Episode 5
Functions

Functions are the next major programming concept we have covered. Inevitably, when we write programs we will have units of code that must be executed over and over, possibly with some changes to some of the variables it uses. Or we may find that we are doing the same repeated actions in many of the programs we are writing. Functions solve these problems, as well as making our code better organized and easier to read. Functions also make it easier for multiple programmers to work together on a project.

A function is a set of statements that are given a collective name and can be called over and over. For example, suppose we need to add two numbers over and over again, but not the same two. We can write a function which takes those two numbers as its parameters. Then we simply invoke the function with different variables.

In Python the keyword for a function is `def` followed by the name, followed by parentheses. The parentheses are always required even if we might have no need for parameters. This statement begins a code block so we must terminate it with a colon.

```
def sum_of_two(x,y):
        """This function computes the sum of two variables, x and
y."""
        return x+y
```

The first line is a triple-quote comment. It is called a documentation string or *docstring*. It should provide a short explanation of what the function does and it must be indented to the same level as the body of the function code. We then return the result of the function. The return must be invoked or the function will not send anything back to the caller

Typing the function into your script merely sets it up. Nothing will happen until we call the function by name.

```
x=12
y=14
z=sum_of_two(x,y)
```

You should place function definitions at the top of your file, after any `import` statement (we will learn about `import` in our next episode).

The names of the variables in the statement in which we call the function do not have to be the same as their names in the function definition. The above code snippet is just as valid as

```
w=12
v=14
z=sum_of_two(w,v)
```

When the function is invoked, `w` and `z` are mapped to the corresponding positions in the function's parameter list; in this case `w->x` and `v->y`. For this reason, the parameters to a function are often called *dummy variables*.

When the interpreter sees a function call it finds the instructions in memory, does the computation, then replaces the function name in the calling statement by the return value. If you merely invoke its name and do not assign the result to a variable or use it in an expression or print it, the return value is lost.

You can return a single variable or an expression. Functions can return only one item but that item can be any valid type, so you can return compound types. Once the interpreter encounters the `return` statement it immediately returns control back to the caller (hence the "return"); no more statements in the function will be executed even if they are present. If you do not provide a `return` statement, Python will return the special value `None`.

Tuples

Python provides a data type that is especially useful for returning multiple results from a function. This is called a *tuple* and it is essentially the same thing as a list, with the important difference that it is immutable. It cannot be changed once it has been defined. We indicate tuples with round parentheses. Like a list, its elements may be different types.

```
mytuple=('one',2,3.0)
```

We can index and slice tuples just like lists.

```
mytuple[2]
mytuple[0:2]
```

We can unpack a tuple into variables

```
(number,i,x)=mytuple
```

When unpacking, the parentheses are optional

```
number,i,x=mytuple
```

We showed an example:

```
def sum_diff(x,y):
    return x+y, x-y


x=11.3
y=19.2
s,d=sum_diff(x,y)
print s,d
```

Now let's return to functions to learn about what computer programmers call variable *scope*. The scope is the section of code in which a variable has a particular value. Examine the following function:

```
def make_list(x,y,z):
    """This function creates a list and adds all arguments"""
    new_list=[x,y,z]
    return new_list
```

The variable new_list is *local* to the function. If we used another variable `new_list` outside the function, it would not interfere.

```
outside_list=make_list(10.,20.,30.)
print "outside_list=", outside_list
print "new_list=", new_list
```

Were you surprised that the interpreter says `new_list` is not defined? What about

```
new_list=[100.]
outside_list=make_list(100.,200.,300.)
print "outside_list=", outside_list
print "new_list=", new_list
```

There is one more thing we must understand about Python functions. When we pass an immutable variable as an argument to a function, its value outside the function is not changed even if we change the corresponding dummy variable inside the function. When we pass a mutable variable such as a list as an argument to a function, then if we change that variable inside the function, it *does* change outside the function. Sometimes this change to the external variable is called a *side effect*.

```
def side_effect(L,x):
   """This function adds an element to a list"""
   L.append(x)
   return None
l_outside=[1,2,3,4]
side_effect(l_outside,11)
print l_outside
print side_effect(l_outside,99)
```

What is `l_outside` after the second call to `side_effect`?

On the other hand,

```
def dummy(x,y):
    """Demonstration of dummy variables"""
    x+=y
    return x

x=12.
y=11.
print dummy(x,y)
print x,y
```

You should see that x did not change outside the function, because it is a floating-point value and that type is immutable. Now try running the above example of `dummy` but passing in two lists as `x` and `y`.

Anonymous Functions

A more advanced concept that we didn't cover in the video is the *anonymous function*. This is a single-line function that is not defined by `def` and so has no specific name. It may have multiple variables but its entire action must be capable of being described by a single expression. If it can be assigned to a variable or can take the place of a named function in certain situations, it is called a *lambda function* and the keyword used to define it is `lambda`.

```
power3=lambda x: x**3
```

We then invoke it with a call such as
```
power3(9.0)
```

Multiple variables would be defined in a tuple:

```
adder=lambda x,y: x+y
adder(11,12)
```

Lambda functions are most widely used when we need to provide a function as an argument to some other function. Functions of functions are called functionals. Three useful functionals for lists are `map`, `reduce`, and `filter`. Each takes a function as its first argument and a list as its second. The `map(f,L)` functional requires a function that takes a single parameter and returns a single scalar value. It is applied in sequence to each element of list L and returns the new list. The `reduce(f,L)` functional requires a function with two arguments that returns a scalar result. It is applied to the first two elements, then the result is applied to the second element, the result of that applied to the third, and so on. The `filter(f,L)` requires a function that takes a single argument and returns a Boolean value. While the name of a predefined conventional function can be the first argument, lambda functions are frequently used if the function can be expressed simply enough and it is not needed for any other purpose.

Examples:
```
doubled=map(lambda x: x*2, [1.,2.,3.,4.])
```
`doubled` is `[2.,4.,6.,8.]`
```
sum_it=reduce(lambda x,y:x+y,[1.,2.,3.,4.])
```
`sum_it` is `10.`
```
positive=filter(lambda x:x>0,[11.,-3.,21.,0.,3.])
```
`positive` is `[11.,21.,3.]`

Note: in Python 3 map and filter do not directly return lists. You need to convert them explicitly if you want the result as a list.
```
doubled=list(map(lambda x: x*2, [1.,2.,3.,4.])) #Python 3
positive=list(filter(lambda x:x>0,[11.,-3.,21.,0.,3.]))#Python 3
```

Another construct that takes an expression function is a *list comprehension*. List comprehensions are an alternative to a `for` loop or to the map() functional when the transformation is simple, and generally are faster than either `for` or `map`, but their syntax can be confusing. They make "new lists from old" by applying an expression to the list, optionally including an if.

Examples:
```
velocity=[1.,3.,5.,7.]
vsquared=[x**2 for x in velocity]
data=[4.,5.,-2.,8.,-3.,0.,7.]
sqroot=[math.sqrt(x) for x in data if x>0]
cubes=[x**3 for x in range(50)]
```

Notice that the list comprehension that returns sqroot is equivalent to

```
sqroot=[]
for x in data:
    if x>0: sqroot.append(x)
```

We now know enough about functions to learn in the next episode about the next construct for code organization, the *module*.

Further References

Functions are a core programming construct and there are many tutorials online. As with several topics, TutorialsPoint has a good discussion (https://www.tutorialspoint.com/python/python_functions.htm).