Handout 5

Defining functions.

A **function** is a collection of statements that are grouped together to perform an operation.
Advantages from defining and using functions:

- improve the quality of the program by modularizing the code
- reduce redundant coding and enable code reuse
- enable division of labor

A function definition consists of a **header** and a **body**.

The header begins with the **def** keyword, followed by function's name and parameters, followed by a colon.

The body consists of the indented code underneath the header.

The function definition does not execute the function body; function gets executed only when the function is called.

```
# The following is a function definition
# This function has no parameters and no return value
def printHelloWorld():
    print ('Hello, world')

# Call the above function three times
printHelloWorld()
printHelloWorld()
printHelloWorld()
```

The variables defined in the function header are known as *formal parameters*.
When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*:

```
#-------------------------------------------
# Function takes a single parameter; returns no value
def printHello (name):
    print ('Hello,' , name , '!')

printHello('Tamara')
aname = input('Greetings! what is your name?')
printHello(aname)
```

A function may return a value using the **return** keyword.
A function that does not return a value eturns a special value, **None** (as the two functions above)

```
#-------------------------------------------------------
# Function takes a single parameter, returns a single value

def helloString (name):
    result = 'Hello,' + name + '!'
    return result

greetingString =  helloString('Joe')
```

Practice problem:

- identify function calls vs function definition
- identify function parameters, return value

```
def toMin (strTimeHM):
    '''
    Parameters:
    strTimeHM - a string of format 'HH:MM', where HH is hour, MM is minute.
    Returns a single integer corresponding to the time specified by
            the parameter, converted to minutes.
    '''
    h , m = strTimeHM.split(':')
    print (' h = ', h, 'm = ', m)
    minutes = int(h)*60 + int(m)
    return minutes


min = toMin('15:00')
print (min)

print (toMin('6:22'))
print (toMin('6:00'))

timeEntry = input('Please enter time, e.g. 7:23 or 11:05')
print (toMin(timeEntry))
```

DEFINITIONS WITH DEFAULT PARAMETER VALUES

When one or more top-level parameters have the form *parameter* = *expression*, the function is said to have ``default parameter values."
For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted.
If a parameter has a default value, all following parameters must also have a default value.

```
def toMin (strTimeHM, separator = ':', seconds = False ):
    if seconds:
        h , m, s = strTimeHM.split(separator)
    else:
        h , m  = strTimeHM.split(separator); s = 0

    print (' h = ', h, 'm = ', m, 's = ', s)

    minutes = int(h)*60 + int(m) + int(s)/60
    return minutes

print (toMin('6:22'))
print (toMin('12.00', '.'))
print (toMin('12.00.10', '.', True))

# following will generate a syntax error
#print (toMin('12:00:10', , True))

print (toMin('12:00:10', seconds =  True))

# When parameter name is specified, can use a different order
print (toMin('12**00**10', seconds =  True, separator = '**'))
#   and even
print (toMin( seconds =  True, separator = '**', strTimeHM = '12**00**10'))
```

RETURNING MULTIPLE VALUES (AS A TUPLE)

Following example, shows function return three values h,m,s.

```
def parseHourMinSec (strTimeHM, separator = ':', seconds = False ):
    ''' Function parses a string definition of time, e.g. 13:34:06
        or 12.33 into an hour, minute and second values
        Params:
        strTimeHM   - a string with time values separated with the separator
        separator - a string that is used to separate components
        seconds - Bool, indicating whether seconds is included in strTimeHM
        Returns:
      Separated time components:
              hour minute and second, as integer numbers
            when seconds were not specified in the strTimeHM –
               returns seconds value of 0
    '''
    print ('parsing ', strTimeHM, end = " --> ")
    if seconds:
        h, m, s = strTimeHM.split(separator)
    else:
        h , m  = strTimeHM.split(separator); s = 0

    print (' h = ', h, 'm = ', m, 's = ', s)

    return h, m, s;
```

```
print (parseHourMinSec('6:22'))
print (parseHourMinSec('12.00.10', '.', True))

hour, min, sec = parseHourMinSec('12:33:45', seconds = True)
print (' hour = ', hour, 'min = ', min, 'sec = ', sec)
```

Python allows a function to have a return statement with multiple values. In that case, the values are returned as a *tuple*, which is a data type for an immutable collection
Parentheses, which group elements, around tuples are optional, so 1, 2, 3 is equivalent to (1,2,3), and hour, min, sec is equivalent to (hour, min, sec)

SCOPE OF VARIABLES

Scope: the part of the program where the variable can be referenced.

A variable created inside a function is referred to as a *local variable*. Local variables can only be accessed inside a function. The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.

In Python, you can also use *global variables*. They are created **outside all functions** and are accessible **to all functions** in their scope.

1.

```
x = 1

def f1():
    x = 2
    print(x) # Displays 2

f1()
print(x) # Displays 1
```

2.

```
x = 1

def increase():
    global x
    x =  x + 1
    print(x) # Displays 2

increase()
print(x) # Displays 2
```

3. Identify global variables in the code listings on previous pages.

The use of global variables should be minimized, to include only global constants, which should be declared in all capital letters.
Global variables make it difficult to trace code and debug the program, since they can be changed by any part of the code. Local variables can only be changed within the methods, in which they participate.

Here's the design that places all code in functions: notice there is a single function call to main(), everything else is defined via and within functions. The name main() has **no special significance** in Python (differently from C, Java).

```
def parseHourMinSec (strTimeHM, separator = ':', seconds = False ):
    print ('parsing ', strTimeHM, end = " --> ")
    if seconds:
        h, m, s = strTimeHM.split(separator)
    else:
        h , m  = strTimeHM.split(separator); s = 0

    return h, m, s;

def main():
    print (parseHourMinSec('6:22'))
    print (parseHourMinSec('12.00.10', '.', True))

    hour, min, sec = parseHourMinSec('12:33:45', seconds = True)
    print (' hour = ', hour, 'min = ', min, 'sec = ', sec)


main()
```

PARAMETER PASSING MECHANISM - PASS BY VALUE

In Python, all data values are objects. A variable for an object stores a reference to the object. When you invoke a function with a parameter, the reference value of the argument is passed to the parameter. This is referred to as *pass-by-value*. For simplicity, we say that the value of an argument is passed to a parameter when invoking a function. Precisely, the value is actually a reference value to the object.

If the object passed to a parameter is changed inside the function, the changes will persist when the function is done executing.

Practice: What will be printed by the following programs?

1.

```
# Append the length of the list to the end of the list.
def appendLength(words):
    words.append(len(words))

def main():
    lst = 'Veni,vidi,vici'.split(',')
    print('before call, lst = ', lst)
    appendLength(lst)
    print('after call, lst = ', lst)

main()
```

2.

```
# Replace punctuation in the line with space
def replacePunctuation(line):
    for ch in line:
        if ch in '"~@#$%^&*()+-,.:;/-!?[]{}<>':
            line = line.replace(ch, "")

    return line

def main():
    text = 'Veni, vidi, vici.'
    print('text = ', text)
    newtext = replacePunctuation(text)
    print ('newtext = ',  newtext)
    print('text = ', text)

main()
```

OTHER NOTABLE FACTS ABOUT PYTHON FUNCTIONS

- Functions are first-class objects and can be passed in as parameters (look, for example at the `map` and `filter` functions descriptions)
- You can create anonymous functions using `lambda`-forms
- Functions can accept a variable number of parameters (check out *unpacking arguments*, `*args` and `**kwargs`)

PRACTICE PROBLEMS ON FUNCTION DEFINITIONS

1. Define and test a function which is passed a year value and returns true or false depending on whether the year is a leap year. Leap years are all those year which are divisible by 4, except for centuries, not divisible by 400.

2. Compose a function **`readmatrix()`** that will read values for a matrix from the user and return the numbers stored in a two-dimensional list format. The function should have parameters defining the dimensions (number of rows and columns) of the matrix. When those parameters are not passed, the function should get those values from the user. Sample interaction demonstrates the execution of the function, when no parameters are passed:

```
Please enter the number of rows and number of columns, e.g. 5 6: 3 2
Enter row 1 numbers separated by a space 3 4
Enter row 2 numbers separated by a space 67 78 89
--each row must have 2 values. Skipping this entry..
Enter row 2 numbers separated by a space 67 78
Enter row 3 numbers separated by a space 56 3
```

Given this interaction, the method should return the following list:

```
[[3, 4], [67, 78], [56, 3]]
```

3. Compose a function, called **`findincreasingrows()`** that would be passed matrix of integers represented as a two-dimensional list, and would return a list of row numbers, which includes those rows that contain numbers in strictly increasing order. For example, given the following input,

   ```
   [[3, 4, 7], [67,-8, 78], [2, 56, 3], [45, 56, 67]]
   ```

   the return value should be [0, 3].