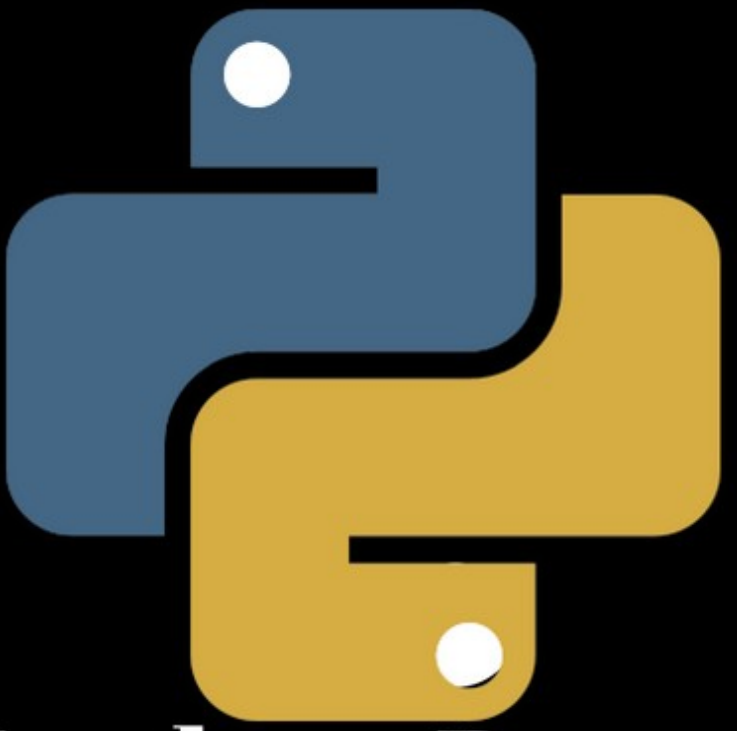
Advanced Python WORKBOOK



Matthew Dewey

Introduction

Hello and welcome to your Advanced Python Course.

First, allow me to introduce myself. I am Matthew Dewey, lead programming instructor at Programming Tut.com. I have studied several programming languages over the years and one of the greatest I will be teaching you further on in this course.

This is an advanced course on Python programming. The content within will include many advanced tips and methods that Python programmers use to start off with, but soon followed with complex variables, methods and functions. The content will help beginners who have been studying Python programming for a while reach the next level of their programming career. We not only focus on teaching these advanced concepts to expand your Python programming, but to also prepare you for OOP, Object Orientated Programming. Note, this course does not contain content on OOP, but rather increases your programming know-how to the point you are ready for the challenge

On top of all this I will provide you with a workbook which contains the information within this course as well as quizzes to help test your knowledge. This workbook can be used to keep up with lessons if you are unable to watch the videos, but I highly recommend following along in the workbook to maximise your learning potential.

Learn Lists, Tuples, Sets, Dictionaries and more in this course. Ending off with learning some vital programming to OOP, Methods and Functions. At the end of this course you will receive a project, but don't stress too much as I will also provide you with one of the answers that you could use. Programming involves a lot of logical creativity, so I always expect a different answer from each student.

LISTS

A list is a form of variable that can be used to hold an array of object types. For example, supposing you have multiple separate values that you want to be contained within one variable you would use a list.

A great example would be to create a variable to contain the data of a person. Let's say the first name, the last name and the age.

You would then create a list variable and place the values within.

Note, that lists can be analyzed and data retrieved using indexing and slicing. Indexing and slicing we will learn later in this course and practice thereafter.

For now, bear in mind that lists can be very useful data containers for any programmer collecting large amounts of data that need to be divided into the various sections.

To create a list in python is as follows.

```
test_list = ['hi', 'my name', 'is', 'John']
```

As you will notice from the line we start off the creation of a list variable as we would any other variable. We type in the variable name, follow it with an equals symbol and then we place the values. When creating a list we open a set of squared brackets and begin to place our values within.

We separate each value with a comma and with that in mind you will note I placed 4 values within the list. Hi' being the first and 'John being the last. Lists are fantastic variable to be used when processing strings of text as well and more often than not that is exactly what they are used for. Of course you can also use a list to contain any data type as long as they are separated with a comma.

```
test_list = ['Hello', 50]
```

Here we have another example, but contained within are two values, one a string and the other an integer. From this we can see a list can hold more than one data type. Python is a truly fantastic language and has made breaking down and storing value in a list easy, which in turn will make our indexing and slicing in the next two lessons much easier.

Strings

Strings are a common data type. Strings are ordered sequences, it is a data type of text, meaning all values within, be they numbers or letters, are text or string. Strings have been taught in basic lessons, but there are some points that need to analyzed with strings.

First, when we begin working with slicing and indexing it is important to note that there is a number system when working with strings. Every character within a string, including white space, has a number value assigned to each which will be used directly with indexing and slicing.

For example, take note of this string here

'I am John'

It contains three words. I am John. Each character in this string has a value assigned from 0 to 8. The first character, 'I', has the value of 0. The whitespace that follows has the value of 1 and so on until 'n' which has the value of 8.

The second thing to note about strings is they can be encapsulated in either single or double quotations. In single quotations you cannot begin or end your text with whitespace or include the single quotation character anywhere either.

However, in double quotations you can include these characters anywhere you wish. It is common enough for programmers in Python to use single quotations, however, when partaking in data retrieval it is sometimes better to use double quotations, especially with user input.

In our practical string lesson we will analyze all these factors and take a small peak at indexing and slicing. Don't worry too much if it seems complicated at first. We will do a discussion and practical lesson with indexing and slicing soon after the practical lesson.

Slicing and Indexing

Slicing and indexing is used to retrieve characters or sets of characters from a string or list. For example, supposing you want to retrieve a character from a string such as this.

```
name = 'John'
```

As you can see there are four characters in the name, numbered 0 to 3. Let's say we want to retrieve the 'o' from John. To do so we simple type the name of the variable, in this case name, follow it with a set of squared brackets and type in the number place of 'o'.

```
name[1]
>> 'o'
```

What this will then do is return the single character 'o' in the output. Of course you can even work backwards when indexing. By using negative numbers you can work from the end of a string towards the start. For example, working on the string we have created we type in -1 to get the last character.

```
name[-1]
>> 'n'
```

This is a fantastic way to retrieve the last letter of a string without knowledge of the length of the string. Of course this can be continued, typing in -2 will get you 'h', -3 will retrieve 'o' and so on.

Now, let us discuss slicing. Slicing is used to retrieve a section of a string, not just a single character. Let us take a look at the following string here.

```
greet = 'My name is John'
```

From this string we can cut out specific parts of the string, returning what it left. This is known as slicing. Of course, slicing the string does not replace the value of the variable, so don't worry, your data will not be overwritten, only the output and value returned will be adjusted.

Slicing is done using the same number system. Suppose from the greet string we want to output everything from ‘name’ to ‘John’. To do so would type the following.

```
greet[3:]
```

What this will do is find the character with the number place of three and the colon lets the program know that everything from the character to the end should be outputted. However, there is a difference that needs to be noted when taking a specific section of characters out of the string.

Here we have a line of code:

```
greet[:7]
>>‘My name’
```

You will notice we have the colon before the number, what this does is return everything from the start of the string to the end of name. What is important to note is that we have the value 7. 7, according to the system, is the place of the white space that follows name. So why is it not printed with the string? When working on the end of the slice you should note that the second value is non-inclusive. If we assumed that the second value worked the same as the first and used 6 instead of 7 you will find the string ‘My nam’ is returned instead.

Finally, let’s take a look at the final lesson on slicing. The step size. The step size is the characters chosen in their order with each slice. For example, suppose I want to only print every second value in my string.

First, I will create a suitable string for you to compare with.

```
num_word = ‘123456789’
```

Suppose I only want to output the odd numbers in this string. What I would do is adjust the step size from 1 to 2.

```
num_word[::2]
>>‘13579’
```

If I wanted to output every even number.

```
num_word[1::2]
>>‘2468’
```

Finally, if I wanted to output every even number up to 6.

```
num_word[1:7:2]
>>‘246’
```

Formatting

Formatting with strings is a special way in which you can append an output string value. For example, suppose you want to output a certain line of text, but you have a blank space that needs to be filled. What you would then do is make use the .format method.

How the format method works is you have your string, but where you would like to insert a string value you would place a set of {}. Once you have done this within your string you add the method to the end of said string. For your method you simply type .format()

Within in that set of normal brackets you place your string or variable which you want to insert. Here is an example:

```
name = 'John'
print('My name is {}'.format(name))

>> 'My name is John'
```

As you can see we have created a variable with the string value of John. We then create a place within the print to insert 'John'. The output is below, printing, 'My name is John', not 'My name is {}'

Now, let's take a look at numbered formatting. Supposing you want to insert multiple values, but want to ensure the order as well. Perhaps the values you attain are not in order; you will need to number your inserts as follows.

```
print('one {1} three {0} five'.format('four', 'two'))
>> 'one two three four five'
```

It is as simple as that. Depending on the order in the formatting you can place values in specific parts of the string. Of course, if your values are in order as they are in the string, you can leave the inside of the {} empty. The benefit of using numbered formatting is that you can reuse values. If you wanted to you could have had both {} contain 0 and four would be printed twice. Formatting is all about adjusting a string with the values at hand and you can be sure that there are methods to which you can place these values.

Another way you can use formatting is to help define a float value. For example, supposing you want to insert a float value but it has too many values after the decimal point. Well, with formatting you can solve this problem.

{value:width.precision f}

Value is the name of the variable, width is the number of spaces between the value and the string and precision is the number of places you want after the decimal point.

num = 5.5555555
print(“Num is {r:1.3f}”.format(r=num))
>> ‘Num is 5.556’

Dictionary

Dictionaries are similar to lists in that they are variables that can contain multiple values. However, that is the only aspect in which dictionaries and lists are similar. Lists can be sorted, indexed and slices because they are ordered sequences, however, you cannot do this with dictionaries.

Instead, dictionaries are unordered, used to contain data where the programmer and user does not know where the value is stored. So how do we retrieve data from the dictionary? We use a key. When creating a dictionary you type the name, equals and follow it with curled brackets. When storing a value in these curled brackets you must note that a key is necessary, a key being a line of text. You can make these single characters or a specific sequence of characters, it is up to you.

Here is an example of a dictionary with two values in. Note, the keys and values are separated with colon, these are paired. When separating these pairs of keys and values we use a comma.

```
test_dict = {'fName': 'Joe', 'sName': 'Blogs'}
test_dict['sName']
>> 'Blogs'
```

In the example I have create a dictionary called test_dict and given it two vales. Each value is assigned a suitable key, fName and sName. To call a value I type in the name of the dictionary and open a set of squared brackets. Within these squared brackets I use one of the keys, sName. The dictionary is then searched and the value returned.

It should also be noted that dictionaries can contain many different types of values, such as lists or even other dictionaries. As with lists you can store a value within a dictionary and the value can be called using an additional set of squared brackets. The same applies with a dictionary within a dictionary.

```
test_dict = {'a': {'b': 10, 'c': [ 5 , 15]}, 'd': 20}
test_dict['a']['c'][1]
>> 15
```

From this example we can see that the second line of code opens the first key in the dictionary, which leads to another dictionary. The second set of brackets opens the dictionary and selects the list. The third set of squared brackets selects the second value in the list, which is 15.

This is just one of the examples one can study when looking at dictionaries. Dictionaries a complicated when working to such a degree, but with experience one can master as easy as any other variable type. Be sure to note that dictionaries are not ordered, meaning you will most likely use them to store large amounts of data with assigned values that can be called upon. Most of the time dictionaries are used to contain information on stock in online stores and the key being the product key.

Tuples

Tuples are another form of variable which can contain many values. Similar to lists in this way, but unlike lists these values are immutable. The values within tuples cannot be replaced.

Tuples can contain values of any type be they string or integer or so on. To create a tuple is as follows.

```
test_tup = (‘Hello’, 50, 6.78)
```

As you can see, unlike lists and dictionaries a tuple makes use of () instead of {} or [] brackets. These values can be indexed as well by using a set of squared brackets as follows.

```
test_tup[1]
>> 50
```

Tuples also have their own methods which you can use to check the data within, count and index. Count is used to count how many times a certain value appears in the tuple, index is used to find the nearest value and return the place in the tuple. Here is an example of a count method.

```
test_tup_two = (‘one’, ‘two’, ‘one’)
test_tup_two.count(‘one’)
>> 2
```

Here is an example of how to use the index method.

```
test_tup_two.index(‘one’)
>> 0
```

You will notice that one appears twice in the tuple but the method returns the place of the first value that matches what the programmer is looking for. Tuples are mainly used when you start working with more advanced programs. The reason you would use a tuple instead of a list is because it is easy to replace a value in a list by accident that it is when you try to replace a value in a tuple. To avoid accidently ruining a program, advanced programmers make use of tuples; an error will be thrown if the programmer tries to replace the value in the tuple, unlike the list. An error with a tuple is much better than an error with the entire program.

Sets

Sets are variables that are used to contain unique values. As such sets cannot contain the same value more than once. You can have two or more integers, but you cannot have two integers of the same value. Sets are basic variable types if you have a look at the variables we have created so far and are more commonly used in stock-keep programs as you don't want to add two versions of the same stock, because it may lead to an error in the programming later on.

To create a set variable is as follows:

```
test_set = set()
test_set.add(50)
test_set.add(55)

test_set
>>{50, 55}
```

As you can see from this example I have created a set with two values, 50 and 55. The set variable type has a method in which you the programmer or the user can add values to the set. To do this you simply type in the following:

```
test_set.add(60)
```

What this line of code will do is add the value of 60 to the set. However, if you were to add 60 again or the other two integer values you will notice that they don't add. As explained, a set can only contain unique values meaning not more than one of the same type and value. With this in mind, the add method will not add the value to the set if the set already contains the value.

There is also a fantastic use to the set variable type to go through a list and remove any duplicates. Meaning, if you were to create a list with many duplicate values you could then cast the list into a set and the set will contain only one of the duplicate values.

Sets in this way can be very useful to you as a programmer who wants to avoid duplicate data. As all programmers know having duplicate data is future problem waiting to happen so it is best to make use of the tools at hand to avoid it.

Methods and Functions

Methods, we have used many to alter, create and process data. For example, we have used most recently the add method to add a value to a set. Most methods we do not use that often, for example, when we create a variable and decide to use a method with it later we could simply add '.' to the end of the variable name and press tab. Doing so will present you with a list of methods that you can experiment and go through. The methods I have shown you so far in this course are the most used methods and depending on your program, the most useful as well.

If you want to learn what a method does in technical terms you can use code within Python to help you. Simply type:

help(a.b)

Replacing 'a' with a variable and 'b' with a method and running the line of code should present you with a short documentation on the method and what it does. Of course for more information one can also take a look at the Python documentation online which offer you broad example as well as specific ones. However, I can highly recommend experimentation for anyone unfamiliar with a method. Most methods are self-explanatory, but some do require explanation.

Now, functions. Functions are sections of code that can be repeated. In other words, these blocks can be called upon without having to type in the code again. Knowing how to create and use functions is a key part of Object Orientated Programming (OOP), and OOP is key knowledge for any effective programmer. If you wish to learn this advanced skill one must begin with creating functions.

To create a function you simply type:

```
def function_name():
    print("Hello")
```

In this code you see I have created a function by typing in def and the name of the function followed by a ():

When you type in function_name() again and run the code you will find that it outputs 'Hello'. A simple function that can be called upon again later in the in the program. One of the simplest examples of a function can often be the most useful; however we can add to it further.

```
def my_name_is(name):
    print(“My name is “ + name)
```

A method that outputs a meaningful string which can be re-used later in the program. To make use of this code all you need to do is include a string between the () when you write the line of code.

```
my_name_is(“Joe”)
>> ‘My name is Joe’
```

It is as simple as that. Creating functions soon becomes the bread and butter of any programmer, but of course, creating an advanced program with functions requires practice. Soon you will be creating multiple functions that all work hand-in-hand. Many of the advanced programs you will find have more code running in the background than on the surface, this is because of functions, the core of OOP.

Conclusion

And that concluded this workbook. I hope you found the course informative and it helped to further your knowledge of Python programming. Python is not a simple language, but if you break things down to their simplest nature it doesn't take much to build it back up and create more advanced programs with ease. If you wish to find more tutorials and workbooks feel free to visit out site, www.programmingtut.com.

At our site you receive news, articles, advice, lessons, tutorials, courses and so much more content with plenty of free bonuses. I hope to see you soon. Good luck with your further studies, programmer!