the body of the function ends, the value of the function call is None
Docstrings

- Let's now provide the right documentation (docstring) for the income_tax function:

```python
def income_tax( income, rate ) :
    """ (float, float) -> float
    income >= 9000.0;  0.0 <= rate < 1.0
    Returns the income tax for the given income and tax rate.
    income >= 0 and rate >= 0; rate is in decimal form.
    i.e. 0.3 means 30%
    >>> income_tax(9000.0,  0.2)
    0.0
    >>> income_tax(9000.0,  0.0)
    0.0
    >>> income_tax(10000.0, 0.3
    300.0
    """

    exemption = 9000.00
    net_income = income - exemption
    return  net_income * rate
```

- The docstring is mainly for human users. We normally write this BEFORE we write the code for the function.
- The command `help(income_tax)` displays the docstring.
- The test code inside the docstring is not part of the function and will not be executed when the function is called.
Function Design Recipe

1. Determine the type contract
   — What are the types of the input and output data?
   — Are there any preconditions on the input values?
2. Write the header
   — Pick meaningful names for the function and parameters
3. Write the description
   — Make sure to include all your parameters in your description
4. Write some examples
   — Pick examples which are as distinct as possible
5. Write the body
   — Pick meaningful names for the local variables
6. Test your function with your examples
   — Better to check now whether your code has a bug
   — Add more examples if you think of some
Example: Room Painting Cost, Version 3

# Functions

def wall_area(length, width, height):
    """
    (float, float, float) -> float
    length, width, height are >= 0
    
    Calculates the wall area of a room with the given length, width and height.
    """

    >>> wall_area(0, 0, 0)
    0.0
    >>> wall_area(10, 10, 10)
    400.0
    ""
    wall_1 = length * height
    wall_2 = width * height
    return 2 * wall_1 + 2 * wall_2

def paint_cost(area, paint_price):
    """
    (float, float) -> float
    area and paint_price are >= 0
    
    Calculates the cost for the paint needed to paint the given area. 
paint_price is the price of one gallon of paint.
    """

    >>> paint_cost(0, 0)
    0.0
    >>> paint_cost(50, 10)
    10.0
    >>> paint_cost(200, 10)
    40.0
    ""
    return area / 50 * paint_price

def labour_cost(area, labour_price):
    """
    (float, float) -> float
    area and labour_price are >= 0
    
    Calculates the labour cost for painting the given area 
labour_price is the labour cost per hour.
    """

    >>> labour_cost(0, 0)
    0.0
    >>> labour_cost(50, 10)
    10.0
    >>> labour_cost(200, 10)
    40.0
    ""
    return area / 50 * labour_price
# Main program

# Paint price per gallon
paint_price = 25

# Labour cost per hour
labour_price = 20

# Get room dimensions and estimate the cost.
room_length = float(input("Enter room's length : "))
room_width = float(input("Enter room's width: "))
room_height = float(input("Enter room's height: "))

# Total area for painting
total_area = wall_area(room_length, room_width, room_height)
print("Total area is ": total_area)

# Calculate the cost of paint needed
cost_for_paint = paint_cost(total_area, paint_price)
print("Paint cost is ": cost_for_paint)

# Calculate labour cost
cost_for_labour = labour_cost(total_area, labour_price)
print("Labour cost is ": cost_for_labour)

# Calculate total cost
total_cost = cost_for_paint + cost_for_labour
print("Total cost is ": total_cost)
print()

print("Bye now. Hope to hear from you soon")

return area / 50 * paint_price
Built-in Functions

Python comes with a number of useful functions already defined and available to use like:

• General math functions
  – \texttt{abs}(number) – absolute value of a number
  • \texttt{abs}(5) is 5, \texttt{abs}(-5) is 5
  – \texttt{pow}(n1, n2) - n1 to the power of n2
  • \texttt{pow}(2, 3) is 8
  – \texttt{round}(number) - rounds a floating-point number
    • \texttt{round}(2.4999) is 2.0, \texttt{round}(2.5) is 3.0, \texttt{round}(2) is 2.0

• Functions that convert a value of one type to another type:
  – \texttt{int}(number) returns an integer
    • \texttt{int}(2.9999) is 2, \texttt{int}(2.0001) is 2
  – \texttt{float}(number) returns a floating-point number
    • \texttt{float}(2) is 2.0, \texttt{float}(2.33455) is 2.33455
  – \texttt{str}(value) returns a string
    • \texttt{str}(2.25) is '2.25', \texttt{str}(123) is '123', \texttt{str('abc')} is 'abc'

Notes 5: Functions
Default Parameters & Keyword Arguments

- You may provide default values for function parameters in the header line:

  ```python
def paint_cost(length, width, height = 8,
                 paint_price = 45, labour_price = 25):
  ...
```

  - Allows you to design flexible functions with simple interfaces
  - Default is evaluated only once (which will matter for mutable data)
  - Parameters with defaults must appear at the end of the parameter list
  - Having default parameter values created optional arguments

- You may provide function arguments out of order using keyword arguments:

  ```python
  print('total:', paint_cost(10, 12, paint_price = 49.99))
  ```

  - Keyword arguments must appear at the end of the argument list
  - You may provide keyword arguments for any parameter (whether or not it has a default value)
  - Each parameter must be given one value
Recap of Key Terminology

- **Function call**: tells python to execute a function
- **Parameters**: the variable(s) that appear in the parentheses in of the function header
- **Arguments**: the expression (values, variable, function) that you pass to the function when you make a function call
  - These map to the parameters in the function definition
- **Local variable**: used in the function definition for temporary storage. Created when a function is called, and erased when the function returns.
Conclusion

- A large task can be split into subtasks using functions
- A new function is created by the command `def` – The function body is the subsequent indented code
- To document a new function, use docstrings and the function design recipe
- Each function can have parameters and local variables that are only accessible inside the function's code
- To execute a function we write a function call that is made of – the function name – a value for each parameter
- When a function is called, the function code is executed in its own environment (frame)
- When the function return its environment is destroyed
- Python comes with a large number of built-in functions (some of which we'll see later)