Python Programming

INFO.2820.061

Instructor: Andy Garside

Simple Functions, Lists, Tuples and Dictionaries

Introduction

We have used a number of built-in Python functions in our lectures, including the **print**, **input**, **open**, **round**, **str, help** and **len** functions. We have also used functions found in modules, like the **randint** function found in the **random** module. In this lecture we will expand our Python knowledge by learning how to write and call functions of our own. We will also discuss three new powerful Python data types, the **list,** the **tuple,** and the **dictionary**, including how to construct, use, and manipulate each.

Lists

In our lectures we have introduced a number of Python data types including the **int**, **float** and **str**. Of these data types, the **str** is the most complex, being a single data type consisting of a sequence of characters.

A Python string is, effectively, a **data structure**. A **data structure** combines multiple data values into a unit so they can be treated as one thing. Low-level languages, such as C, typically provide only simple built-in data types, like integers, floats and characters. High-level languages, like Python, provide more complex built-in data types. One of those more complex data types is the **list**.

A Python **list** is, in a number of ways, similar to a Python string. Both data types are sequence data types, with a string being a sequence of characters and a list being a sequence of objects. As with strings, you can access items in a list using indices, with the first element in a list being at index 0 and the last element in the list having an index of the length of the list minus 1.

As sequence data types, lists and strings support many of the same **operators** (for example, **+** and **[]**) and a number of the same **functions** (for example, **len**() and **print**()).

As objects, you manipulate both strings and lists using methods. For example, you manipulate Python strings using string object methods like **split**(), **lower**(), **upper**(), **isalpha**(), **isdigit**(), etcetera. We will see that you manipulate Python lists using a set of list object methods including **append**(), **insert**() and **extend**().

Why lists?

The list is an incredibly useful data type. In programming, it is common to maintain a list of items of indefinite size. For example, you may want to keep a list of…

- names or phone numbers.
- file names or directories.
- Auto part numbers
- Air temperatures
- Hurricane dates
- Stock Market closing values
- Airline departure/arrival times

The possible uses for a list of items are almost limitless.

Basic List Operators and List Literals

We have seen that literal strings are written as a sequence of characters enclosed in quotes (either single or double quotes). For example

```
saying = "Hello World"
```

You define a literal list by enclosing the list elements in square brackets, [], and separating the elements by commas. For example

```
temperatures = [ 83.2, 85.6, 88.0]   # A list with three numbers.
words = [“Hello”, “World”]           # A list with two strings.
Y = [“Fish”, 4, 23.3]    # A list with three elements: a string,
                         # an integer and a float.

z = []              # An empty list (similar to “” for s string)
```

Since a list can contain any object, you can also have a list of lists. For example, the following list contains a sequence of lists where each list contains two floating-point elements representing the high and low temperatures for a series of days.

```
temperatures = [ [83.2, 71.3],  [85.6, 74.6], [88.0, 77.3] ]
```

It is important to note that when you create a list using variables or expressions, the Python interpreter evaluates the variables and expressions before assigning the values to the list. For example, below we create variables x and y and use them in a list declaration. Notice that neither variable exists as such within the created list. The variables and expressions are evaluated and their values are placed in the list.

```
>>> x = 7
>>> y = 8
>>> z = [x, x + y]
>>> print(z)    # NOTE: The print function strips quotes from
                # a string but does not strip the [] from a list.
[7, 15]
```

The `list` function.

Earlier in our lectures we made use of the **list** function in the following example.

```
>>> list(range(1,4))
[1, 2, 3]
```

We explained that the **range**() function generated a sequence of numbers and the **list**() function created a Python list whose elements were the generated numbers. In actuality, the list() function will create a list from any iterable sequence of elements. For example, if you give **list()** a string, it will create a list where each element is a character of that string.

```
>>> list("Hello World!")
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

The `len` function and the `[]`, `+` and `==` operators.

The **len()** function and the **[]** operator work on a **list** just as they do for strings.

```
>>> len([20, 18, 7])
3
>>> nums = [20, 18, 7]
>>> len(nums)
3
>>> nums[0]
20
>>> nums[0:2]          # Notice the [] operator returns a list when
                       # two or more elements are referenced.
[20, 18]
```

The **+** and **==** operators also work similarly to how they work for strings.

```
>>> nums2 = nums + [33, 41]  # Concatenate two lists
>>> print(nums2)
[20, 18, 7, 33, 41]

>>> nums2 == [20, 18, 7, 33, 41]   # Compare two lists
True
```

Testing if an element exists in a list

You can use the **`in`** operator with an **if** statement to test whether an element exists in a **list**.

```
temperatures = [ 83.2, 85.6, 88.0]
if 85.6 in temperatures:
    print("Found it!")
```

Found it!

Printing elements of a list

As we have seen, you can pass a **list** to the **print** function to display its contents.

```
>>> temperatures = [ 83.2, 85.6, 88.0]
>>> print(temperatures)
[83.2, 85.6, 88.0]
```

While this is a quick and simple way to display the contents of a **list**, it's not always the best way to display results to the user. Often, a better solution is to use a **for** loop. Since a **list** is an iterable data type, you can use it with the **in** operator in a **for** loop as follows. Each time through the loop, the next value in the list is assigned to the loop variable temp.

```
print("Daily high temperatures")
temperatures = [ 83.2, 85.6, 88.0]
for temp in temperatures:
    print(temp)

Daily high temperatures
83.2
85.6
88.0
```

Summary

The following table summarizes the list operators and functions we've covered.

Operator or Function	Description
<listname>[<integer expression>]	Access the list element at the position (subscript) represented by the integer expression.
<listname>[<start>:<end>]	Returns a new list which is a "slice" of the list <listname>
<listname1> + <listname2>	Returns a new list consisting of the elements of <listname1> and <listname2>
print(<listname>)	Prints the literal representation of list <listname>
len(<listname>)	Returns the number of elements in the list <listname>

list(range(<upper>))	Returns a list containing the integers in the range 0 through <upper> - 1.
<listname1> <operator> <listname2> where, <operator> is one of ==, !=, <, >, <=, >=	Compares two lists, element by element. Returns **True** if all results are true. Otherwise, returns **False**.
For <variable> in <listname>: <statements>	Iterates through the elements of list <listname> assigning each successive element of the list to <variable>.
<value> in <listname>	Returns **True** if <value> exists in list <listname>, otherwise returns **False**.

Replacing a List Element

One of the most significant differences between a **string** and a **list** is that a string cannot be changed (it is **immutable**) but a list **can** be changed (it is **mutable**). That is, you can remove, insert or replace values in a **list** while the list maintains its name (identity).

If strings are immutable, why does the following work?

```
>>> saying = "Hello World"
>>> saying = saying + "!"
>>> print(saying)
Hello World!
```

When we say an object is **immutable** we mean the contents of the object cannot be changed. In the example above, the variable *saying* is created as a string object with the value “Hello World”. When the second line is executed, the Python Virtual Machine creates a new string object containing enough memory for the original characters of *saying* and the string “!”. The PVM then copies the contents of the original string object to the new object along with the characters of string “!”. The variable *saying* is then assigned to this new string object. The memory for the original string object and the original string “!” are then deleted when the Python Virtual Machine does its memory “garbage collection”.

In contrast, in the **list** example below, the second line concatenates the list [3, 4] to *list1* by simply extending the already existing list object called *list1*. That is, the variable *list1* points to the same **list** object both before and after line 2. For this reason, **list** operations can be much more efficient than string operations since memory need not be as frequently allocated and deallocated.

```
>>> list1 = [ 1, 2 ]
>>> list1 = list1 + [ 3, 4 ]
>>> print(list1)
[1, 2, 3, 4]
```

You can replace a list element by assigning a new value to element using the syntax **<listname>[<index>]** where **<index>** is the index of the element you wish to replace. For example,

```
>>> list1
[1, 2, 3, 4]
>>> list1[2] = 10
>>> list1
[1, 2, 10, 4]
```

> Note that with **lists**, square brackets, [], are used for two distinct purposes: One, to define a list, and two, as the subscript operator, used to reference an element of a list at a specific index.

As another example, consider the following **for** loop that converts a list of temperatures from Fahrenheit to Celsius.

```
temps = [83.2, 85.6, 88.0]
for index in range(len(temps)):
    temps[index] = (temps[index] - 32) * 5 / 9
    print(round(temps[index]))

28
30
31
```

The **for** loop produces a sequence of values from 0 to the length of the list **temps** minus 1 using the statement **`range(len(temps))`**. The loop variable **index** is assigned a new value on each pass.

The first line in the body of the **for** loop converts the current temperature at subscript **index** to Celsius and assigns the result to the list at that same index, effectively replacing the original Fahrenheit value.

The second line in the body reads the converted temperature from the same subscript index, rounds it to the nearest integer, and prints the result.

Inserting, Appending and Removing a List Element

A **list** is an object and, as such, has a number of **methods** for operating on the list. This includes methods for **inserting**, **appending** and **removing** elements from a list.

List Method	Description
<listname>.append(<element>)	Adds <element> to the end of list <listname>
<listname1>.extend(<listname2>)	Adds the elements of <listname2> to <listname1>
<listname>.insert(<index>, <element>)	Inserts <element> at <index> if <index> is less than the length of <listname>. Otherwise, inserts <element> at the end of <listname>
<listname>.pop(<index>)	Remove and return the element at <index> in <listname>. If <index> is not given, removes and returns the last element in <listname>. If <index> is out of range, an exception is raised.

Below are some examples using the list methods above.

```
>>> temps = [ 83.2, 85.6, 88.0]
>>> temps.insert(1, 86.1)        # insert 86.1 at index 1
>>> temps
[83.2, 86.1, 85.6, 88.0]

>>> temps.extend([81.5, 90.9])   # add elements 81.5 and 90.9
>>> temps
[83.2, 86.1, 85.6, 88.0, 81.5, 90.9]

>>> temps.append(79.2)           # append element 79.2
>>> temps
[83.2, 86.1, 85.6, 88.0, 81.5, 90.9, 79.2]

>>> temps.pop(2)                 # remove the element at index 2
85.6
>>> temps
[83.2, 86.1, 88.0, 81.5, 90.9, 79.2]
```

The **extend** method and the **+** operator do similar things. However, there is one significant difference between the two. The **extend** method adds new elements to the end of the current list object. The **+** operator creates a new list consisting of the elements of its two operands. This makes the **+** operator less efficient and it also has an interesting side effect, as seen below

Notice that the **extend** method adds the elements 3 and 4 to the existing list object, n.

```
>>> n = [1, 2]
>>> n.extend([3, 4])
>>> n
[1, 2, 3, 4]
```

When we use the **+** operator to add elements 5 and 6, notice that a new list is created containing the additional values, but the list, **n**, does not change.

```
>>> n + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> n
[1, 2, 3, 4]
```

If we want to change the original list, **n**, using the **+** operator, we need to assign the newly created list to **n** using the assignment operator.

```
>>> n = n + [5, 6]
>>> n
[1, 2, 3, 4, 5, 6]
```

Finding the index of an element in a List

Unfortunately, unlike the string object, the **list** object does not include a **find** method. Finding the index of an element in a list involves two steps. You first check if the element exists in the list, using the **in** operator, and then use the **index** method to find the index of the element. For example

```
temps = [ 83.2, 85.6, 88.0]
if 88.0 in temps:     # First check if the element exists!
    print(temps.index(88.0))

2
```

It is important to first check that the element exists in the list since Python will throw an exception if you ask for the index of an element and that index does not exist in the list.

Sorting a List

The elements in a list are maintained in the order in which they are entered. Sometimes, it is useful to rearrange the elements of the list into a sorted order, either ascending or descending. This can be done using the list object's **sort** method.

```
n = [4,3,10,8]
n.sort()
print(n)

[3, 4, 8, 10]
```

By default, the **sort** method sorts elements in ascending order. You can sort the elements in descending (reverse) order by setting the optional argument **reverse** to **True**, as shown below.

```
n.sort(reverse=True)
print(n)

[10, 8, 4, 3]
```

Note: We will discuss optional arguments in detail in a future lecture.

Mutator Methods and the Value None

A common mistake among new Python programmers is to attempt to use the **list** object's **sort** method as follows

```
n = [4,3,10,8]
newlist = n.sort()  # Wrong!
print(newlist)
```

```
None
```

Above, the programmer has tried to sort the list, **n**, and assign the result to a new list called **newlist**. The problem is that the **sort** method does not return a list, it returns the NoneType value **None**. This is because the **sort** method sorts the elements of a list **in-place**. That is, given a list, **n**, the call **n.sort()** mutates the order of the elements of **n**, so that the original list is now sorted. Methods that behave in this manner are known as **mutators**, since they simply mutate the internal state of the original object.

There are a number of list object methods that are **mutators** and return the value **None.** These include the **insert**, **append**, **extend** and **sort** methods.

Aliasing and Side Effects

Consider the following program. In the code, after creating the list n1, n1 is assigned to n2 and, subsequently, the first element of n1 is changed to the value 77. Notice that this results in the same change appearing in the list n2!

```
n1 = [1, 2, 3]     # create n1 with three elements
print(n1)
n2 = n1            # assign n2 = n1
n1[0] = 77         # change the first element in n1 to 77
print(n1)          # print n1
print(n2)          # print n2. THE FIRST ELEMENT HAS ALSO CHANGED!
```

```
[1, 2, 3]
[77, 2, 3]
[77, 2, 3]
```

In Python, an assignment statement like A = B effectively says that name A should refer to the same object as B, or simply, that A is an **alias** for B. If B is an **immutable** object, like an **integer**, **float** or **string**, this has the same effect as copying the value of B to A, since the object pointed to by B cannot be changed. If a future assignment like B = 10 should occur, a new object with the value 10 will be created and B will be set to point to that new object, leaving the object pointed to by A unchanged.

When an object is **mutable**, like a **list**, an assignment like A = B can result in unexpected **side effects**, as shown in the program above. This is because Python does not make copies of **mutable** objects when they are modified. Since A is an alias for B, any change made to the list B is seen in list A.

Aliasing like this is very efficient since objects are not routinely copied every time they are assigned. However, it means the programmer must be aware of this behavior and explicitly make a copy of **mutable** objects anytime the program modifies an object and an alias exists.

The Python **list** function can be used to make a copy of a **list**. It takes the name of an existing list as an argument and returns a copy of the list. The syntax is as follows

<new list> = list(<existing list>)

The program below is identical to the one we saw earlier except the assignment n2 = n1 has been change to n2 = list(n1). Notice that after changing n1[0] = 77, n2 remains unchanged.

```
n1 = [1, 2, 3]
print(n1)
n2 = list(n1)    # Make a COPY of n1 and assign it the name n2
n1[0] = 77
print(n1)
print(n2)
```

```
[1, 2, 3]
[77, 2, 3]
[1, 2, 3]
```

The `is` operator: Object Identity vs Structural Equivalence

Since Python uses aliases, there are times when you will want to know if two variable names (identifiers) with the same value point to the same object (called **object identity**) or whether they point to two different objects with identical values (known as **structural equivalence**).

The Python **`is`** operator is used to check for object identity. For example,

```
n1 = [1, 2, 3]
n2 = n1          # creates an alias for n1 called n2 (same object)
n3 = list(n1)    # creates a new list called n3 identical to n1
print(n2 is n1)
print(n3 is n1)

True
False
```

Unlike the **`is`** operator, the == operator returns **True** if two identifiers refer to the same object *or* to two different but equivalent objects. For example,

```
print(n2 == n1)
print(n3 == n1)

True
True
```

Tuples

A **tuple** is a sequence data type that is very similar to a **list**. You can define and print a tuple much as you would a list, with the difference being that tuples are defined and displayed with enclosing parenthesis, (), rather than the square brackets, [], used for a list. For example,

```
t1 = (1, 2, 3)
print(t1)

(1, 2, 3)
```

When creating a **tuple** of **one element** it is required that the element be followed by a comma. This is necessary because a statement like `t1 = (1)` is equivalent to `t1 = 1`, which is an assignment of the integer 1 to the variable t1.

```
t1 = (1)
print(t1)

1
```

Including a comma after the first element tells Python to, instead, assign t1 a tuple with one element whose value is 1.

```
t1 = (1,)
print(t1)
```

```
(1,)
```

The most significant functional difference between a **list** and a **tuple** is that a list is **mutable** and a **tuple** is **not**. This means the **mutator** methods of **lists** do not exist for tuples, since the data of a tuple cannot be modified. However, most of the operators and functions used with a **list** will work with **tuples**. For example, the following code makes use of the **+** operator to concatenate two tuples.

```
t1 = (1, 2, 3)
t2 = t1 + (4, 5)
print(t2)
```

```
(1, 2, 3, 4, 5)
```

In general, you should choose a **tuple** over a **list** whenever the elements should not be modified. For example, a tuple would be appropriate for a sequence of strings used to represent equivalent numbers, like

nums = (“one”,”two”,”three”)

Using a tuple ensures that the elements will not be modified at runtime by the code that uses it.

When needed, you can also create a tuple from a list using the **tuple** function.

```
nums = ["one","two","three"]
print(tuple(nums))
```

```
('one', 'two', 'three')
```

Dictionaries

Lists are incredibly useful data types. They provide a way to collect a sequence of objects (possibly of different types) in a linear fashion. Once you create a list, you can add new elements to the beginning of the list, the end, or even in-between existing elements. And, when you traverse a list, you're guaranteed to find the same elements in the same order each time.

However, there are some drawbacks to a list. One of the drawbacks is the time it takes to find something in a list. In order to find a specific element you may have to examine every element in the list and, as the list gets larger, this can become prohibitively slow. Also, once you find an element in a list, you'll need to use an integer index to reference the element. While this is efficient, an integer index is not always a natural way to refer to an element.

Consider a problem where you have a list of names and phone numbers that you want to read from a file.

"Pradeep Acharya" "301-555-3276"
"James Chen" "256-555-3331"

You could put the data into a list like the following

```
data = ["Pradeep Acharya","301-555-3276","James Chen","256-555-3331"]
```

The problem is, as this list gets large, the time to find an element increases. Also, once you find an element you're looking for, your reference to that element is an integer index rather than something more descriptive, like the person's name.

To help address these issues, Python supports a **mutable** data type called a **dictionary**. A **dictionary** is an **associative list** (also known as an **associative array**) where the elements (**values**) are associated with a set of **keys** used to access the values.

The **keys** of a dictionary can be of any **immutable** data type and the **values** can be of any data type, **mutable** or **immutable** (even another dictionary). Further, the keys and values in a dictionary do not all have to be of the same object type.

Defining dictionaries

A Python dictionary is written as a sequence of key:value pairs separated by commas and enclosed in curly braces, {}. The general form of a dictionary declaration is as follows

{ <key1> : <value1>,...<keyN> : <valueN> }

For example, we can define a dictionary to hold our names and phone numbers as follows

```
data = {"Pradeep Acharya":"301-555-3276", "James Chen":"256-555-3331"}
```

In the above dictionary, the **keys** are the names and the **values** are the phone numbers.

An empty dictionary is defined as {}.

> Note: though the keys and values of a dictionary look as though they are in a fixed order, they are not. That is, if you add key1/value1 and key2/value2, there is no guarantee that the items will display in that order if you print the dictionary. Never make an assumption about key/value order with a standard Python dictionary.

Accessing values

You can access the values of a dictionary by providing a dictionary key to the subscript operator. For example, given the **data** dictionary we defined above, we can find the phone number for "James Chen" by doing

```
print(data["James Chen"])
```

```
256-555-3331
```

The problem with the subscript operator is that it will raise an exception if the key does not exist in the dictionary. To avoid this problem, we should first test if the key exists using the **in** operator, as follows

```
name = "James Chen"
if name in data:  # Test if the key exists before using it below!
    print(data[name])
```

```
256-555-3331
```

You can also access the values of a dictionary using the dictionary object's **get** method. The **get** method takes two arguments, the **key** and a **default** value to return if the key does not exist. Using the **get** method avoids the risk of exception if the key does not exist.

```
print(data.get("Grim Reaper", None))
```

```
None
```

Adding and replacing values

You use the following subscript syntax to add a new key/value pair to an existing dictionary.

<dictionaryname>**[**<key>**]** = <value>

Note that the dictionary must exist before you can add values to it. For this reason, programmers often create an empty dictionary using {} before they begin adding values. The program below dynamically creates the **data** dictionary we defined earlier by first defining an empty dictionary and then adding each key/value pair to the dictionary.

```
data = {}
data["Pradeep Acharya"] = "301-555-3276"
data["James Chen"] = "256-555-3331"
print(data)
```

```
{'Pradeep Acharya': '301-555-3276', 'James Chen': '256-555-3331'}
```

The subscript operator is also used to replace a current key value with a new value. The following code replaces the original phone number assigned to "Pradeep Acharya" with the number `"412-555-2121"`.

```
data["Pradeep Acharya"] = "412-555-2121"
print(data)
```

```
{'Pradeep Acharya': '412-555-2121', 'James Chen': '256-555-3331'}
```

Removing keys/values

You remove a key/value pair from a dictionary using the dictionary object's **pop** method. The pop method is typically called with two arguments, the **key** and the **default value** to return if the key doesn't exist. If the second argument (the default value) is not specified and the key does not exist, the **pop** method will raise an exception.

```
data.pop("Pradeep Acharya", None)
print(data)
```

```
{'James Chen': '256-555-3331'}
```

Traversing a Dictionary

There are multiple ways to traverse a dictionary.

A common way is to use a **for** loop with the **in** operator. In the example below, the variable **key** in the loop header is assigned a different key from the dictionary **data** each time through the loop. The **print** statement then prints the **key** and the **value** found at that key using **data[key]**. Keep in mind that there's no guarantee that the for loop returns the keys in the same order you added them.

```
data = {"Pradeep Acharya":"301-555-3276", "James Chen":"256-555-3331"}
for key in data:
    print(key, data[key])
```

```
Pradeep Acharya 301-555-3276
James Chen 256-555-3331
```

A second way to traverse a dictionary is using the dictionary object's **items()** method. The **items** method returns a list of tuples where the first item in each tuple is a key and the second item is the value. For example,

```
print(list(data.items()))
```

```
[('Pradeep Acharya', '301-555-3276'), ('James Chen', '256-555-3331')]
```

When the elements of a list are tuples, you can use the following form of **for** loop to assign each item in the tuple to different variables (identifiers). In this example, each pass through the loop will assign the variables **key** and **value** a key/value pair from the **data** list. The **print** statement in the body of the loop will then print the **key** and **value**.

```
for (key, value) in data.items():
    print(key, value)
```

```
Pradeep Acharya 301-555-3276
James Chen 256-555-3331
```

Finally, if you want to extract values based on a specific order of the keys, you can use the dictionary object's **keys**() method to produce a list of keys that you can sort or modify before using them in a loop. For example, the following loop prints the names and phone numbers of everyone in our dictionary, **data**, in alphabetical order.

```
data = {"Pradeep Acharya":"301-555-3276", "James Chen":"256-555-3331"}
keys = list(data.keys())  # Get list of the keys/values as tuples
keys.sort()               # Sort the list
for key in keys:          # Iterate through the sorted keys
    print(key, data[key]) # print the key and value
```

```
James Chen 256-555-3331
Pradeep Acharya 301-555-3276
```

Dictionary Operations

Below is a table of common dictionary operations and a description of what each does. In the table, D, represents a dictionary name.

Dictionary Operation	Description
len(D)	Returns the number of entries in D
D[key]	Used to insert a new key/value pair, replace an existing value at key, or

	extract an existing value at key.
D.get(key [, default])	Returns the value at key. Returns default if key does not exist. If default is omitted and key does not exist, an error is raised.
D.pop(key [, default])	Removes key and returns the associated value. Returns the default if key does not exist. If default is omitted and key does not exist, an error is raised.
list(D.keys())	Returns a list of keys
list(D.values())	Returns a list of the values
list(D.items())	Returns a list of tuples containing the key/value pairs
D.clear()	Removes all the keys
for key in D: <statements>	Iterate through all the keys in D, assigning a new key on each pass.

Simple Functions

We have seen and used functions in a number of places this semester. For example, we've used the Python built-in function **print** as follows:

```
>>> print("Hello world!")
Hello world!
```

Functions provide the ability to group statements into a reusable body of code that can be called from multiple locations, thereby eliminating repeated code and making it easier to share code with other programs. In this section, we discuss the syntax of simple functions and describe how a Python programmer can define and user their own functions.

Simple Function Syntax

By definition, a Python function takes zero or more arguments and returns a result. For example, in its simplest form, the **round** function takes a number as an argument and returns the floating-point equivalent of that number rounded to the nearest integer.

```
>>> x = round(3.14159)
>>> x
3
```

To define a Python function you use the following syntax:

```
def <function name>(<parameter-1>, …, <parameter-n>):
    <body>
```

The function definition begins with a **header**, which includes the keyword **def**, followed by the function's name, a set of enclosing parenthesis, and a terminating colon. If the function has **parameters**, they are listed within the parenthesis separated by commas.

Parameters verses Arguments: An **argument** is a literal value or variable provided in the call to a function. For example, in the statement y = my_func(x, 7), x and 7 are **arguments**. A **parameter** is the name given to the **argument** within the function definition. For example, in the function **header** below the **parameter** val1 maps to the first **argument**, x, and the **parameter** val2 maps to the second **argument**, 7.

def my_func(val1,val2):

Function names: Function names typically follow the same naming rules as **variable names**. That is, function names should consist of alphabetic and numeric characters separated by underscores or in Camecase.

The **body** of a function can be any valid sequence of Python statements. The **body** must be indented in a manner similar to how you would indent the body of a loop or conditional statement.

A function is **called** by providing the function's name along with any **arguments** contained in parenthesis. When the **call** is made, the **arguments** are mapped to their respective **parameters** in the function definition. The names of the **parameters** can then be used as **variables** within the function itself.

The `return` Statement

When a function concludes execution, it automatically transfers control back to the caller of the function, and execution continues at the statement following the call. Since a function always returns a value, a function, by default, automatically returns a value of **None**.

Note: **None** is a keyword in Python. It indicates a **null** value (object). You can assign **None** to any variable, which simply indicates the variable has been created but has no value. **None** is not the same as 0, **False**, or the empty string. It is a datatype of its own (**NoneType**) that can only have the value **None**. Comparing **None** to *anything* will always return **False** except **None** itself.

Interesting note: All variables that are assigned **None** point to the *same* object.

In many instances, it's desirable to return a function value other than the default. In Python, you do this with the **return** statement. The **return** statement takes the following form,

return <expression>

When a **return** statement is reached within a function, the function terminates and control is transferred back to the caller. The value represented by <expression> is returned to the caller as the return value of the function. A function may have **multiple return** statements, representing multiple terminating points within the body, each of which may return different values.

Simple Function Example

The code below defines a function called **square** which calculates the square of a number and returns it as the function's value.

Note: Functions can be defined in a Python shell, but it is more convenient to define functions in a Python script where they can conveniently be edited and saved.

```
def square(x):
    """Returns the square of x."""
    return x * x

print(square(4))
```

The function **header** defines the function name as **square** and includes one parameter, **x**.

The first line in the **body** of the function is a **docstring**. All function definitions should include a docstring. A docstring not only provides local documentation for a function but can be accessed by the programmer using the **help()** function in the Python shell or, within a script, using the call **<function-name>.__doc__**

The second line in the function **body** is a **return** statement containing the expression **x * x**. When this statement is reached, the expression x * x is calculated and control is transferred back to the caller along with the computed value of the expression.

The **print** statement in the program is outside the function. We know this because of its indentation. The **print** statement itself includes a call to the function **square** as its expression. When called, **square** passes the argument **4** as the value of the first and only parameter to the **square** function. **square** then calculates the value **4 * 4** and returns **16** to the **print** function which then displays a value.

> Note: like variables, Python functions must typically be **defined before** they are called. For this reason, you will typically find function definitions towards the beginning/top of a script.

Boolean Functions

A Boolean function is a function that returns **True** or **False**. Typically, such a function takes one or more parameters and tests for the presence or absence of some property. For example, the function below takes a single parameter, **x**. It returns **True** if **x** is an **odd** integer and **False** otherwise. Notice that the function includes two **return** statements. No matter what path the code takes, the function will always return a **True** or **False** value.

```
def is_odd(x):
    """Returns True if x is odd, otherwise returns False."""
    if x % 2 == 1:
        return True
    else:
        return False
```

The `main` Function

Some programming languages, like C, require a program to have a **main** function. A **main** function is the function that is first called whenever the program is run. In C, all other code is subsequently called from within the **main** function.

Python **does not** require a program to have a **main** function. By default, when a Python program is run, execution begins at the first statement that appears outside of any function (as well as outside some other constructs we will discuss later) and execution continues with successive statements within the file.

For example, let's assume we have the following Python script (module) called **square.py** that contains the **square** function we saw earlier along with a prompt for the user to enter a number, and a **print** statement to print the result. When this program is run, the Python Virtual Machine compiles the **square** function and the three lines below it into byte-code. It then begins executing the program beginning with the first statement outside the function, `input("Enter a number: ")`.

```
def square(x):
    """Returns the square of x."""
    return x * x

number = float(input("Enter a number: "))
result = square(number)
print("The square of", number, "is", result)
```

While this form of program execution is common, it has a significant drawback in that it is not possible to include the module **square.py** within another program without executing the statements outside the function. This makes it impossible to share the function **square** with other programs without executing this additional code.

To avoid this problem, we can make use of a special variable called **__name__**. When a Python module is run from the **command line** or from within a **Python shell** (like IDLE), the variable **__name__** is assigned the string value "**__main__**". When the module is **imported** into another module, the variable **__name__** is, instead, given the name of the module. In this case, the string "**square**".

If we take the statements outside the **square** function and put them into a function of their own, we can then use an **if** statement to conditionally execute the statements only when the script is run from the command line or from a Python shell.

The script (module) below includes a function called **main** that contains the three statements used for prompting the user for a number, calling the **square** function, and printing the result. The **if** statement at the bottom of the module tests whether the variable __**name**__ is equal to **“__main__”**, indicating the module is being run from the command line or from the Python shell. When the condition is **True**, the **main** function is called, the user is prompted for input, and the results are displayed.

If **square.py** is imported into another module, the value of __**name**__ will be **“square”** and the **if** condition will be **False**. As a result, the **main** function will not be executed. This means **square.py** can be imported into other modules and the function **square** can be used without needing to call the **main** function.

```
"""
File: square.py
Illustrates the definition of a main function.
"""

def main():
    """The main function for this script."""
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)

def square(x):
    """Returns the square of x."""
    return x * x

# The entry point for program execution
if __name__ == "__main__":
    main()
```

Are function arguments passed by value, reference, other?

When a variable is passed as an argument to a function it is important to know whether the function might change the value of the original variable. For example, in the following call, the function my_func takes a single parameter and assigns a value to the variable, **y**. It is not clear from this call, however, whether the value of the variable **x** may be changed by the function.

y = my_func(x)

In programming, function parameters are often referred to as **call-by-reference** if the called function may change them and **call-by-value** if it cannot. For example, the C programming language supports both call-by-reference and call-by value parameters, and the programmer explicitly indicates how the arguments will be passed. In C, **Call-by-reference** arguments are typically passed as a **pointer** to a variable so that the function can explicitly modify the original variable's value. **Call-by-value** arguments are handled by passing a copy of the argument's value to the function's parameter so that the function can use the value without modifying the original argument.

Python does things a little differently. Python passes arguments in a manner known as **Call-by-Object** (also known as **Call-by-Object-Reference**). You can think of this as passing a reference to the object by value. That is, a reference to the object is passed and the object's reference cannot change, but content of the object might change depending on whether the object type is **immutable** or **mutable**. If the contents of the argument (object) are **immutable**, like an **integer** or **string**, the behavior is effectively **Call-by-value**. If the contents of the argument (object) are **mutable**, like a **list** or a **dictionary**, the effect is more like **Call-by-Reference**.

For example, the code below defines three functions. The function **change_int** adds 1 to an integer passed as an argument. The function **change_str** appends the string "1" to a string passed as an argument. And the function **change_list** attempts to change the first element of a list passed as an argument to the value 1.

```
def change_int(x):
    x += 1
    return x

def change_str(x):
    x += "1"
    return x
```

```
def change_list(x):
    x[0] = 1
    return x
```

Below the three functions are called with the values 3, “3” and [3], respectively.

```
x = 3
y = change_int(x)
print(x,y)

x = "3"
y = change_str(x)
print(x,y)

x = [3]
y = change_list(x)
print(x,y)
```

In the output below, the first line is the output to the call to `change_int`. Notice the original argument, x, remains unchanged after the call, while the return value is 4. This is because the object, x, passed to `change_int` is an integer, which is immutable.

The second line is the output from the call to `change_str`. Again, notice that the original argument, x, remains “3” after the call, while the return value is “31”. x remains unchanged because x is a string, which is immutable.

The third line is the output from the call to `change_list`. Notice the original arugment, x, **does** change. After the call, the list x contains the value [1], just like the **list** y. The **reference** to the **list** object x has not changed, but the **contents** of the **list** object has changed. This is because a Python **list** is mutable.

```
3 4
3 31
[1] [1]
```

Conclusion

In this lecture we have expanded our Python knowledge by learning how to write and call functions of our own. We have also introduced three new powerful Python data types, the **list,** the **tuple,** and the **dictionary**, including how to construct, use, and manipulate each. We will make use of all of these items in our future work.