Week 3 Notebook 1 - Functions and If Statements

August 21, 2018

1 Functions

Functions are blocks of code designed to make life easier for both the programmer and the user. If we have an often used code routine, rather than copying and pasting the same code in several places, we would write the code once and call it when we need it. Python has a large set of built-in functions (such as list.sort() and sum(list_of_numbers)) and you can add thousands of extra functions by using add-on packages, but it is also important to be able to write your own functions... and we'll do that shortly.

1.1 General Form for a Function

We must use the def keyword to begin a function. We also give the function a name (hopefully descriptive) and the number of arguments or values we intend to pass to the function. Many simple functions take only one argument, but we can pass multiple values (and even lists) if needed. We then provide a procedure to work with or on the supplied arguments. Once the function has been defined by the Python interpreter, we can call it from anywhere in the program as many times as we wish... even later on in the Jupyter notebook!

```
In [ ]: # define the function using a name
        def function_name(arg1, arg2):
            # do something with
            # arg1 and arg2
            # arg1 and arg2 are supplied
            # when you call the function

        # elsewhere in the program use the function name
        # to call the function
        function_name(val1, val2)
```

1.1.1 An example function

```
In [ ]: # write a function that squares a number and then subtracts the original number
        # then repeats the process on the first result
        def square_minus(n):
            x = (n**2-n)
            y = (x**2-x)
            print(y)
```

```
        # call the function as many times as you like
        square_minus(10)
        square_minus(12)
        square_minus(37)
        square_minus(62)
```

```
In [ ]: # call the function again
        square_minus(11)
```

One common error is to call the function before it's defined. You must define all functions early in the program before you call the function.

We can use a function for any type of data. Here are some sorting operations on a list WITHOUT a function.

```
In [ ]: astronauts = ['armstrong', 'aldrin', 'collins','lovell','glenn','shepard']

        # Put astronauts in alphabetical order.
        astronauts.sort()

        # Display the list in its current order.
        print("The astronauts are currently in alphabetical order.")
        for astronaut in astronauts:
            print(astronaut.title())

        # Put astronauts in reverse alphabetical order.
        astronauts.sort(reverse=True)

        # Display the list in its current order. Note the \n newline whitespace character
        print("\nThe astronauts are now in reverse alphabetical order.")
        for astronaut in astronauts:
            print(astronaut.title())
```

Now let's do the same thing using functions.

```
In [ ]: def show_astronauts(astronauts, message):
            # Print out a message, and then the list of astronauts
            print(message)
            for astronaut in astronauts:
                print(astronaut.title())

        astronauts = ['armstrong', 'aldrin', 'collins','lovell','glenn','shepard']

        # Put astronauts in alphabetical order.
        astronauts.sort()
        show_astronauts(astronauts, "The astronauts are currently in alphabetical order.")

        # Put astronauts in reverse alphabetical order.
        astronauts.sort(reverse=True)
        show_astronauts(astronauts, "\nThe astronauts are now in reverse alphabetical order.")
```

The function code is much cleaner and more efficient. When using functions, if we have a change to make in the function code it only needs to be modified in one place. If we don't use functions and repeat our code multiple times in the program, then we would have to modify each copy of the code. This isn't too bad if the program is short, but imagine a program with 5000 lines of code! Use functions as much as possible!

1.1.2 Returning values and introducing if statements

We don't have to use a print() statement to display a value from a function. If we are wanting to display a list of values as in the astronaut example, a print() statement is appropriate. However, if we are only wanting to return a single value, such as the result of a calculation or a single string, then 'return' is more appropriate.

```
In [ ]: def number(n):
            # Takes in a numerical value, and returns
            #  the name corresponding to that number.
            if n == 1:
                return 'one'
            elif n == 2:
                return 'two'
            elif n == 3:
                return 'three'
            elif n == 4:
                return 'four'
            else:
                return "I don't know that number!"

        # Let's try out our function.
        for num in range(0,6):
            number_name = number(num)
            print(num, number_name)
```

1.1.3 Explanation of the if-elif-else logic

I think it is best to explain the structure and logic of the program above and translate it into English.

- Define a function named 'number' and this function expects to receive a single value when it's called.
- IF the supplied value (stored in n) equals 1, then return the word 'one'
- Otherwise (elif or else if) if n equals 2, then return 'two'
- Otherwise if n equals 3, then return 'three'
- Otherwise if n equals 4, then return 'four'
- If there are no True statements (else), return "I don't know..."

Then use a for loop to supply the values from 0 through 5 and call the number() function using the loop index 'num' storing the returned expression in the number_name variable. Finally, print the number and the name... or the number and the 'else' "I don't know..." expression.

Can you rewrite the program with one less line of code?

A common error in functions is placing a print statement after the return statement. Once a function has issued a return, the function is over and anything indented at the function level AFTER the return will not be printed. Indentation is important in Python and a very common source of program errors. Here is an example:

```
In [ ]: def number(n):
            # Takes in a numerical value, and returns
            #  the name corresponding to that number.
            if n == 1:
                return 'one'
            elif n == 2:
                return 'two'
            elif n == 3:
                return 'three'
            elif n == 4:
                return 'four'
            else:
                return "I don't know that number!"

            # this print will never display
            print("Hello World!!")

        # Let's try out our function.
        for num in range(0,6):
            number_name = number(num)
            print(num, number_name)
```

We will dive into if/elif/else statements later in the lesson.

1.1.4 Try It Yourself

Write a function that finds the product of two numbers. Supply program several statements to verify the function works as expected.

2 *if* Statements

if statements are conditional statements and execute code depending on whether a condition is true or false. There are also *elif* (else if) and *else* key words for more complex conditionals. *if* statements provide a method for allowing the computer to make decisions... a very powerful feature!

```
In [ ]: # A list of foods I enjoy
        foods = ['pizza', 'spaghetti', 'hot wings', 'sub sandwiches']
        fav_food = 'spaghetti'

        # Print the foods, but flag my favorite
        for food in foods:
            if food == fav_food:
```

```
        # This food is my favorite... capitalize it
        print(food.title() + " is my favorite food!")
    else:
        # I like these foods
        print("I like " + food)
```

2.1 Logical Tests

Every if statement is either True or False. Both True and False are Python keywords and based on these two conditions, we can have Python make decisions. Again, this is very powerful!

2.1.1 Equality

The '==' sign tests for equality. Do NOT use '=' for equality testing. The single '=' is the variable assignment symbol. Here are some examples. Run each cell in order, but try to predict whether the result will be True or False before running or executing the cell.

```
In [ ]: 10 == 10

In [ ]: 21 == 20

In [ ]: 7.00 == 7

In [ ]: 'john' == 'John'

In [ ]: # case is important...
        'John' == 'John'

In [ ]: 'John'.lower() == 'john'

In [ ]: 'JOHN' == 'john'.upper()

In [ ]: 'John' == 'john'.title()

In [ ]: '12' == 12

In [ ]: '12' == str(12)
```

2.1.2 Inequalities

The inequalities are: * Unequal: != * Greater than: > * Less than: < * Greater than or equal to: >= * Less than or equal to: <=

```
In [ ]: 9 != 3

In [ ]: 8 != 8

In [ ]: 7 < 89

In [ ]: 89 < 7
```

```
In [ ]: 9 > 5

In [ ]: 8 > 10

In [ ]: 4 <= 4

In [ ]: 4 <= 5

In [ ]: 5 <= 4

In [ ]: 5 >= 3

In [ ]: 5 >= 5

In [ ]: 5 >= 9
```

We can also combine conditional statements using 'and' and 'or' as well as creating more complex conditionals

```
In [ ]: # read from left to right
        6 > 4 > 2

In [ ]: n = 7
        4 > n or n > 3

In [ ]: n = 5
        n > 3 and n > 9
```

2.1.3 Functions again... writing a useful function

The factorial is a mathematical operation used in probability and many other areas of mathematics in which an integer value is multiplied as follows:

n*(n-1)*(n-2)*...*1

The ! is used as the factorial sign and an example would be 5! = 5*4*3*2*1 = 120

As n increases, the size of the factorials increases very quickly. Our task will be to create a function which can calculate factorials.

```
In [ ]: # define the factorial function
        def fact(n):
            start = 1
            for x in range(n,0,-1):
                start *= x
            return start

        # calculate the factorial by calling the function
        fact(10)
```

Now try to find the factorial of a much larger value

```
In [ ]: fact(100)
```

Now you try one... how large can you go? You should be able to do at least 10000! and probably larger. If the calculation seems to hang or it takes too long, you may need to click on the Stop button or use one of the Kernel > Interrupt or Kernel > Restart commands.

Just an aside... I calculated 100000! and the value had 456574 digits. However, it only took about 20 seconds to calculate, so my assumption is that I could go higher. Try it if you dare!

```
In [ ]: # here is the calculation for 100000!
        # I store the value in f so I can find it's length
        f = str(fact(100000))

In [ ]: # print the value... it's big
        print(f)

In [ ]: # now print the length or number of digits by converting the value to a string using s
        print(len(str(f)))
```

2.2 Super-d Numbers

Super-d numbers are numbers such that

$$d \times n^d$$

contains d consecutive d numbers. That's a bit murky, so here is an example. A Super-3 number is a number such that when we perform the following calculation:

$$3 \times n^3$$

yields 3 consecutive 3s in the number. As you can imagine, Super-3 numbers would be difficult to find manually. However, Python with the string package can do this easily.

There is a conditional in this script:

```
if 'substring' in 'string:
    print(n, sd)
```

Basically we translate this as 'if the substring we are looking for (i.e. 333) is IN the number (which we've converted to a string using the str() function), then the if statment is True. Very handy!

Look closely at the # statements for an explanation of the program. You can verify the output to see if, for example, there are 3 consecutive 3s in the Super-3 number.

```
In [ ]: # define the Super-d function.  The function will take two variables, d and the upper
        # so if we want to search the first 1000 numbers for Super-3 numbers we would enter: s
        def superd(d, ub):
            # this variable will serve as a flag to determine whether or not we find
            # any Super-d numbers.  We'll start with False

            sflag = False

            # try the numbers from 1 to the upper bound (ub)
            for n in range(1, ub):
```

```
            # perform the calculation to create a potential Super-d number
            # this statement should work for any Super-d

            sd = d*n**d

            # we need to create the substring.  If d = 3, we need 333, if d = 4, we need 4
            # to do this, we'll use a nested loop... a loop inside a loop
            # first, create an empty variable substring... something creative like substr.
            # string... no space, just two consecutive single or double quotes

            substr = ''

            # use a for loop (just because... you could use while) to range from 1 to the
            # remember that if d = 3, range(1,d) would only give use 1 and 2.  The d+1 tak
            # the full range of 1, 2 and 3.

            for i in range(1, d+1):

                # concatenate the ds to the substr variable.  If d = 3, we'll end up with
                substr = substr + str(d)

            # this is almost magical... for Super-3 numbers, it checks to see if substr (o
            # in the potential Super-3 number.  If so, print the number and the Super-3 nu
            # if we don't find any Super-3 numbers (or Super-d numbers) less than the uppe
            # nothing is printed.

            if substr in str(sd):
                print(n, sd)

                # OK, we found at least one Super-d number.
                sflag = True

        # print that there were no Super-d numbers
        # this is still in the function, so the if block needs to be
        # indented.  If sflag is still False, it's value was unchanged
        # from its initial assignment and there were no Super-d numbers

        if sflag == False:
            print("There were no Super-" + str(d) + " numbers less than " + str(ub))

# test for Super-3 numbers and an upper bound of 1000
superd(3, 1000)

print(' ')

# test for Super-5 numbers and an upper bound of 1000
superd(5, 1000)
```

```
        # End of program
```

```
In [ ]: # try Super-5 numbers again, with an upper bound of 10000
```

2.2.1 Try It Yourself

Write 10 *if* statements using a combination of string and number statements. Have 5 statements print True and 5 statements print False. You may need to use individual print(x==y) type statements in order for all 10 to display properly.