Python Crash Course

Python - Programming, Computer Science, Algorithms and Data Structures

Part 1, version 1.1

by Mashrur Hossain

Table of contents

Command line Basics

The commands below will allow you to work through the content in this course without any difficulty.

To see which directory you are currently in:

Windows-> `cd`

Mac/Linux-> `pwd`

To list files and folder under current directory:

Windows-> `dir`

Mac/Linux-> `ls`

To move to a directory listed under current directory:

Windows, Mac/Linux-> `cd name_of_directory`

To move up one directory from current directory:

Windows, Mac/Linux-> `cd ..`

To create a new directory under the current directory:

Windows, Mac/Linux-> `mkdir name_of_directory`

To clear the terminal/command prompt screen:

Windows-> `cls`

Mac/Linux-> `clear`

Back to table of contents

Strings

Official Python documentation on strings can be found here:
https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Working with text in Python is accomplished by using strings. To create a string you can wrap the text you want to create/use with either single, double or triple quotes. The following are all valid strings.

```
'Hello world using single quotes'
"Hello world using double quotes"
"""Hello world using
triple quotes, also known as
multi-line strings"""
```

To type a comment in Python you can use the # sign. Anything written after the # in a python file will be ignored by the Python interpreter. Example below.

```
# This is a comment
```

To print an object to the screen, you can use the print function.

```
print('Hello world') # This will print Hello world
```

To run the program that contains the code above, first make sure the file ends with an extension of .py Second, make sure you are in the same directory in your terminal as where the file is saved. So let's say you have saved the code above in a file named lecture_1.py and you are in the same directory in your terminal where this file is saved. You can then enter the following command in your terminal to run the program. (Ignore the $ in the beginning of the line, that signifies the command prompt)

```
$ python lecture_1.py
```

Running the command above will result in the following output to your terminal screen:

```
$ python lecture_1.py
Hello world
```

To use an apostrophe ' in your print statement, you can wrap your string in double quotes.

```
print("I'm using an apostrophe in this statement")
```

To use quotes within your print statement, you can wrap your string in single quotes like below.

```
print('This is a "quote from a book" which I like')
```

Variables are used to reference objects in Python, you can think of variables as memory addresses. They should be named using snake case (lower case and words separated by underscore). Example of snake case -> my_name.

```
message = "Hello! welcome to the Algorithm's course"
name = "Mashrur"
```

You can use variables in your print function to print to the screen

```
message = "Hello! welcome to the Algorithm's course"
name = "Mashrur"
print(message)
print(name)
```

You can print multiple strings using the print function separating them by using commas in-between. A comma will automatically add a whitespace between the strings.

```
print(message, name)
```

Running the code above will result in the following output to your terminal screen:

```
$ python lecture_1.py
Hello! welcome to the Algorithm's course Mashrur
```

String concatenation

You can use string concatenation and add multiple strings together using the + operator

```
message = "Welcome to the course"
name = "Mashrur"
print(message + name)
```

To add an empty space in between the two strings, which is not automatically added, you can concatenate an empty whitespace string in the middle of the two strings like below.

```
print(message + " " + name)
```

Running the code above will result in the following output to your terminal screen:

```
$ python lecture_1.py
Welcome to the course Mashrur
```

Indexing

Strings are sequences of characters which are indexed. We can index into a string by using square brackets. Try out the examples below in your editor.

```
movie_name = "Interstellar"
print(movie_name[0]) # This will print 'I' which is at index 0
print(movie_name[1]) # This will print 'n' which is at index 1
print(movie_name[11]) # This will print 'r'
print(movie_name[-1]) # This will also print 'r', last item
print(movie_name[-2]) # This will print 'a', reverse index
```

You can also test this code out directly in the Python console or Python interactive shell which is provided by every python installation. To start the console you can type in python from your command line. Mine looks like below.

```
$ python
Python 3.6.0 |Anaconda custom (x86_64)| (default, Dec 23 2016, 13:19:00)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note: If you have both python 2 and 3 installed, you may need to specify python3 to start the console using Python 3

You can directly type Python code in this console, and you won't need the print function to display the output. Example below.

```
>>> movie_name = "Interstellar"
>>> movie_name[0]
'I'
>>> print(movie_name[0])
I
>>>
```

To exit the console at any point you can type exit() like below.

```
>>> exit()
$
```

Strings are immutable

You cannot change the value of a specific string once it is created. For example, say you have the string my_name = "joan". If you wanted to change the character 'a' at index 2 to an 'h', essentially changing "joan" to "john", you cannot use code like.

```
my_name = "joan"
my_name[2] = "h" # This line will result in an error
print(my_name)
```

If you ran the code above, you would get the following output.

```
$ python lecture_1.py
Traceback (most recent call last):
  File "lecture_1.py", line 2, in <module>
    my_name[2] = "h"
TypeError: 'str' object does not support item assignment
$
```

The line *"TypeError: 'str' object does not support item assignment"* is letting you know that strings are immutable and their value cannot be changed. To make the change above you have to create a new string object and assign it like below (You can use the same variable name, but it will be a different string object).

```
my_name = "joan"
print(my_name)
my_name = "john" # This will create a new my_name string object
print(my_name)
```

If you ran the code above, you would get the output you were looking for.

```
$ python testing_practice.py
joan
john
```

Slicing

You can use slicing to select portions of the string or substrings to work with. For example, if we used slicing notation on our movie_name variable defined above, it would look like below.

```
print(movie_name[0:5])  # This will print 'Inter'
print(movie_name[:5])   # We can omit the starting index 0, if we
                        # want to start at the beginning to get
                        # 'Inter' as well
print(movie_name[5:])   # This will print 'stellar', begin at
                        # index 5 and go to the end
print(movie_name[:])    # This will print 'Interstellar', begin
                        # to end
print(movie_name[5:9])  # This will print 'stel', index 5 to 8,
                        # excluding the stop index of 9
```

You can specify the step size as an optional third argument like below.

```
print(movie_name[5:9:2])    # This will print 'se', moving forward
                            # by 2 steps instead of the default 1
print(movie_name[::2])      # This will print "Itrtla"
print(movie_name[::-1])     # This will reverse the string and
                            # print 'ralletsretnI'
```

Methods and functions

Python provides built-in methods and functions which we can run on string objects based on our needs. You can find more information on built-in functions by the python standard library here: https://docs.python.org/3/library

Try out some built-in functions on strings like below, see if you get the output you were expecting.

```
message = "Welcome to the course"
print(len(message)) # This gives us the number of characters
                    # in the string
print(type(message)) # This gives us the type of object
                    # message is, string in this case
print(id(message))  # This gives us an integer representation
                    # of the message variable in memory
```

Running the code above would result in the following output in the terminal.

```
$ python lecture_1.py
21
<class 'str'>
4530028168
$
```

You can also run built-in methods on string objects provided by python. Methods differ from functions in that methods run on specific objects (a specific string here) on which they are called. They are associated with that object type. We will explore these differences in much more detail as we progress through the course.

For now, think of the methods below as specific to string objects and to use them you can 'chain' them to the string using the dot or '.' notation. Try out the code below to see what each method does to the string you run it on.

```
message = "Welcome to the course"
print(message.capitalize())
print(message.upper())
print(message.lower())
```

Running the code above will result in the following output.

```
$ python lecture_1.py
Welcome to the algorithms course
WELCOME TO THE ALGORITHMS COURSE
welcome to the algorithms course
$
```

You can use the split() method to separate each word in a string and return a list of words. By default the split will happen on whitespace. You can also specify what to split on like below.

```
message = "Welcome to the Algorithms course"
print("Initial message:", message)
print(message.split())
print(message.split(" ")) # same output as above line

message = "Welcome-to-the-Algorithms-course"
print("Updated message:", message)
print(message.split("-")) # specifying to split on - instead of
                          # the default whitespace
```

Running the code above will result the following output.

```
$ python lecture_1.py
Initial message: Welcome to the Algorithms course
['Welcome', 'to', 'the', 'Algorithms', 'course']
['Welcome', 'to', 'the', 'Algorithms', 'course']
Updated message: Welcome-to-the-Algorithms-course
['Welcome', 'to', 'the', 'Algorithms', 'course']
$
```

You can convert the items from the resulting list back to a string by using the join() method. Example below.

```
message = "Welcome-to-the-Algorithms-course"
print("Initial message:", message)
message_list = message.split("-")
print("Converted to a list:", message_list)
new_message = " ".join(message_list)
print("Converted back to string:", new_message)
```

Running the code above will result in the following output.

```
$ python lecture_1.py
Initial message: Welcome-to-the-Algorithms-course
Converted to a list: ['Welcome', 'to', 'the', 'Algorithms', 'course']
Converted back to string: Welcome to the Algorithms course
$
```

You can import more features and functionality available to strings by importing the string module in your code. Documentation can be found here: https://docs.python.org/3.7/library/string.html

If you want to import the entire string module you can use the following code on top of your python script.

```
import string
```

If you want to import a specific feature, let's take the ascii_lowercase constant as an example, then you can do so like below.

```
from string import ascii_lowercase
```

You can then use it in your code.

```
from string import ascii_lowercase
print("Lowercase alphabet:", ascii_lowercase)
```

Running the code above will result in the following output.

```
$ python lecture_1.py
Lowercase alphabet: abcdefghijklmnopqrstuvwxyz
```

This concludes our introductory look at strings. We will continue to learn more about them in upcoming chapters.

Back to table of contents

Print formatting

There are multiple ways you can format how text or objects are displayed by your print statements. We will explore 3 of them in this chapter.

Take the example below where we have a variable stock_price defined to be the 1100 (this 1100 is a number, an integer specifically, we will explore them more in the next chapter). One way to print this would be to use a comma after the first string within the print statement which we have seen already.

```
stock_price = 1100
print("Price of google stock is:", stock_price)
```

You can also use the format() method or f'strings. Let's look at format() first.

```
stock_price = 1100
print("Price of google stock is: {}".format(stock_price))
```

Python simply takes the object listed within the parenthesis of the format method call and places it within the {} of the string.

You can do the same using f'strings.

```
stock_price = 1100
print(f"Price of google stock is: {stock_price}")
```

If all 3 methods above were placed one after the other and we ran such a script it would give us the following output in the terminal.

```
$ python lecture_2.py
Price of google stock is: 1100
Price of google stock is: 1100
Price of google stock is: 1100
```

You can use multiple variables as well.

```
today_price = 1100
yesterday_price = 1000
print("Today's price:", today_price, "yesterday's price:", yesterday_price)
print("Today's price: {}, yesterday's price: {}".format(today_price, yesterday_price
print(f"Today's price: {today_price}, yesterday's price: {yesterday_price}")
```

Output of the program above would be like below.

```
$ python lecture_2.py
Today's price: 1100 yesterday's price: 1000
Today's price: 1100, yesterday's price: 1000
Today's price: 1100, yesterday's price: 1000
$
```

Special characters

You can use special characters within the quotation marks of strings. These characters have special meaning. 3 such characters are listed below.

\ - is an escape character, it escapes the special character following it in a string

\n - species a new line within the string

\t - adds a tab in it's place within the string

Take the line of code below as an example.

```
print("My name is jon snow and not only do I know nothing but I also do nothing")
```

This statement can be broken up into multiple lines with \

```
print("My name is jon snow and \
not only do I know nothing but \
I also do nothing")
```

If you run the code above it will display the line as a single line. Python simply 'escapes' the implied new lines at the end of each line because of the \ character that was used.

\n can be used to print it in multiple lines.

```
print("My name is jon snow\nI know nothing\nI also do nothing")
```

You can run the code above to get the output below.

```
$ python lecture_2.py
My name is jon snow
I know nothing
I also do nothing
```

\t can be used to introduce tabs within the lines

```
print("My name is jon snow\n \tI know nothing\n\t\tI also do nothing")
```

This concludes our look at print formatting and special characters. We will learn more about and practice using them in upcoming chapters.

Back to table of contents

Numbers

Numbers in Python (standard library) are represented using integers, floats and complex numbers. In this course we will cover integers and floats which are most common. Documentation on integers and floats can be found here: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Integers are whole numbers, no decimals (example 1, 2, 10, 500). Floats on the other hand include decimals or digits after the decimal. To check the type of an object (this will work with other types beyond integers and floats), you can use the type function, example below.

```
num_1 = 10
print(type(num_1)) # This will return class 'int'
```

You can perform standard math operations using `+`, `-`, `*`, `/` operators

```
print(4+4) # output 8
print(10-5) # output 5
print(10*2) # output 20
print(10/2) # output 5.0
```

Division will give you a float (decimal value number) by default. To perform floor division, or remove the decimal you can use `//`

```
print(10//3) # output 3
```

You can use the mod operator `%` to find the remainder.

```
print(10%3) # output 1
print(10%2) # output 0
```

To take one number to the power of another number, you can use the `**` operator

```
print(5**2) # output 25 (this is 5 squared)
print(5**3) # output 125 (this is 5 cubed)
```

You can import the math module and use functions provided by it. Documentation on the math module can be found here: https://docs.python.org/3/library/math.html

```
import math
print(math.pow(10,5)) # output 100000.0, this is 10 to the power 5
```

Similarly, you can import the random module and generate random integers using the randint function. More information on the random module can be found here: https://docs.python.org/3/library/random.html

```
import random
print(random.randint(0,1000))   # output random integer between 0
                                # and 1000
```

You can use other functions on numbers as well. One such function is the abs function which returns the absolute value of a number (removes the negative sign)

```
num_2 = -10
print(abs(num_2))   # output 10
```

Type casting

You can convert one data type to another using type casting. To convert the string “10” to the integer 10, you can cast it to an int like below.

```
my_string = "10"
my_num = int(my_string) # this will convert "10" to int 10
print(type(my_num)) # output will be class 'int'
```

Similarly you can convert an int to a string, 10 to “10” by casting it to a string (str)

```
my_num = 10
my_new_string = str(my_num)     # this will convert 10 to "10"
print(type(my_new_string))      # output will be class 'str'
```

You can get input from the user of the program using the input() function. This is displayed in the video lecture on numbers where a multiplication program is used. The input received from the user is always in string format. Therefore, if you want to use that input to perform math operations, you have to cast it to an int or float first. The multiplication program code is shown below.

```
print("Welcome to the multiplication program")
print("-"*30)
num_1 = int(input("Enter a number to multiply-> "))
num_2 = int(input("Enter a second number to multiply-> "))
result = num_1 * num_2
print(result)
```

This concludes our introductory look at working with numbers. We will learn more about them as we continue to work with them in later chapters.

Back to table of contents

Branching and control flow using if/elif/else

You can control the execution flow of your program using if/elif and else blocks. This is known as branching. The if and elif expressions use conditional operators which evaluate to `True` or `False`.

Documentation on conditional operators can be found here:
https://docs.python.org/3/library/stdtypes.html#comparisons

Branching is best explained with examples, some are below.

```
choice = '1'
if choice == '1':
    print('You have chosen option 1')   # make note of the indentation
                                        # of this line
elif choice == '2':
    print('You have chosen option 2')   # make note of the indentation
                                        # of this line as well
else:
    print('You have made an invalid choice')
```

if/elif/else blocks take the form of the condition followed by a `:` and then the code you want executed if that condition is true underneath it. This block of code underneath the condition needs to be indented in by 1 tab in order to be considered as part of the code block associated with the condition.

Note: elif and else are optional based on your program's requirements. You can just have an if condition without either an elif condition or an else. Or you can have an if/else condition without an elif. You can also have multiple elif conditions if you are testing for multiple conditions before getting to the else condition. In the example above if you wanted to see if the choice was 3, you could've simply added an additional elif condition.

Some more examples below. First of a boolean test using the `not` operator.

```
made_payment = True
a = 'Please pay monthly premium'
b = 'Welcome to your homepage'

if not made_payment:
    print(a)
else:
    print(b)
```

Second, you can combine comparison operators and conditional tests.

```
i = 20
j = 10
k = 30

if i < j and i < k:
    print("i is less than j and k")
elif i == j and i == k:
    print("i is equal to j and k")
else:
    print("i is greater than j")
```

You can also compress and simplify simple if/else blocks using ternary operators. Below is an example of using ternary operators to compress an if/else block used earlier.

```
made_payment = True
a = 'Please pay monthly premium'
b = 'Welcome to your homepage'

print(a) if not made_payment else print(b)
```

The last line in the code above can also be written as below.

```
print(b) if made_payment else print(a)
```

You can nest if/elif/else blocks within other if/elif/else blocks. Below we have an example of the multiplication program we initially saw in the numbers chapter. Here it has been modified to perform either multiplication or division based on user input and a nested if condition within it.

```
print("Welcome to the calc program")
print("-"*30)
choice = input("Choose 1 to multiply, 2 to divide-> ")
if choice == "1" or choice =="2":
    num_1 = int(input("Enter first number-> "))
    num_2 = int(input("Enter second number-> "))
    if choice == "1":
        print(f"{num_1} multiplied by {num_2} is: {num_1*num_2}")
    else:
        print(f"{num_1} divided by {num_2} is: {num_1/num_2}")
else:
    print("You've made an invalid selection")
```

This concludes our introductory look at branching with if/elif/else conditions. We will see a lot of examples and usage of branching as it's one of the staples of programming in future chapters.

Back to table of contents

Lists

Lists are compound data structures. They are collections of objects, mutable, they maintain order and are indexed. You can find additional documentation on lists here: https://docs.python.org/3/library/stdtypes.html#list

Lists begin with an open square bracket `[` and close with a closing square bracket ']' and each element/object contained within the list is separated by commas. These objects can be other compound data structures like lists as well.

An example list is displayed below.

```
my_list = [1,2,3,4,5,6,7,'hello','world']
print(my_list)
```

You can run functions on lists like len(), min(), max(). Examples below.

```
my_list = [15, 6, 7, 8, 35, 12, 14, 4, 10, 15]
print(len(my_list)) # will return 10, number of integers in list
print(min(my_list)) # will return 4, minimum value in list
print(max(my_list)) # will return 35, maximum value in list
```

You can add objects to the list using the append() method. Append will add the object to the back of the list. Example below

```
my_strings_list = ["comp sci", "physics", "elec engr", "philosophy"]
print(my_strings_list)
my_strings_list.append("art")
print(my_strings_list)
```

If you run the code above, you'll get output like below in your terminal.

```
$ python testing_practice.py
['comp sci', 'physics', 'elec engr', 'philosophy']
['comp sci', 'physics', 'elec engr', 'philosophy', 'art']
```

If you use the append() method to add another list to your list, then the new list will show up in list form within your list. Example below.

```
my_strings_list = ["comp sci", "physics", "elec engr", "philosophy"]
print(my_strings_list)
my_new_list = ["art", "econ"]
my_strings_list.append(my_new_list)
print(my_strings_list)
```

The code above will result in the output below.

```
$ python testing_practice.py
['comp sci', 'physics', 'elec engr', 'philosophy']
['comp sci', 'physics', 'elec engr', 'philosophy', ['art', 'econ']]
```

If you wanted to just add the elements in the new list to your list instead of as a list itself, then you can use the extend method.

```
my_strings_list = ["comp sci", "physics", "elec engr", "philosophy"]
print(my_strings_list)
my_new_list = ["art", "econ"]
my_strings_list.extend(my_new_list)
print(my_strings_list)
```

The code above will result in the output below.

```
$ python testing_practice.py
['comp sci', 'physics', 'elec engr', 'philosophy']
['comp sci', 'physics', 'elec engr', 'philosophy', 'art', 'econ']
```

You can use indexing and slicing with lists just like with strings. Some examples below.

```
my_list = [15, 6, 7, 8, 35, 12, 14, 4, 10, 15]
print(my_list[0])   # this will return 15
print(my_list[5])   # this will return 12
print(my_list[4:])  # this will return a sublist from index 4 to the end
print(my_list[1:5]) # this will return a sublist from index 1 to index 4
                    # note: slice notation does not include the item at
                    # the stopping index provided
print(my_list[-1])  # this will return the last element of the list
print(my_list[:])   # this will return a copy of the entire list
print(my_list[::-1])    # this will return a reversed copy of the list
print(sorted(my_list))  # this will return a sorted copy of the list
print(my_list)      # original list un-altered, order is still the same
my_list.sort()      # this will sort the list in-place
print(my_list)      # now the list is in sorted order, original list order
                    # has been changed by the sort() method
```

Since lists are mutable you are able to change elements within the list without needing to re-assign to a new list. An example of this is displayed in the code block above when the sort() method sorts the same list (my_list) in place. The order of the elements in the list are altered by this method.

You can also use index notation to change elements in indices that you want, example below.

```
my_strings_list = ["comp sci", "physics", "elec engr", "philosophy"]
my_strings_list[1] = "art" # this will place "art" in place of "physics"
print(my_strings_list)
```

The code above will result in the output below.

```
$ python testing_practice.py
['comp sci', 'art', 'elec engr', 'philosophy']
```

This concludes our introductory look at lists. We will work with lists a lot in the course and while programming with Python in general. A large portion of the algorithms section of the course is done using lists.

Back to table of contents

Dictionaries

Dictionaries are compound data types which are collections of key value pairs. An example dictionary is displayed below.

```
my_dictionary = {'name':'john doe','course':'python'}
```

Documentation on dictionaries can be found here: https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

To access a value associated with a key in a dictionary you will need to provide the key associated with the value. For example in the example dictionary if you wanted the course name from the dictionary, you can use the code below.

```
my_dictionary = {'name':'john doe','course':'python'}
print(my_dictionary['course']) # this will print python
```

Dictionaries can take various forms, all 3 below are valid dictionaries.

```
my_dictionary = {'name': 'mashrur', 'course': 'python', 'fav_food': 'ice cream'}
phone_dict = {'mashrur': '555-55-5555',
              'zoolander': '999-99-9999',
              'jon_snow': 'fail-o-so-bad'}
word_dict = {'a':
                {
                 'apple': 'the round fruit of a tree of the rose family',
                 'ant': 'an insect which cleans up the floor'
                },
             'b':
                {
                 'bad': 'of poor quaity or low standard',
                 'business': 'season 8 of GOT'
                }
            }
print(my_dictionary)
print(phone_dict)
print(word_dict)
```

You can have other data types and objects as the values in dictionaries. It is good practice to keep the keys as immutable data types like strings and integers so they cannot be altered.

You can re-assign the value associated with a key by using the key. For example in if you wanted to change the name from 'mashrur' to 'john' in my_dictionary above, you can use the code below.

```
my_dictionary['name'] = 'john' # this will re-assign 'mashrur' to 'john'
print(my_dictionary)
```

You can add new key, value pairs to the dictionary with the same notation. Example below.

```
my_dictionary['job'] = 'python programmer'
print(my_dictionary)
```

If you run the code above, the new key 'job' and value 'python programmer' would be added to the dictionary like below.

```
$ python lecture_dicts.py
{'name': 'john', 'course': 'python', 'fav_food': 'ice cream', 'job': 'python prograr
```

You can go deeper into dictionaries by providing keys followed by other keys in cases where you have dictionaries within dictionaries. Take the word_dict below as an example. To access the value associated with business, you can use the following code `word_dict['b']['business']` like below.

```
word_dict = {'a':
                {
                 'apple': 'the round fruit of a tree of the rose family',
                 'ant': 'an insect which cleans up the floor'
                },
             'b':
                {
                 'bad': 'of poor quaity or low standard',
                 'business': 'season 8 of GOT'
                }
            }

print(word_dict['b']['business']) # this will print season 8 of GOT
```

Some methods you can use to get keys, values and key, value pairs as an iterable are shown below.

```
my_dictionary = {'name': 'john', 'course': 'python', 'fav_food': 'ice cream', 'job':

print(my_dictionary.keys()) # this will print all the keys in the dictionary
print(my_dictionary.values()) # this will print all the values
print(my_dictionary.items()) # this will print all the key, value pairs
```

If you run the code above, you will get the output below.

```
$ python lecture_dicts.py
dict_keys(['name', 'course', 'fav_food', 'job'])
dict_values(['john', 'python', 'ice cream', 'python programmer'])
dict_items([('name', 'john'), ('course', 'python'), ('fav_food', 'ice cream'), ('job
```

The items() method returns an iterable which you can iterate through to get the key, value combos one by one. To iterate through the dictionary using items() you can use a simply for loop like below.

```
my_dictionary = {'name': 'john', 'course': 'python', 'fav_food': 'ice cream', 'job':
for key, value in my_dictionary.items():
    print(f'Key: {key}, Value: {value}')
```

Running the code above will give you the following output.

```
$ python lecture_dicts.py
Key: name, Value: john
Key: course, Value: python
Key: fav_food, Value: ice cream
Key: job, Value: python programmer
```

This concludes our introductory look at Dictionaries. Dictionaries are a staple of Python and used a lot. In the data structures section of the course we will build our own dictionary data structure from scratch.

Back to table of contents

Sets

Sets are un-ordered collections of objects which don't allow duplicates. An example set is shown below.

```
my_set = {1,6,2,'java','ruby',8,9,10,21,1000,'python'}
```

Since sets don't allow duplicated, if you tried to create the same set above with an additional duplicate 6, it would simply discard the duplicate. Example below.

```
my_set = {1,6,2,'java','ruby',8,9,10,21,1000,'python',6}
print(my_set) # duplicate 6 will be discarded
```

Documentation on sets can be found here: https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Sets are useful in a lot of ways. One use can be if you have a list with duplicates and you wanted to get rid of the duplicates. You can cast it to a set to get rid of duplicates.

```
my_list = [1,6,2,'java','ruby',8,9,10,21,1000,6,'python','java']
print(my_list)
my_set = set(my_list)
print(my_set)
# you can cast my_set back to a list if you wanted a list as output
```

You can find information in sets using 'in'. Example below.

```
my_set = {1,6,2,'java','ruby',8,9,10,21,1000,'python'}
print('java' in my_set) # this will return True
print('javascript' in my_set) # this will return False
```

Sets are optimized to perform mathematical operations like finding unions, intersections, differences etc. Some examples using these methods are shown below.

```
my_set = {1,6,2,'java','ruby',8,9,10,21,1000,'python'}
programming_set = {'java','ruby','javascript','python','c'}
print(my_set.intersection(programming_set)) # elements in common in both sets
print(my_set.union(programming_set)) # all elements in both, no duplicates
print(my_set.difference(programming_set)) # in my_set, not in other
print(programming_set.difference(my_set)) # in programming_set, not in my_set
```

If you run the code above, it will give you the output below.

```
$ python lecture_sets.py
{'python', 'ruby', 'java'}
{1, 2, 6, 8, 9, 10, 'python', 'c', 21, 'javascript', 'ruby', 1000, 'java'}
{1, 2, 6, 8, 9, 10, 1000, 21}
{'javascript', 'c'}
```

You can iterate through items in a set using a simple for loop, example below.

```
my_set = {1,6,2,'java','ruby',8,9,10,21,1000,'python'}
for item in my_set:
    print(item)
```

This concludes our introductory look at Sets. We will look at Tuples in the next chapter.

Back to table of contents

Tuples

Tuples are immutable collections of objects which are indexed, which means they maintain order. Tuples behave a lot like lists except for the key fact that they are immutable unlike lists. So you cannot update an object inside a tuple, you have to use a new tuple. A tuple may look like the example below.

```
my_tuple = ('hello','world','night','king','says','bye',8,3)
```

Documentation on tuples can be found here: https://docs.python.org/3/library/stdtypes.html#tuple

You can use indexing and slicing notation to access objects or sublists of tuples like the example below.

```
my_random_tuple = ('first',1,7,6,4,5,8,'hi there',2,3,1,5,2,1,9,10)
print(my_random_tuple[7])
print(my_random_tuple[7:])
```

Running the code above will give you the output below.

```
$ python lecture_tuples.py
hi there
('hi there', 2, 3, 1, 5, 2, 1, 9, 10)
```

You can 'unpack' the elements in a tuple by using tuple unpacking

```
my_tuple = ('first_value','second_value','third_value')
first, second, third = my_tuple # this line is 'unpacking' the
                                # values of the tuple
print(f"First value: {first}\nSecond value: {second}\nThird value: {third}")
```

Tuple unpacking is very useful when you receive tuples as return values from functions which need to return multiple values. You will see examples of this in the functions chapter as well. The code above (using tuple unpacking) allows you to access individual pieces of the tuple to work with. Running it will give you the output below.

```
$ python lecture_tuples.py
First value: first_value
Second value: second_value
Third value: third_value
```

You can find out if an object exists in your tuple using 'in'.

```
my_random_tuple = ('first',1,7,6,4,5,8,'hi there',2,3,1,5,2,1,9,10)
print('hi there' in my_random_tuple) # this will return True
```

You can iterate through the objects in a tuple using a simple for loop just like lists.

```
my_random_tuple = ('first',1,7,6,4,5,8,'hi there',2,3,1,5,2,1,9,10)
for item in my_random_tuple:
    print(item)
```

Try running the index and count methods, along with the len function on the tuples defined above.

This concludes our introductory look at Tuples. We will look at iterators in the next chapter.

Back to table of contents

For loops and generators

You can use for loops to iterate through iterables. Iterables can be collections of objects like lists, dictionaries, sets. They can also be sequences like strings.

Let's look at using a simple for loop to iterate through the elements in the list below.

```
l = [6, 8, 1, 4, 10, 7, 8, 9, 3, 2, 5]

for num in l:
    print(num)
```

You can perform functions like calculating the sum of all the integers in that list like below:

```
l = [6, 8, 1, 4, 10, 7, 8, 9, 3, 2, 5]

sum = 0
for num in l:
    sum += num
print(f"Sum using list: {sum}")
```

To iterate through key, value pairs in a dictionary, you can use the items method. It returns tuples of key, value pairs which you can iterate through and perform any function necessary. An example of using this along with a for loop is shown below. The 'unpacking' for the key, value pair from each tuple takes place in the for loop declaration itself (for k, v in…).

```
my_dict = {'py': 'python', 'rb': 'ruby', 'js': 'javascript'}
for k, v in my_dict.items():
    print(f"Extension of .{k} means it's a {v} program")
```

Generators and Iterables

You can also use the range function to generate an iterable of a specific number, then use a for loop to run that many times as you look at the elements in the index generated like below.

```
l = [6, 8, 1, 4, 10, 7, 8, 9, 3, 2, 5]

sum = 0
for num in range(len(l)):
    sum += l[num]
print(f"Sum using range generator: {sum}")
```

Range is a very useful function to use if you don't know ahead of time the number of iterations by the for loop to take place, example below where input from the user is used to define this number.

```
run_times = int(input("How many times do you want the program to run? "))
for num in range(run_times):
    print(f"Run: {num+1}")
```

You can generate a list of random integers using the range function in combination with the randint function from the random module like below.

```
from random import randint
l1 = []
for num in range(25):
    l1.append(randint(1, 100))
```

You can do the same function above using **list comprehension** in 1 line as shown below.

```
from random import randint
l1 = [randint(1,100) for num in range(25)]
```

Another generator which is very useful is the zip function. The zip function is used quite often to merge values from two lists together to form an iterable of tuple values. It can cast to a list and printed out to show the merged list of tuples. An example is shown below.

```
l1 = ['.py','.js','.rb','.java','.c']
l2 = ['python','javascript','ruby','java','c']
tupled_list = list(zip(l2, l1))
print(tupled_list)
```

If you ran the code above, it would result in the output below.

```
$ python lecture_forloops.py
[('python', '.py'), ('javascript', '.js'), ('ruby', '.rb'), ('java', '.java'), ('c',
```

If one list is bigger than the other, the unmatched items are simply ignored, for example if l2 looked like below:

```
l2 = ['python','javascript','ruby','java','c','c++']
```

And we ran the same code we did, `c++` would be dropped and not included in any of the tuples, output displayed below.

```
$ python lecture_forloops.py
[('python', '.py'), ('javascript', '.js'), ('ruby', '.rb'), ('java', '.java'), ('c',
```

This concludes our introductory look at iterators with for loops and generators. We will look at while loops in the next chapter.

Back to table of contents

While loops

You can use a simple while loop with a conditional test that evaluates to either `True` or `False` (or the boolean directly) and the code within the block will keep running while the condition is true. The while loop

will only stop when either the condition evaluation turns to false or a `break` keyword is used within the code block. An example of a while loop is shown below.

```
x = 0
while x < 10:
    print(x)
    x += 1
```

The while loop above will print values 0 through 9 and stop when x reaches value of 10. At that point the condition `x < 10` will be false since `10 < 10` is false and it will exit out of the while loop.

The example below shows the usage of a while loop to find a number in a generated list of numbers and the index at which it was found.

```
from random import randint
l1 = [randint(1,100) for num in range(1000)] # generated list of numbers
i = 0
num_to_search = 25  # the number to search
while i < len(l1):  # while loop won't run beyond the total number of
                    # ints in the list
    if l1[i] == num_to_search:
        print(f"{num_to_search} found at index {i}")
        break   # if the number is found, break out of nearest loop
    i += 1
```

The break keyword above breaks out of the nearest loop it is in (it happens to be the while loop in this case). If there are nested loops and you want to find which loop the break keyword will break out of, then look for the nearest for/while loop above the break statement.

Notice in the code above how we used the variable i to track the index in the list (by initially setting it to 0 and incrementing it at every iteration). You can simplify this and use the enumerate function instead to get the index in an iteration through an iterable.

For example if you ran the same code above with a for loop without tracking i, you can use enumerate like below:

```
l1 = [randint(1,100) for num in range(1000)]
num_to_search = 50
for index, value in enumerate(l1):
    if value == num_to_search:
        print(f"{num_to_search} found at index {index}")
        break
```

While loops are most useful when you don't know when the program execution will end. An example would be in a program where a user clicks on an exit button to exit the program and thus stops the loop execution like in case of a menu shown below:

```
while True:
    print("Please choose an option from the list below:")
    print("Press 1 for selection 1")
    print("Press 2 for selection 2")
    print("Press 3 to quit")
    selection = input("Enter your choice-> ")
    if int(selection) == 3:
        break
```

This concludes our introductory look at while loops. In the next chapter we will start looking at building our own custom functions.

Back to table of contents

Functions

You can build custom functions in Python using the `def` keyword followed by the name of the function. Functions may or may not take in arguments/parameters to process. A basic function example is provided below which does not take any arguments (no code within the open and close parenthesis after the function name).

```
def say_hello():
    print('Hello World!')
```

Functions by themselves are simply objects which are recognized by the python interpreter as functions when a script including them is executed. The code block inside the function is not executed by default. To actually run the code inside functions, they have to be called. For example, to run the function say_hello() defined above the following code could be used.

```
def say_hello():
    print('Hello World!')

say_hello() # this line calls the function
```

Running the code above would result in the output below.

```
$ python lecture_functions.py
Hello World!
```

Functions can take arguments as parameters to use within their function code. The following function uses 2 parameters to perform the add operation defined within it. As a result, two numbers (strings will work too)

will need to be provided as arguments when the function is called

```
def add_two_nums(num_1, num_2): # takes two arguments num_1 and num_2
    print(num_1 + num_2)

add_two_nums(4, 5) # 4 and 5 are provided in the function call
```

Functions usually return a value that they evaluate. In the example above, instead of printing out the summed value of the two numbers, the function can be made to return the value. This is done using the keyword return followed by the value. If nothing is specified by the explicit use of the return keyword, then the None datatype is returned by default by all functions to the caller of the function.

```
def add_two_nums(num_1, num_2):
    return num_1 + num_2

my_added_num = add_two_nums(4,5) # The return value from the function
                                 # call is being assigned to my_added_num
print(my_added_num)              # printing out value of my_added_num
```

We can do this without the variable `my_added_num` above.

```
print(add_two_nums(4,5))
```

Notice how we used the print function in the above examples to print out the value received from the function call. If we try to print the value of a function call that returns `None`, meaning no return value was specified, then it prints None to the screen.

```
def say_hello():
    print('Hello World!')

print(say_hello())
```

The code above will result in `Hello World!` being printed to the screen due to the code inside the function, followed by `None`, which is the return value from the function.

Output below.

```
$ python lecture_functions.py
Hello World!
None
```

You can use doctstrings inside functions to provide information about them. They are created using triple-quoted strings. An example docstring is shown below.

```
def say_hello():
    """This function prints Hello World! to the screen"""
    print('Hello World!')
```

Docstrings can be very detailed as well.

```
def add_two_nums(num_1, num_2):
    """
    This function adds two objects
    Input: Two objects to add, num_1 and num_2
    Output: The value of the two objects added together
    """
    return num_1 + num_2
```

When a function returns multiple values separated by commas in its return statement, these values are returned as tuples which can then be unpacked and used.

In the functions video in the course a step by step example is shown on converting repeated/duplicate code and logic in a script to a function, clearning up the code. That example code is shown below, initially the starter code in the script followed by the ‘extracted’ logic code using a function.

```
# Initial code
print("Welcome to the program, what is your name?")
name_result = input("Enter your response here -> ")
print(f"Your response was {name_result}")

print("What did you think of the food you ate today?")
food_result = input("Enter your response here -> ")
print(f"Your response was {food_result}")

print("What tv show ending did you dislike the most ever?")
show_result = input("Enter your response here -> ")
print(f"Your response was {show_result}")
```

```
# Final code using a function
def get_input_from_user():
    return input("Enter your response here -> ")

print("Welcome to the program, what is your name?")
name_result = get_input_from_user()

print("What did you think of the food you ate today?")
food_result = get_input_from_user()

print("What tv show ending did you dislike the most ever?")
show_result = get_input_from_user()

print(f"To summarize: Your name is {name_result.capitalize()},\
 you ate {food_result} food today and hated the ending\
 of {show_result.upper()}")
```

This concludes our introductory look at functions. We will use custom-built functions a lot in this course and get a lot of practice in the process. In the next chapter we will start looking at creating custom objects in python using classes.

Back to table of contents

Classes

You can create a custom object in Python by creating a class. In the course videos we create a custom Student class, at a minimum you can define it like below.

```
class Student:
    pass
```

The pass keyword indicates our intention to fill in code for the class later on.

We can create instances of the class, or instances of student objects by calling the class name followed by parenthesis `()` like below:

```
mashrur = Student()
john = Student()
```

Note: In the code above `mashrur` and `john` are separate instances of students. They are not the same object but are both type student (same class).

We want our student class to have attributes of first*name, last*name and eventually courses. You can add attributes to your objects like below.

```
mashrur.first_name = 'mashrur'
mashrur.last_name = 'hossain'
john.first_name = 'john'
john.last_name = 'doe'
```

Once you have written the code above, you can access the attributes associated with each student object like below.

```
print(mashrur.first_name, mashrur.last_name)
print(john.first_name, john.last_name)
```

You can use the special `__init__()` method to assign the attributes when the instance of the object is created instead of assigning them as shown in the code above.

The init method takes a default first argument which is called `self` by convention in python. `self` is the object being created itself, followed by any other attributes you wish to initialize when the object is created.

In the video we modified the class like below to reflect setting up first*name, last*name and courses attributes followed by some execution code at the bottom so we could test it out.

```
class Student:

    def __init__(self, first, last, courses=None):
        self.first_name = first
        self.last_name = last
        if courses == None:
            self.courses = []
        else:
            self.courses = courses

courses_1 = ['python','rails','javascript']
courses_2 = ['java','rails','c']
mashrur = Student('mashrur','hossain',courses_1)
john = Student('john','doe',courses_2)
print(mashrur.first_name, mashrur.last_name, mashrur.courses)
print(john.first_name, john.last_name, john.courses)
```

Running the code above will give you the following output.

```
$ python lecture_objects.py
mashrur hossain ['python', 'rails', 'javascript']
john doe ['java', 'rails', 'c']
```

Adding methods to the student class

You can add methods to your Student class. Regular methods look and feel almost the same as functions except they take self as an argument as well. In the Student class, the add and remove course methods were added in the course videos. The class looked like below once they were added, along with some execution code.

```
class Student:

    def __init__(self, first, last, courses=None):
        self.first_name = first
        self.last_name = last
        if courses == None:
            self.courses = []
        else:
            self.courses = courses

    def add_course(self, course):
        if course not in self.courses:
            self.courses.append(course)
        else:
            print(f"{self.first_name} is already \
enrolled in the {course} course")

    def remove_course(self, course):
        if course in self.courses:
            self.courses.remove(course)
        else:
            print(f"{course} not found")

courses_1 = ['python','rails','javascript']
courses_2 = ['java','rails','c']
mashrur = Student('mashrur','hossain',courses_1)
john = Student('john','doe',courses_2)
print(mashrur.first_name, mashrur.last_name, mashrur.courses)
print(john.first_name, john.last_name, john.courses)
print("Courses added to students")
mashrur.add_course('c')
john.add_course('c++')
print(mashrur.first_name, mashrur.last_name, mashrur.courses)
print(john.first_name, john.last_name, john.courses)
print("Courses removed from students")
mashrur.remove_course('rails')
john.remove_course('java')
print(mashrur.first_name, mashrur.last_name, mashrur.courses)
print(john.first_name, john.last_name, john.courses)
```

Running the code above results in the following output.

```
$ python lecture_objects.py
mashrur hossain ['python', 'rails', 'javascript']
john doe ['java', 'rails', 'c']
Courses added to students
mashrur hossain ['python', 'rails', 'javascript', 'c']
john doe ['java', 'rails', 'c', 'c++']
Courses removed from students
mashrur hossain ['python', 'javascript', 'c']
john doe ['rails', 'c', 'c++']
```

Adding special methods to the student class

You have already seen the `__init__()` special method we used earlier which gets called when a new object of the class is created. There are other special methods as well. For the Student class videos we explored the `__str__()` special method to get a string representation of the Student object, we also added the `__len__()` and `__repr__()` methods.

`__len__()` gives you the ability to display what you want when the len function is run with the object you are working with as the argument.

`__repr__()` is also a string representation of the class but it's meant more for other developers and not users of your program. `__repr__()` is usually supposed to display how to create an instance of the class in question, it also has the characteristic that it will be called to get a string representation of the class if `__str__()` is not present.

At the conclusion of adding these special methods, the Student class looked like below.

```
class Student:

    def __init__(self, first, last, courses=None):
        self.first_name = first
        self.last_name = last
        if courses == None:
            self.courses = []
        else:
            self.courses = courses

    def add_course(self, course):
        if course not in self.courses:
            self.courses.append(course)
        else:
            print(f"{self.first_name} is already \
enrolled in the {course} course")

    def remove_course(self, course):
        if course in self.courses:
            self.courses.remove(course)
        else:
            print(f"{course} not found")

    def __len__(self):
        return len(self.courses)

    def __repr__(self):
        return f"Student('{self.first_name}','{self.last_name}',{self.courses})"

    def __str__(self):
        return f"First name: {self.first_name.capitalize()}\nLast name: {self.last_r
        \nCourses: {', '.join(map(str.capitalize, self.courses))}"


courses_1 = ['python','rails','javascript']
courses_2 = ['java','rails','c']
mashrur = Student('mashrur','hossain',courses_1)
john = Student('john','doe',courses_2)
print(mashrur)
print(john)
print(f"{mashrur.first_name.capitalize()} is enrolled in {len(mashrur)} courses")
print(f"{john.first_name.capitalize()} is enrolled in {len(john)} courses")
```

At this point in the course videos we explored adding the ability to work with files (persistent memory), then we explored two other topics with static methods and inheritance. These are best explained with demonstrations in the videos themselves so I recommend checking them out. The text lectures provided with the videos include code used in them as well.

This concludes our introductory look at the Python programming language. I hope you enjoyed this e-book and found the content useful. We will use concepts we learnt here heavily through the rest of the course so you will get a lot of opportunity to practice programming concepts with Python. I look forward to working with you through the rest of the course starting with the sorting Algorithms section.

Back to table of contents