Bootcamp: Python - Module 4

John Rajadayakaran Edison, Sara Miner More, Eli Sherman, Soumyajit Ray

October 2022

Contents

1 Files

Most computer programs process data to obtain a result. This data can be provided in a couple of ways. It can be hard-coded in the program, collected interactively from the user, or supplied in external files. The latter is what we will focus in this section. We will learn about (i) obtaining data from files, (ii) writing data to files and (iii) a few useful file formats.

1.1 File Object

Below are the contents of a file called `inventory.dat`. This is a text file that contains 5 lines. Suppose we need to read the data in this file, or in any one that stores text or binary data.

```
Civic 10
CRV 20
City 39
Pilot 5
Odyssey 4
```

To obtain access to this file, we use the built-in Python function `open`. This function requires at least one argument: the path, or location of the file in the filesystem. In this case, since the file is present in the same directory as the script, we will simply provide the filename, without the path (see the following subsection). Alternatively, you can provide the absolute path to the file within the filesystem (e.g., `/Users/dayakaran/Dropbox/bcp/module_4/inventory.dat`).

1.2 Reading a file

If you'd like to open a file to read data only, use the `mode` keyword and pass `'r'` as an argument (see line 1 below). The `open` function returns a `File` object.

```
1 my_file = open('inventory.dat', mode='r')
2 print(type(my_file))
```

```
<class '_io.TextIOWrapper'>
```

Actually, as you can see from the output of the `print` statement above, `my_file` is an instance of the class `TextIOWrapper` found in the `io` module. This module is part of Python's standard library, and therefore its members can be accessed without any `import` statements. The table below describes some useful methods of the `File` class that allow us to read data.

Method	Description
`readline()`	Returns the next line in the file.
`readlines()`	Returns a list, with each item in the list corresponding to a single line of data in the file.
`read()`	Returns all the contents of the file.

Every file object that is created has a tracker that corresponds to a specific position in the file. Upon instantiation of the file object (line 1), this tracker points to the beginning, or the first line of the file. When one of the three read methods above is executed, the tracker changes its position accordingly.

```
1 my_file = open('inventory.dat', mode='r')
2 print(my_file.readline(), end="")
3 print(my_file.readline(), end="")
4 my_file.close()
```

```
Civic 10
CRV 20
```

In the code above, the `readline` method is used twice, and afterwards, the tracker will point to the third line. You can move the tracker to a specific position in the file using the `seek` method. A complete list of the methods associated with file objects can be found at https://docs.python.org/3/library/io.html.

Once a file is processed, it is good coding practice to close it. This is because this file might be accessed or over-written several times within the same program or script. In order to avoid corruption of data, it is best to close the file once the task is complete. In Python, we close a file using the `close` method shown on line 4 above.

1.3 Writing to a file

Files can also be opened to write data. In this case, the `mode` keyword of the `open` function is set to `'w'`. The table below shows some of the possible values of the `mode` keyword.

Mode	Description
`'r'`	Open file only for reading. (Default)
`'w'`	Open file for writing. Overwrites the contents of if the file is already present.
`'x'`	Creates a new file and fails if the file already exists.
`'a'`	Open file and append data to it.
`'+'`	Open file for both reading and writing.

Let's write some data to the `'inventory.dat'` file, which we accessed above. If we want to update the data in the file, and not overwrite the contents, we must use the

append (`'a'`) mode. In append mode, any new data gets added to the end of the file. If desired, we could instead overwrite the existing data using the write (`'w'`) mode. In order to write data, we use the `write` method of the file object. This method can take any string, formatted or otherwise and add it to the file.

```
my_file = open('inventory.dat', mode='a')
my_file.write('Accord 21\n')
my_file.close()

my_file = open('inventory.dat', mode='r')
for line in my_file.readlines():
    print(line, end="")
my_file.close()
```

```
Civic 10
CRV 20
City 39
Pilot 5
Odyssey 4
Accord 21
```

This concludes our discussion of working with text files.

1.4 JSON format

The process of reading and writing to a file involves just a few lines of code. However all of our data must be converted to strings before it is written to a file. This is not much of a hassle if we work with simple data types like numbers and text. However, if there is a need to store more complex, and structured data objects, like lists, or dictionaries, the above file read/write methods prove inconvenient. Extra code has to be written to convert the objects to strings, and later when strings are read back in, to convert them back into objects. This is where JSON files can help.

JSON stands for JavaScript Object Notation. The `json` module provides functions `dump` and `load` that conveniently store and retrieve Python objects like sets, lists and dictionaries. In the code snippet below, a list of numbers are written and read from a file, with a single line of code each.

```
import json

# create a list
my_list = [i for i in range(5)]

```

```
# open a file for writing; use dump to put entire list contents in it
my_file = open('list.json', 'w')
json.dump(my_list, my_file)
my_file.close()

# now open for reading; use load to read in entire list contents at once
my_file = open('list.json', 'r')
read_list = json.load(my_file)
my_file.close()

# output to the console the list contents that were read in
print(read_list)
```

```
[0, 1, 2, 3, 4]
```

2 Error handling

In this section, we will learn how to handle errors in Python scripts. You probably have encountered a few syntax errors by now. They are typically flagged before the execution of the code. For example, a common syntax error is missing a colon at the end of an `if` statement. However, even code that is syntactically correct may contain execution errors. In this section, we will learn about some error handling strategies.

2.1 Debugging

A typical software package developed in Python contains many thousand lines of code, split over multiple files. Often many developers work together on such projects. As a developer, you may not have complete control over the behavior of every single component of this software. Hence it is important to add code that anticipates erroneous behavior for the purposes of debugging. The `assert` statement can be used to flag errors at the debug stage of the program development.

2.1.1 Using `assert`

```
import math

def compute_func (x , y):
    return 3*(x+y)

print(compute_func('a','b'))
```

ababab

Consider the function above. Every line of code is syntactically correct. The function `compute_func` will run without any errors, for all float or integer inputs. If data of another type, say a string or list is passed as an argument, it will still execute (see output). However we cannot feed the output of this function to other components of a program that expect numbers as inputs. Hence it is good practice to add code that alerts the developer of such situations in the development stage of the program. Here is how we can put the `assert` statement to use in this situation.

```
1 import math
2
3 def compute_func (x, y):
4     assert (isinstance(x,int) or isinstance(x,float)), "Pass a number"
5     return 3*(x+y)
6
7 print(compute_func('0','1'))
```

Next to the `assert` keyword we write a conditional expression. If the expression evaluates to `False` an exception is forced. The `assert` statement written on line 4 above is equivalent to the following code segment.

```
1 if __debug__:
2     if not (isinstance(x,int) or isinstance(x,float))
3         raise AssertionError, "Pass a number"
```

Note that the `__debug__` flag is `True` by default. It is turned off only when Python is run with optimization flags. The keyword `raise` forces an exception to occur and halts the program. The expressions following the `raise` keyword are then printed to standard output.

2.2 Runtime Errors

Here are a few common error types you may encounter when writing code in Python. A complete list of the standard exceptions can be found at https://docs.python.org/3/library/exceptions.html.

Exception	Occurs when
`ImportError`	there is a problem with an `import` statement.
`IndexError`	using indexing to access a list or tuple element that does not exist.
`KeyError`	a specified dictionary key is not found.
`TypeError`	an operation is applied to an incompatible datatype, e.g., `name**2`, where name is a string.
`FileNotFoundError`	trying access a file that does not exist.

Ideally when your code is shipped as a package or distributed as a module to other users, you want it to run seamlessly, and not throw any exceptions. You must anticipate that the end user may potentially use your package or module incorrectly. And therefore your code must include functionality to trap common errors and allow the code to continue without halting. One way to do this is by using the `try... except` block.

2.2.1 `try...except...else`

Let's pick one of the common errors, from the table above: `FileNotFoundError`. We'll assume we have a function in our code where a file is being accessed to collect some data. This function receives a filename as an argument. We can anticipate that the provided filename could have typos or the end user may provide the wrong path. Here is how we can use the `try` block to handle this situation.

```
1  def process_data (filename):
2      try:
3          my_file = open(filename,'r')
4          for line in my_file.readlines():
5              print(line)
6
7      except FileNotFoundError:
8          print("File "+filename+" does not exist")
9
10     except:
11         print("Unknown error while accessing data in " +filename)
12
13     else:
14         my_file.close()
15
16     finally:
17         print("Function execution complete")
18
19     return
20 process_data('inventry.dat')
```

```
File inventry.dat does not exist
Function execution complete
```

In the above example, we pass an incorrect filename to the function. Instead of the program failing, an error message gets printed. Here is the how the above block of code executes. Figure 1 summarizes the flow of execution.

- First, the statements in the `try` block are executed. If no exception occurs, the `except` clause is skipped and execution moves to the `else` clause.
- If an exception occurs, execution moves to the corresponding `except` clause. In other words, if a `FileNotFoundError` error occurs, the flows moves to the `except` clause that corresponds to this error. Note the name of the exception mentioned next to the `except` keyword in line 7.

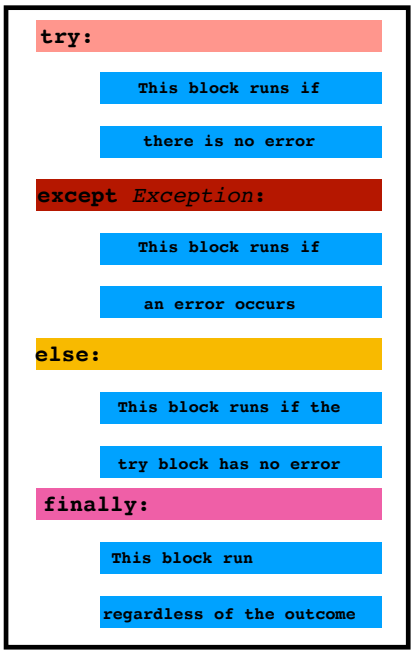

Figure 1: Schematic that describes the flow of execution of a `try...except...else` block.

- Mentioning the type of exception after the `except` keyword is optional. If an error other than the `FileNotFoundError` occurs in the `try` block, the the execution moves to the `except` clause in line 10. Since no specific exception type is listed, this expression handles all other possible errors.
- One can also add a `finally` clause to the `try` block. Statements in the `finally` clause are executed regardless of the outcome of the `try` block.

3 NumPy

In this final section, we will learn about a Python package called `NumPy` (often pronounced "numb-pie", or in other words, we say the "num" as in number and the "py" as in Python; an

alternative pronunciation is "num-pee" which sounds funny like 'clumpy'), which is widely used for scientific computing. The core of this package is written in the low-level C programming language and is highly optimized. The package provides several functions, modules and objects that can be accessed and created from a Python script.

The central object that NumPy makes available to users is a multi-dimensional numerical array. Since Python already provides a list data structure, you may wonder how NumPy arrays are helpful. Unlike lists, NumPy arrays are strictly homogeneous, i.e., every element of the array is of the same datatype. NumPy arrays are leaner, and data processing is much faster thanks to the optimized code. If you are a MATLAB user, you may find it helpful to refer to https://numpy.org/devdocs/user/numpy-for-matlab-users.html to learn the NumPy equivalents of several MATLAB functionalities.

Since NumPy is optimized, is incredibly popular for use in AI and machine learning applications. While Python lists are very syntactically convenient (e.g. mutable, able to store heterogeneous data types), these conveniences are computationally costly and so lists are not typically light-weight enough for the heavy data processing needed in machine learning applications. Beyond it's direct use among AI/ML researchers, other popular AI/ML libraries are built on top of NumPy, such as Pandas, pyTorch, and TensorFlow. In other words, NumPy is a very good library to know well!

3.1 Creating a NumPy array.

There are several ways of creating a NumPy array. The code snippet below uses some of the functions available in the NumPy module to create the arrays. All of these functions return a NumPy array object. In order to access these functions, one should first load the NumPy module.

```
import numpy as np

x = np.zeros(5, dtype=np.int32)
print("Zeros:", x)

x = np.ones(5, dtype=np.float64)
print("Ones:", x)

x = np.array([1,2,4,5], dtype=np.float32)
print("From list:", x)

# create an array with 6 evenly-spaced values spread over [0,1]
x = np.linspace(0,1,6)
print("6 pts between 0, 1:", x)

# Similar to the range function
x = np.arange(5)
print("Arange:", x)
```

```
Zeros: [0 0 0 0 0]
Ones: [1. 1. 1. 1. 1.]
From list: [1. 2. 4. 5.]
6 pts between 0, 1: [0.  0.2 0.4 0.6 0.8 1. ]
Arange: [0 1 2 3 4]
```

Every array creation function may take an optional keyword argument called `dtype`. This sets the datatype of the `NumPy` array. The entire list of datatypes provided by `NumPy` can be found in at https://numpy.org/devdocs/user/basics.types.html. Several of these types are displayed below.

dtype	Description
`np.int32`	Integer (-2147483648 to 2147483647)
`np.uint32`	Unsigned Integer (0 to 4294967295)
`np.float64`	64 bit Double precision floats.
`np.float32`	32 bit Single precision float.

3.2 Attributes

Some useful attributes of a `NumPy` array object are `ndim`, `size`, `shape`. These attributes describe the size and structure of the array.

```
1  import numpy as np
2
3  # Create a two by two matrix
4  a = np.zeros((2,2))
5
6  print(a)
7
8  print("No. of dimensions of a ", a.ndim)
9  print("Shape of array a ",a.shape)
10 print("No. of elements of a ",a.size)
```

```
[[0. 0.]
 [0. 0.]]
No. of dimensions of a  2
Shape of array a  (2, 2)
No. of elements of a  4
```

3.3 Reshaping data

NumPy allows you to modify the shape of the array after it has been created, using the reshape method. The data in the array stays intact.

```
1 import numpy as np
2
3 x = np.linspace(1,6,6,dtype=np.int32)
4 print(x)
5
6 b = x.reshape(2,3)
7 print("Array after reshaping ")
8 print(b)
```

```
[1 2 3 4 5 6]
Array after reshaping
[[1 2 3]
 [4 5 6]]
```

3.4 Accessing array elements

NumPy array elements can be accessed in the same fashion as list elements, via indices. NumPy arrays also support slicing. Creating subsets of data using conditional expressions in the following manner is a functionality that lists don't provide. NumPy arrays do, as we can see in the code segment below.

Applying a conditional expression to a NumPy array returns a boolean array (see line 9). When the boolean array is used as the set of indices for a NumPy array, the elements that correspond to the True value are returned.

```
1 import numpy as np
2
3 # Similar to the range function
4 x = np.arange(1,10)
5
6 divisible_by_3 = x[x%3==0]
7 divisible_by_3_2 = x[(x%3==0) & (x%2==0)]
8
9 print(x%3==0)
10 print(divisible_by_3)
11 print(divisible_by_3_2)
```

```
[False False  True False False  True False False  True]
[3 6 9]
[6]
```

3.5 Expressions

The NumPy array class has methods that overload all arithmetic, relational and bitwise operators of Python. Here is a simple arithmetic expression between two NumPy arrays. The operator applies the operation to every element of both arrays, and returns the result asanother NumPy array.

```
1 import numpy as np
2 x = np.ones(5)
3 y = np.ones(5)
4 print(x+y)
5 print(x+5)
```

```
[2. 2. 2. 2. 2.]
[6. 6. 6. 6. 6.]
```

Note that even when a scalar is used in an arithmetic expression involving a NumPy array, as in line number 5 above, the operation is carried out for every single element of the NumPy array.

3.6 Functions

In addition to these operators, NumPy provides a vast collection of highly optimized functions that can be applied to NumPy arrays. A page that describes each function in more detail can be found here: https://numpy.org/doc/stable/reference/routines.math.html. With respect to mathematical functions, the math module functions we have seen earlier can only be applied to scalars, but the NumPy versions of these functions typically take NumPy arrays as arguments. The code below demonstrates this difference.

```
1 import numpy as np
2 import math
3 x = np.ones(10)
4
5 print(np.sin(x))
6 print([math.sin(i) for i in x])
```

```
[0.84147098 0.84147098 0.84147098 0.84147098 0.84147098 0.84147098
 0.84147098 0.84147098 0.84147098 0.84147098]
[0.8414709848078965, 0.8414709848078965, 0.8414709848078965, 0.8414709848078965, 0.8414709848078
```

In addition you can find `NumPy` functions for doing statistics, random number generation, Fourier transforms, file I/O and linear algebra. When constructing expressions that involve `NumPy` arrays, avoid writing expressions that perform operations at the single element level. Let's say we want to compute the following expression for two arrays x and y. Each of these arrays has 10000 random numbers between 0 and 1.

$$f(\mathsf{x}, \mathsf{y}) = \sum_i \cos(x_i + y_i) + x_i^2 \tag{1}$$

Here is one way to compute the above expression.

```
import numpy as np
x = np.random.random(10000)
y = np.random.random(10000)

f_sum = 0
for a,b in zip(x,y):
    f_sum += np.cos(a+b) + a**2

print(f_sum)
```

```
8308.198621529242
```

Here is another way to compute the above expression. Note that the result won't match exactly in the two cases since we use random numbers.

```
import numpy as np
x = np.random.random(10000)
y = np.random.random(10000)

f_sum = np.sum(np.cos(x+y) + x**2)
print(f_sum)
```

```
8284.42389372466
```

In the second code block above, we took advantage of the NumPy array operations. We did not compute the expression element-wise, rather we computed it in one go. The second method is much faster, as it simply sends the entire data from both arrays to optimized code written in C to perform the calculations. In the first method, we sent two scalars each time to the C code and spent a lot of time fetching back data and adding it to a Python variable, which is much slower than performing the same operation in C.

Using the timeit module we can estimate the gain in speed obtained from array operations. The timeit module requires that we package the statements that need to be timed into a string in triple quotes. Reviewing the output below, notice the huge difference in execution times (60X) between the element-wise and the array-wise operation.

```
import timeit

element_wise = """
import numpy as np
x = np.random.random(10000)
y = np.random.random(10000)

f_sum = 0
for a,b in zip(x,y):
    f_sum += np.cos(a+b) + a**2
"""

print("Element-wise execution time ", \
      timeit.timeit(stmt=element_wise, number=100), " seconds")

array_wise = """
import numpy as np
x = np.random.random(10000)
y = np.random.random(10000)
f_sum = np.sum(np.cos(x+y) + x**2)
"""

print("Array-wise execution time ", \
      timeit.timeit(stmt=array_wise, number=100), " seconds")
```

```
Element-wise execution time  1.7754999349999707  seconds
Array-wise execution time  0.029072588000417454  seconds
```

3.7 The matplotlib Module

This concludes our brief introduction to NumPy. Before ending this writeup, however, we draw your attention to Matplotlib, a powerful and easy-to-use module for plotting

data that is often used with `NumPy`. For examples, tutorials and more information, see https://matplotlib.org/3.1.1/tutorials/index.html.