

Module 4 Part 1: Object-Oriented Programming with Python and Additional Python Functions

Introduction

We will start this module with a quick review of the basics of **Object-Oriented Programming (OOP)** and how it applies to Python. We will explain the meaning of the statement "in Python **everything** is an object."

Then, we will review two useful functions, `map()` and `lambda`, that you will use frequently, especially as you learn about **pandas**, Python's implementation of a **DataFrame** concept which will give you very powerful tools for data handling.

This module consists of 3 parts:

- **Part 1** - Object-Oriented Programming with Python and Additional Python Functions
- **Part 2** - Introduction to pandas
- **Part 3** - Modifying a DataFrame, data aggregation and grouping, Case Studies

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

Learning Outcomes

By the end of this Module, you will learn the following topics:

- Basics of Object-Oriented Programming (OOP) and its implementation in Python
- More Python:
 - `map()` and `lambda` functions
 - Defining arguments for a function
- Pandas:
 - Introduction to Series
 - Basics of **pandas**, creating a **DataFrame**
 - Reading/writing data from/to a **.csv** file
 - Selecting data from the **DataFrame** by column name(s) or index, slicing and indexing
 - Modifying a **DataFrame**

- Pandas data aggregation and grouping, understanding functions `groupby()` , `stack()` / `unstack()` , `reset_index()`
- Case Studies to practice pandas

Readings and Resources

The majority of the notebook content draws from the recommended readings. We invite you to further supplement this notebook with the following recommended texts:

1. Python Community (2018). *The Python Tutorial*. Retrieved from <https://docs.python.org/3/tutorial/index.html>.
2. Kuchling A.M. (2018). *Functional Programming HOWTO*. Retrieved from <https://docs.python.org/3.6/howto/functional.html>.

Table of Contents

- [Module 4 Part 1: Object-Oriented Programming with Python and Additional Python Functions](#)
- [Introduction](#)
- [Learning Outcomes](#)
- [Readings and Resources](#)
- [Table of Contents](#)
- [Object-Oriented Programming](#)
 - [Basics of Object-Oriented Programming](#)
 - [Classes and objects in Python](#)
 - [Characteristics of Python objects](#)
 - [EXERCISE 1: Explore Python objects](#)
- [More Python](#)
 - [map\(\) and lambda functions](#)
 - [Defining arguments for a function](#)
- [References](#)

Object-Oriented Programming

Before we dive into pandas library and DataFrame structures, let's talk about Object-Oriented Programming (OOP) and the specifics of OOP in Python. This will help us to better understand some of the concepts of data manipulation in Python.

Basics of Object-Oriented Programming

Object-oriented programming (OOP) refers to a programming language model which defines *objects* containing data and functions that determine what types of operations can be applied to the data. If you are new to OOP, you can familiarize yourselves with the basics by reviewing the following web page on Wikipedia for general understanding of terms and concepts: [Object-Oriented Programming](#). If you are looking for a book explaining the concepts of object-oriented software development, you can get a copy of the classic book "[Design Patterns: Elements of Reusable Object-Oriented Software](#)" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

At the core of the OOP are the concepts of a **class** which *defines the data format and available procedures* of an **object**. Any object is *an instance of a corresponding class*.

Here is one of the definitions of classes and objects:

Concept	*Definition*
Classes	The definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods), i.e. classes contain the data members and member functions.
Objects	Instances of classes

Think of a class as a *template or a blueprint* for any object created of this class.

We will use a simple example to explain the concept and components of OOP without writing any code first. For this initial example, we will define a class **Vehicle**. This class can define the following *fields* and *functionality* of a generalized vehicle:

Class Example	*Field Examples*	*Functionality Examples*
Vehicle	model, year, colour, type, transmission type	EngineStart(), accelerate(), EngineStop(), turn() and others

These functions are called **methods**. Together, fields and methods are called **attributes** of the class.

Using this class, we can create multiple **objects** of a class Vehicle which will have different combinations and values of the attributes defined by the class.

When a new object is created, we say that we create an instance of a class, or **instantiate** a new object.

Classes and objects in Python

In the next example we will create a class **Book**. It will accept two parameters, an author of the book and its title. The class will have a method which will print out the book description as a single sentence listing the title and author of the book.

A class in Python is created using the keyword `class`, then we type the name of the class we want to create, and a colon (`:`). The class body is indented (usually 4 spaces).

NOTE: Python tracks how spaces and tabs are used when indenting, resulting in the ability to mix spaces and tabs. We suggest always using 4 spaces to indent Python code.

Here is the definition of a class `Book`:

```
In [4]: class Book:
        def __init__(self, author, title):
            self.author = author
            self.title = title
        def bookdesc(self):
            return "Book '{}' by {}".format(self.title, self.author)
```

The class `Book` defines two methods, `__init__()` and `bookdesc()`.

The name of the method `__init__()` starts and ends with double underscores (often called "dunder") — it is one of the so-called magic methods. This method is called automatically every time the new instance of a class is created. This method is called a **constructor** and it **initiates** fields of the class. In our example, method `__init__()` has 3 arguments. The argument `self` is always the first argument of a method `__init__()` and it refers to the object itself which helps Python keep track of different instances of the class.

The other two arguments of a method `__init__()` are the author name and title of a book. The function will **initialize** the new object by assigning initial values to the object.

The second method, `bookdesc()`, will print out the name and title of the book. It also takes `self` as an argument. Let's create some books:

```
In [2]: bookA = Book("George R.R. Martin", "A Game of Thrones")
```

Let's explain what Python did when the above line of code was executed:

1. Python interpreter has created a new instance of a class `Book`
2. Method `__init__()` was executed automatically. It used the string `"George R.R. Martin"` as the value of parameter `author` and string `"A Game of Thrones"` as a value of parameter `title` for the method `__init__()`. The code within the function initialized attributes `author` and `title` of a new object. These are called **instance attributes**.

3. A new object is assigned to a variable `bookA` . This object has two attributes, `author` and `title` .
4. The new object has a method `bookdesc()` which will print out the title and author of a book. This method is called an **instance method**.

We can check the type of an object:

```
In [3]: type(bookA)
```

```
Out[3]: __main__.Book
```

Variable `bookA` points to an object of class `Book` .

We can check the values of the `title` and `author` instance attributes of this object:

```
In [4]: bookA.author
```

```
Out[4]: 'George R.R. Martin'
```

```
In [5]: bookA.title
```

```
Out[5]: 'A Game of Thrones'
```

Calling instance method `bookdesc()` :

```
In [6]: bookA.bookdesc()
```

```
Out[6]: "Book 'A Game of Thrones' by George R.R. Martin"
```

More information about Python classes can be found in the Python documentation, [Chapter 9, Classes](#). in The Python Tutorial.

Characteristics of Python objects

Each object in Python has three characteristics:

1. Object type
2. Object value
3. Object identity

Object type tells Python what kind of an object it's dealing with. A type could be a string, or a list, or any other type of Python object, including custom-defined objects, like we saw above when we checked the type of object that the variable `bookA` is pointing to.

Object value is the data value contained by the object. In the example above, it was the title of the book and name of an author. For an integer, for example, it can be the value of a number.

Object identity is an identity number for the object. Each distinct object in the computer's memory will have its own identity number.

To check an identifier of an object, we can use method `id()`. It returns an integer which uniquely identifies an object:

```
In [7]: id(bookA)
```

```
Out[7]: 4482394376
```

The number above is an ID of an object which variable `bookA` is pointing to. Multiple variables can point to the same object. Let's illustrate this with a simple example.

First, we will create a new list object `listobj` of two integers:

```
In [8]: listobjA = [22, 33]
```

This object has a **type** and **id** which we can easily obtain using the corresponding methods:

```
In [9]: type(listobjA)
```

```
Out[9]: list
```

```
In [10]: id(listobjA)
```

```
Out[10]: 4476928968
```

We can create another variable, `listobjB`, which will point to the same list object `[22, 33]`:

```
In [11]: listobjB = listobjA  
listobjB
```

```
Out[11]: [22, 33]
```

We can easily confirm that `listobjB` points to the same list by validating its `id()`:

```
In [12]: id(listobjB)
```

```
Out[12]: 4476928968
```

And just as we expected, the **ID** of `listobjA` and `listobjB` are exactly the same.

Another way of confirming that `listobjA` and `listobjB` are pointing to the same object is to use the `is` operator:

```
In [13]: listobjA is listobjB
```

```
Out[13]: True
```

We can also validate the *equality* of these two objects by using the operator `==` :

```
In [14]: listobjA == listobjB
```

```
Out[14]: True
```

And we can see from the above that `listobjA` and `listobjB` variables have the same value and are pointing to the same list object.

Before we continue, let's create another list object, name it `listobjC` and it will be a copy of `listobjA` .

```
In [15]: listobjC = list(listobjA)
```

```
In [16]: # We can validate that it has the same value:
```

```
listobjC
```

```
Out[16]: [22, 33]
```

```
In [17]: # but different ID:
```

```
id(listobjC)
```

```
Out[17]: 4481851720
```

```
In [18]: listobjC is listobjA
```

```
Out[18]: False
```

```
In [19]: # even though they are equal:
```

```
listobjC == listobjA
```

```
Out[19]: True
```

Now let's add a third integer, `44` , to the end of the original list object using `append()` function:

```
In [20]: listobjA.append(44)  
listobjA
```

```
Out[20]: [22, 33, 44]
```

```
In [21]: # The ID of this object didn't change, it is the same object as above:
```

```
id(listobjA)
```

```
Out[21]: 4476928968
```

What happened to `listobjB` ? If it was pointing to the same list object as `listobjA` , it should have changed as well. Let's check:

```
In [22]: listobjB
```

```
Out[22]: [22, 33, 44]
```

And it changed indeed. But what about `listobjC` which we created as a copy of the original list?

```
In [23]: listobjC
```

```
Out[23]: [22, 33]
```

`listobjC` didn't change because it is a separate object with its own ID.

This is a very important concept to remember when working with Python objects.

We also need to point out that each element within a list object is an object as well. The list object `listobjA` is a container of objects which are the elements of the list. In our case the elements are integers with individual IDs. Let's check the object ID of each element within a list:

```
In [24]: id(listobjA[0])
```

```
Out[24]: 4432927216
```

```
In [25]: id(listobjA[1])
```

```
Out[25]: 4432927568
```

```
In [26]: id(listobjA[2])
```

```
Out[26]: 4432927920
```

```
In [27]: # Since listobjB is pointing to the same list object as listobjA,  
# the object ID of the last, third element, should be the same  
  
id(listobjB[2])
```

```
Out[27]: 4432927920
```

```
In [28]: # List elements are of type `integer`  
# whereas the list itself is of type `list`  
  
print(type(listobjA[2]))  
print(type(listobjA))
```

```
<class 'int'>  
<class 'list'>
```


EXERCISE 1: Explore Python objects

To visually demonstrate this important concept, we will use the **Python Tutor** website which helps to visualize what happens when each line of code is executed:

<http://pythontutor.com/live.html#mode=edit>.

We will open the interactive session so that we can see the changes, and reproduce all the steps above. You are encouraged to open the link in a separate browser window and follow along.

NOTE: If you want to make sure that you see the same visualization as the screen shots below, please set the following values for the three drop-down lists at the bottom of the screen:

- First drop-down: "**show exited frames (Python)**"
- Second drop-down: "**render all objects on the heap (Python)**"
- Third drop-down: "**draw pointers as arrows [default]**". During the exercise, you can also select "use text labels for pointers" in this drop-down, it will allow you to see object ID instead of arrows.

Refer to the image below:



Drop-down settings in Python Tutor window

Picture 1. Drop-down settings in Python Tutor window


Step 1 - Create `listobjA` as a list object with two values: `[22, 33]`

A screenshot of a Jupyter Notebook cell. The cell contains the text "List `listobjA` is created" in a monospace font. To the left of the text is a small icon of a document with a green checkmark.

Picture 2. Create list object `listobjA`

As you can see in the image above, the list `listobjA` is a container of type `list` with two elements, both elements are integer objects. The visualization also shows indexes for the list elements within a list object.

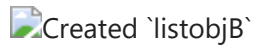
If we change the value of the third drop-down to "use text labels for pointers", we will see that the list object itself has an id = `id1` and the list elements have IDs `id2` and `id3`. The tool also shows each element of a list being an independent object of type `int`.

 List object `listobjA` with object IDs displayed

Picture 3. List object `listobjA` with object IDs displayed

NOTE: IDs in the code above and IDs generated by the Python Tutor will be always different. For simplicity and demonstration purposes, Python Tutor IDs will start from `id=1` when you refresh the screen and start a new session.

Step 2 - Change the value of the third drop-down back to "draw pointers as arrows [default]". Create `listobjB` as follows `listobjB = listobjA` in the Python Tutor window. You should see the following result:

A screenshot of a Python REPL window. The text 'Created `listobjB`' is displayed, indicating that a new variable has been created and assigned to the same object as the previous one.

Picture 4. Created `listobjB`

As we expected, Python simply created a new variable, `listobjB`, and pointed it to the same list object that `listobjA` is pointing to.

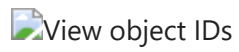
Step 3 - In the Python Tutor window, create `listobjC` as follows: `listobjC = list(listobjA)` .

Here is the result of this step:



Picture 5. Create `listobjC`

We can note an interesting detail — Python will create a new list object for `listobjC` but it will point to the same list elements, integer objects `22` and `33`. If we switch into "use text labels for pointers" view, we can see that an ID for `listobjC` is `id4` :

The image shows a small portion of a Jupyter Notebook interface. It features a button with a document icon and the text "View object IDs".

Picture 6. View object IDs for all objects

Step 4 - Using the arrow view, add an element with value `44` to the end of `listobjA`

 Modified `listobjA` by adding a new element to the list

Picture 7. Modified `listobjA` by adding a new element to the list

Step 5 - If we modify `listobjA` by assigning a list with different values to a variable `listobjA`, will `listobjA`'s ID change? For example, you can do something like this:

```
listobjA = [1,2,3,4,5]
```

You can validate the result either by typing your code below and/or using Python Tutor.

```
In [6]: listobjA = [22, 33]
```

```
In [8]: listobjB = listobjA
listobjC = list(listobjA)
```

```
In [16]: listobjA.append(44)
listobjA = [1,2,3,44,5]
listobjA
listobjB
```

Out[16]: [22, 33, 44, 44, 44]

In [29]: *# Solution, part 1 - code*

```
listobjA = [1,2,3,44,5]  
listobjA
```

Out[29]: [1, 2, 3, 44, 5]

In [14]: *# checking the type of the object listobjA*

```
type(listobjA)
```

Out[14]: list

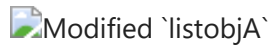
In [15]: *# validating an ID for the object listobjA*

```
id(listobjA)
```

Out[15]: 1468619840640

You can compare the ID with the ID in the beginning of this notebook chapter. You will see that Python created a new list object and pointed the `listobjA` variable to this new list object which is a container for five integer objects. However, the `listobjB` variable still points to the original object — nothing changed for `listobjB`.

Here is a screenshot from Python Tutor to demonstrate the changes:



Picture 8. Modified `listobjA`

Here is the [link](#) to the screen above.

More Python

In this section, we will cover additional Python functions: `map()` and the `lambda` function. They will come handy when we start manipulating DataFrame objects in the next section of this module. We will also learn how we can define a function when we do not know in advance how many attributes there will be when the function is called.

`map()` and `lambda` functions

It is important to note that in Python, functions can be passed as arguments to another function. Function `map()` usually takes a function as one of the parameters and applies it to

the elements of an **iterable** object, which can be list, dictionary, or even a string.

Let's define **iterables** and **iterators** in Python:

According to the Python's [Glossary of Terms](#): **iterable** is "an object capable of returning its members one at a time." In other words, iterable is a Python object we can loop over. Lists, strings, tuples, dictionaries are all examples of iterables.

An iterable object has a built-in method `__iter__()` which returns an **iterator** object. An iterator in Python is "an object representing a stream of data" — it enables iteration over each element within an iterable container. This object returns the data one element at a time with the method `__next__()`.

We don't have to explicitly use any of these methods, but when we use, for example, a `for` statement to loop over a list, the `__next__()` method is called automatically to get each item from the iterator:

```
for element in [1, 2, 3]: print(element)
```

You can find more examples in the Python online documentation, chapter ["9.8. Iterators"](#) of [The Python Tutorial](#) and section ["Iterators"](#) of ["Python Functional Programming HOWTO"](#).

If you have a list, you might need to loop through the list and apply a calculation defined by a function to each element of a list, or only to certain elements. In the example below, we will square each element of a list using Python function `map()`.

Syntax of a function: `map(function, iterable)`. Let's review how this function is used.

```
In [32]: # First, we create a custom square_it() function:
```

```
def square_it(x):  
    return x * x  
  
# Create a list  
  
listA = [1, 2, 3, 4, 5, 6]  
  
# Now we can use map() function:  
  
list(map(square_it, listA))
```

```
Out[32]: [1, 4, 9, 16, 25, 36]
```

NOTE: `map()` returns an iterator object. In order to print out the values, we converted this iterator object to a list.

Function `square_it()` is very short function and can be replaced with the **lambda function**. It is an anonymous function which does not have a name and thus cannot be re-

used elsewhere. It can be useful if, for example, we need to pass a simple one-line custom function to the other function.

NOTE: The `lambda` function should only be used for applying very simple functions.

We can re-write the code above using the lambda function as follows:

```
In [33]: list(map(lambda x: x * x, listA))
```

```
Out[33]: [1, 4, 9, 16, 25, 36]
```

```
In [34]: # another example of using the Lambda function:
```

```
list(map(lambda x: x + x, listA))
```

```
Out[34]: [2, 4, 6, 8, 10, 12]
```

Defining arguments for a function

While creating a function, we might not know of all possible use cases when the function is going to be used. In order to handle a variable number of arguments during execution, we can define the function using `*args` and `**kwargs` syntax to define arguments.

The **single-asterisk keyword** (`*`) is used when we need to use a list of arguments of variable length.

Here is an example of a function which will take a list of numbers and sum them up:

```
In [35]: def my_sum(*args):  
         a = 0  
         for num in args:  
             a += num  
         print(a)
```

```
In [36]: my_sum(4, 6, 10)
```

```
20
```

```
In [37]: my_sum(3, 5.12, 45.78, 100, 123)
```

```
276.9
```

The **double-asterisk keyword** (`**`) is used when we pass keyword arguments which are in the form of a dictionary.

```
In [38]: def myfunc(**kwargs):  
         for k,v in kwargs.items():  
             print ("%s = %s" % (k, v))  
  
         # Using the function with one parameter:  
         myfunc(a=1)
```

```
a = 1
```

```
In [39]: # Using 2 parameters  
myfunc(b=2, a=1)
```

```
b = 2
```

```
a = 1
```

NOTE: The arguments don't have to be named `args` and `kwargs`. This is just a naming convention accepted by the community. However, the asterisk(s), `*` and `**`, are what define the behaviour of the function call. To demonstrate, we can re-write the first function as follows:

```
In [40]: def my_newsum(*listofargs):  
    a = 0  
    for num in listofargs:  
        a += num  
    print(a)  
  
my_newsum(4, 6, 10)
```

```
20
```

You can read more on [Arbitrary Argument Lists](#) in the Control Flow Tools section of [The Python Tutorial](#) documentation.

```
In [41]: def catch_all(*args, **kwargs):  
    print("args =", args)  
    print("kwargs = ", kwargs)  
  
catch_all(1, 2, 3, a=4, b=5)
```

```
args = (1, 2, 3)
```

```
kwargs = {'a': 4, 'b': 5}
```

End of Part 1

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

References

VanderPlas, Jake (2017). Basic Python Semantics: Variables and Objects. Everything Is an Object. In *A Whirlwind Tour of Python*. Free book, available at the website:

<https://jakevdp.github.io/WhirlwindTourOfPython/03-semantics-variables.html#Everything-Is-an-Object>

Wikipedia. (2018). Object-oriented programming. Retrieved from https://en.wikipedia.org/wiki/Object-oriented_programming

Python Documentation. The Python Tutorial. (2018). Chapter 9, Classes. Retrieved from <https://docs.python.org/3.3/tutorial/classes.html>.

Python Tutor Website. (2018) Available at the following URL <http://pythontutor.com/live.html#mode=edit>

Python Documentation. (2018). Glossary, term *iterable*. Retrieved from <https://docs.python.org/3.6/glossary.html#term-iterable>

Python Documentation. (2018). Glossary, term *iterator*. Retrieved from <https://docs.python.org/3.6/glossary.html#term-iterator>

Python Documentation. The Python Tutorial. (2018). Section 9.8. Iterators. Retrieved from <https://docs.python.org/3/tutorial/classes.html#iterators>

Kuchling, A.M. (2018). Iterators. In *Python Documentation. Python HOWTOs. Functional Programming HOWTO*. Retrieved from <https://docs.python.org/dev/howto/functional.html#iterators>

Python Documentation. The Python Standard Library. (2018). Chapter 2. Built-in Functions, `map()` function. Retrieved from <https://docs.python.org/3/library/functions.html#map>

Kuchling, A.M. (2018). Small functions and the lambda expression. In *Python Documentation. Python HOWTOs. Functional Programming HOWTO*. Retrieved from <https://docs.python.org/dev/howto/functional.html#small-functions-and-the-lambda-expression>

Python Documentation. The Python Tutorial. (2018). 4.7.3. Arbitrary Argument Lists. Retrieved from <https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists>