Functions

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if you find yourself writing the same code at several different pits in your program. You can put that code in a function and call the function whenever you want to execute that code.

Basics: functions are defined with the ***def*** statement. The statement ends with a colon, and the code that is part of the function is indented below ***def*** statement. Here we create a simple function that just prints something.

```
def print_hello():
    print('Hello!')

print_hello()
print('1234567')
print_hello()
```

The first two lines define the function. In the last three lines we call the function twice.

One use for function is if you are using the same code over and over again in various parts of your program, you can make program shorter and easier to understand by putting the code in a function. For instance, suppose for some reason you need to print a box as several points in your program. Put the code into a function, and then whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def draw_square():
    print('*' * 15)
    print('*', ' '*11, '*')
    print('*', ' '*11, '*')
    print('*' * 15)
```

One benefit of this is that if you decide to change the size of the box, you just have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

Arguments: we can pass values to functions. Here is an example:

```
def print_hello(n):
    print('Hello ' * n)
    print()

print_hello(3)
print_hello(5)
times = 2
print_hello(times)
```

When we call the print_hello function with the value 3, that value gets stored in the variable n. we can then refer to that variable n in our function's code. You can pass more than one value to a function:

```python
def multiple_print(string, n):
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('A', 10)
```

Returning values: we can write function that perform calculations and return a result. Here is a simple function that converts temperatures from Celsius to Fahrenheit.

```python
def convert(t):
    return t*9/5+32

print(convert(20))
```

The ***return*** statement is used to sent the result of a function's calculations back to the caller. Notice that the function itself does not do any printing. The printing is done outside the function. That way, we can do math with the result, like below.

```python
print(convert(20)+5)
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would never be able to do anything with it.

A function can return multiple values as a ***list***. Say we want to write a function that solve the system of equations x = (de-bf)/(ad-bc) and y = (af-ce)/(ad-bc). We need our function to return both the x and y solution.

```
def solve(a,b,c,d,e,f):
    x = (d*e-b*f)/(a*d-b*c)
    y = (a*f-c*e)/(a*d-b*c)
    return [x,y]
```

```
xsol, ysol = solve(2,3,4,1,2,5)
print('The solution is x = ', xsol, 'and y = ', ysol)
```

This method uses the shortcut for assigning to lists.

A ***return*** statement by itself can be used to end a function early.

```
def multiple_print(string, n, bad_words):
    if string in bad_words:
        return
    print(string * n)
    print()
```

The same effect can be achieved with an if/else statement, but in some cases, using return can make your code simpler and more readable.

Default arguments and keyword arguments: you can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value. Here is an example:

```
def multiple_print(string, n=1)
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('Hello')
```

default arguments need to come at the end of the function definition, after all of the non-default arguments.

Keyword arguments: a related concept to default arguments is keyword arguments. Say we have the following function definition:

```
def fancy_print(text, color, background, style, justify):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, python allows you to name the arguments when calling the function, as shown below:

```
fancy_print(text='Hi', color='yellow', background='black',
            style='bold', justify='left')
```

```
fancy_print(text='Hi', style='bold', justify='left',
            background='black', color='yellow')
```

As we can see, the order of the arguments does not matter when you use keyword arguments. When defining the function, it would be a good idea to give defaults. For instance, most of the time, caller would want left justification, a white background, etc. using these values as defaults means the caller does not have to specify every single argument every time, they call the function. Here is an example.

```
def fancy_print(text, color='black', background='white',
                style='normal', justify='left'):
    # function code goes here

fancy_print('Hi', style='bold')
fancy_print('Hi', color='yellow', background='black')
fancy_print('Hi')
```

Local variables: let's say we have two functions like the ones below that each use a variable i:

```
def func1():
    for i in range(10):
        print(i)

def func2():
    i=100
    func1()
    print(i)
```

A problem that could arise here is that when we call ***func1***, we might mess up the value of I in ***funct2***. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions, and, fortunately, we don't have to worry about this. When a variable is defined inside a function, it is ***local*** to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

Global variables: on the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a ***global*** variable. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller program. Here is a short example:

```
def reset():
    global time_left
    time_left = 0

def print_time():
    print(time_left)

time_left=30
```

In this program we have a variable time_left that we would like multiple function to have access to. If a function wants to change the value of that variable, we need to tell the function that time_left is a global variable. We use a ***global*** statement in the function to do this. On the other hand, if we just want to use the value of the global variable, we do not need a ***global*** statement.

Arguments - revisited: we finish this lecture with a bit of technical detail. Here are two simple functions:

```
def func1(x):
    x = x + 1

def func2(L):
    L = L + [1]

a=3
M=[1,2,3]
func1(a)
func2(M)
```

When we call ***func1*** with a and ***func2*** with L, a question arises: ***do the functions change the value of a and L***?

The value of a is unchanged, but the value of L is changed. The reason has to with difference in the way that python handles numbers and lists.

- Lists are said to be mutable objects, meaning they can be changed.
- Numbers and string are immutable, meaning they cannot be changed.

Exercises

1. Write a function called rectangle that takes two integers m and n as arguments and prints out mXn box consisting of asterisk. Shown below is the output of rectangle (2,4).

```
****
****
```

2. Write a function called add_excitement that takes a list of string and adds an exclamation point (!) to the end of each string in the

list. The program should modify the original list and not return anything.

3. Modify the same function except that it should not modify the original list and should instead return a new list.
4. Write a function called sum_digits that is given an integer num and returns the sum of the digits of num.
5. Write a function called first_diff that is given two strings and returns the first location in which the string differ. It the string are identical, it should return -1.