STAT 479: Machine Learning
Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat479-fs2019/

Fall 2019

Contents

3 Using Python **3**

3.1 Introduction . . . 3

3.1.1 About this Lecture . . . 3

3.1.2 Python . . . 4

3.2 Setup . . . 5

3.2.1 Installing Python . . . 6

3.2.2 Managing Environments . . . 8

3.2.3 Installing and Updating packages . . . 8

3.3 Running Python Code . . . 10

3.3.1 Interpreter/REPL . . . 10

3.3.2 IPython . . . 11

3.3.3 Scripts (.py files) . . . 12

3.3.4 Jupyter Notebooks . . . 13

3.3.5 JupyterLab . . . 13

3.3.6 Jupyter Notebooks and Homework Submissions . . . 14

3.4 Relevant Python Topics . . . 14

3.4.1 Basic Types . . . 14

3.4.2 Operators . . . 14

3.4.3 Strings . . . 15

3.5 Data Structures . . . 15

3.5.1 List . . . 15

3.5.2 Dictionary 15
3.5.3 Set 15
3.5.4 Tuple 15
3.6 Conditionals 15
3.7 Iteration 16
3.7.1 Generators 16
3.7.2 Comprehensions 16
3.8 Functions 16
3.9 Classes 16
3.10 Command Line Arguments via Scripts 16
3.11 Reading and Writing files 16
3.12 Importing Libraries 16
3.13 Standard Library 17
3.14 GIL and Multiprocessing 17
3.15 Subprocesses 17
3.16 Exceptions 17
3.17 Debugging 17
3.18 Resources 17

STAT 479: Machine Learning
Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat479-fs2019/

Fall 2019

3 Using Python

3.1 Introduction

3.1.1 About this Lecture

This lecture provides a brief overview of the Python programming language. While Python is among the programming languages that is easiest to learn – one of the reasons why it has become so popular across different fields – it is impossible to cover all essential aspects of the language adequately within an hour, or, even during the course of this semester (since we primarily want to focus on Machine Learning, not programming!). Hence, it is expected that you already have basic programming knowledge and experience with coding up simple problems, as listed in the course pre-requisites. No worries, even if you haven't done any programming, yet, having worked with R would also be a useful experience.

As mentioned at the beginning of this course and listed on the course website, students who have not used Python before will need to spend a few hours on becoming more familiar with it, by working through one of the recommended resources listed on the course website[1] or other books or courses, which I shared about a month ago via the mailing list. However, if you haven't had a chance to look at one or two of these recommended resources, it is by no means too late to learn the basics for this course (you can easily catch up and learn the basics on a single weekend).

Throughout this lecture, it is important to keep in mind that we are not trying to learn Python from a computer science, programming, or software engineering perspective. In this course, our objective is to use Python as a scientist or researcher with scientific computing needs in mind. People often say that programming is a computer science topic, which I disagree with. Learning a programming language is similar to learning how to use a pen. Moreover, learning how to write words with a pen does not mean that we are automatically novelists. Since many computer science topics are very relevant to machine learning (for example, complexity theory and big-O notation, which we briefly covered the last lecture), it is highly recommended to read an introductory computer science textbook if you would like to pursue machine learning further after this course.

[1] http://stat.wisc.edu/~sraschka/teaching/stat479-fs2019/

That being said, this lecture and the following lectures will provide a quick overview of the relevant topics that you will need for several homework exercises and as a foundation for using the scientific computing libraries for machine learning. Note that this lecture mainly covers the Python language itself, whereas the next lecture will focus more on scientific computing libraries for Python, which we will be using to implement and use various machine learning algorithms covered in this class.

3.1.2 Python

In short, Python[2] is an interpreted, dynamic language that does not require static type declarations. In that sense, while being a multi-purpose programming language, it is more similar to the R language rather than the C programming language[3], which has a static type system.

For example, consider a simple program that prints the string "a+b=c" and the result of the integer division "1+2." If we were to implement that in its simplest way In C, we first have to write a file with the following contents (note the static type declarations):

```
#include <stdio.h>

int main ()
{
   int result;
   char word[6] = "a+b=c";
   result = 1 + 2;
   printf("%s\n", word);
   printf("%d\n", result);
   return 0;
}
```

Then, we would need to compile it into a program and run it. For example, if we saved the file as `example.c`, we could compile it via the GNU compiler tools, GCC [4]:

```
$ gcc -o example example.c
```

Next, we would execute the compiled program, `example` from the command line\footnote{Lines starting with an `$` symbol indicate that we execute a command in a command line terminal, not in a Python interpreter'}:

```
$ ./example
```

Executing the `example` program then produces the following output:

```
a+b=c
3
```

The equivalent program written in Python syntax might look as follows:

[2]The first version of Python was released on 1990 by its creator Guido van Rossum. The origin of the name "Python," despite its current logo, is unrelated to the snake but is derived from Monty Python comedy group and their show, Monty Python's Flying Circus, which was popular in the 1970s and 1980s.

[3]While there are many different Python interpreters out there, the official Python interpreter is itself written in C

[4]https://gcc.gnu.org

```
result = 1 + 2
word = "a+b=c"
print(result)
print(word)
```

We can execute this previous code in an interactive Python interpreter, or we can copy and paste it into a text file (for example, `example.py`), so that we can then run it as a script, for example, by running the following command in your command line terminal:

```
$ python example.py.
```

Some more details about executing Python code will follow later.

Here, the main point of this section is that Python is a dynamic, interpreted programming language. This means that we do not have to specify the types of the variables in Python, and we do not need to compile anything in order to obtain the results. Also, as illustrated above, instead of writing code interactively, we can write the code in a `.py` script file, which we can execute using a Python interpreter (e.g., `python script.py`). But again, no separate compilation step is required. Python, as a dynamic, interpreted language, is very flexible and convenient to use, which makes it especially attractive for scientific computing.

One downside of dynamically typed languages is that they generally perform computations magnitudes slower than statically typed languages. However, in the next lecture, we will work with libraries that implement the more "expensive" computations in C or Fortran code and use Python as a so-called "glue" or "wrapper" language. This way, by calling functions in Python that have been implemented lower-level programming languages, we leverage Python's convenient syntax while using computationally efficient code.

Another downside of dynamic typic is that errors, except for syntax errors, are only raised during runtime. In certain scenarios, this can have important implications. For example, consider the following Python code snippet:

```
if cond:
    text = "abc" + 123
else:
    text = "abc" + "!!!"
```

The line nested under the `if` statement gets executed if a certain condition `cond` is true; otherwise, the line that follows the `else` statement gets executed instead. Now, there is an illegal expression in this code example, `"abc" + 123`: we cannot add an integer-type value to a string-type value. However, if the condition (`cond`) is always false, the code runs just fine because the line will never be executed – no harm done. The worst case scenario is though if `cond` is rarely true because this error occurs very rarely – it may slip through our test suites if we are not very thorough. In contrast, equivalent errors in static-type languages such as C are usually caught by the compiler so that we can fix them before we run, share, "ship," or deploy our code.

3.2 Setup

This section provides an overview of the different ways Python can be installed and set up. Please do not follow these instructions on your first read-through. *Read* through the complete "Setup" section first as it lists several alternative approaches. Once you read the section, you can revisit it and set up Python the way you prefer.

3.2.1 Installing Python

Note that many different operating systems already come with a default Python installation. While the default Python version on macOS is relatively outdated (some old version of Python 2.7), most Linux distributions come with a relatively old version of Python 3.

In general, you can check the pre-installed Python version by executing

```
$ which python
```

in a Linux or macOS/Unix command line terminal (the Windows equivalent is `where python`).

Regardless of which version comes already installed with your operating system, I strongly advise not to tinker with it. Moreover, I highly recommend installing a newer version of Python *separately* on your computer. The reason is that the "built-in" Python version is used by certain processes and services of the operating system, and updating or modifying it is not only cumbersome but is also likely to cause issues for/with your operating system.

Python 2.7 vs. Python 3 For those who are curious why there is a debate: about 10 years ago, Python developers wanted to add substantial improvements to the Python language. However, these improvements would have been backward incompatible. Hence, people would have been forced to rewrite certain sections of their "old" code in order use the latest Python versions, which can be cumbersome if you have developed large code bases. Hence, the Python community decided to branch off and develop Python 3 seperately, while maintaining Python 2.7 with minimal updates (bugfixes, security updates, etc.).

Instead of making the switch to Python 3, many people kept using Python 2.7. However, 10 years after the introduction of Python 3, more people use Python 3 than Python 2.7 (according to most surveys you can find on the internet), and almost all major libraries have been ported to Python 3[5]. In fact, many major libraries are going to drop (or already dropped) Python 2.7 support [6]. Furthermore, Python 2.7 will lose official support by 2020, which means that no updates (no bug fixes, no security updates, etc.) will be made[7] after 2019.

In sum, there is no point in learning Python 2.7 now, and we will be using Python 3 in this class[8]. In particular, I created and tested all the code for this class in Python 3.7, which was released last summer (summer 2018). There is a beta version of Python 3.8 already, but the final version has not been released yet while I am writing this. However, in case you are installing Python 3.8, since Python 3 versions are backwards compatible, the code will also run fine on Python 3.8. The code may even work on Python 3.6 (however, I have not tested it explicitly). Thus, I recommend installing Python 3.7 to be on the safe side.

Windows, Linux, and macOS Unfortunately, I am not very familiar with the Windows operating system; hence, this tutorial is more geared towards Linux and macOS[9]. In general, the Python interpreter should work on Windows in the same way as it does on Linux and macOS/Unix. However, there will be certain differences when working with data files on your computer's storage disk. For instance, Linux and macOS/Unix uses forward slashes

[5]http://py3readiness.org
[6]http://python3statement.org
[7]https://pythonclock.org
[8]In case you are interested, here is an article that covers some of the most relevant differences between Python 2.7 and Python 3: https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
[9]macOS has a Unix core, which inspired the development of Linux as a "free" Unix clone. Hence, these two operating systems are very similar.

("/home/sebastian/my_data_file.txt") Windows uses backslashes ("C:\sebastian\my_data_file.txt").

In general, if you have a computer running Windows that you want to use for scientific computing and want to make your life easier, I recommend installing Linux alongside Windows on your main operating system as a dual-boot environment and use Linux for scientific computing tasks. Linux is much more popular than Windows in the scientific computing community; thus, you will find much more help and tutorials online that are targeted to the Linux and macOS/Unix environments compared to Windows. Also, many libraries and tools require Linux and macOS/Unix operating systems – I am not aware that this is true for any of the libraries we will be using in this class, though.

There's also a program called Cygwin[10], which is a bundle of Linux programs for Windows. Recent Microsoft Windows versions now also support having a Linux subsystem within Windows[11].

Again, if you use Windows on your main computer, you will probably be fine. However, using Linux or macOS/Unix are highly recommend for scientific computing in general – that's what most researchers and scientists ues. If you do not want to install Linux right now (which is understandable, because it takes quite some time, and you have many other things to do), I at least recommend looking into Cygwin, because I think that Cygwin could make your life easier.

The following two paragraph will outline the two main ways for installing Python. Personally, I recommend using Anaconda/Miniconda (I am using it myself, too. And the same is true for most of my colleagues who are using Python for machine learning and deep learning).

python.org The "official" way to install Python is to obtain an official distribution from the python.org website. You can find a Python installer for several different operating systems and versions of Python (I recommend you to use one of the most recent one; right now, the most recent version is 3.7.4) at https://www.python.org/downloads/. If you choose to install Python from there, please follow the instruction provided on the https://www.python.org website.

Anaconda/Miniconda Based on my experience, the most popular and most convenient way to use Python for scientific computing is to use the Anaconda or Miniconda distributions and package managers. (both are free). You can think of Anaconda as an alternative distribution of Python that comes with a package manager (called `conda`), which makes installing scientific packages easier by handling complex dependencies[12].

Anaconda comes with a whole bunch of Python packages pre-installed. It contains most of the packages that we will be using in this class and many additional ones that we do not need. In comparison, Miniconda is a "leaner" distribution that does not come with many packages pre-installed. Since it is smaller and leaner, and it is easy to install packages via the conda manager, Miniconda is my favorite choice.

To use the Anaconda or Miniconda distribution (highly recommended!) instead of the Python distribution, please download the respective installer from the anaconda.com website and follow the instructions there. The Miniconda installer is available from https://docs.conda.io/en/latest/miniconda.html. Since the installation procedure depends on your operating system, it is too complicated to explain it in this document for each possible scenario. However, if you have troubles installing it both I and the TA are happy

[10]https://www.cygwin.com
[11]https://docs.microsoft.com/en-us/windows/wsl/install-win10
[12]Here, dependencies mean that certain packages depend on other packages. And often, only certain versions of these packages are compatible with each other.

to help you with this.

3.2.2 Managing Environments

Another important aspect of using Python is managing environments and packages. This is especially useful if we are working on different projects which each require different Python packages and different versions thereof. Essentially, a virtual environment is like a "container" on your computer that contains only those libraries that are relevant for a given project, and you can only use them if the environment is "active." If you have multiple projects (or take multiple Python-related classes), virtual environments are a powerful organizational tool.

virtualenv The probably most widely used tool for creating and managing Python environments is `virtualenv`. You can find more about `virtualenv` on the website: https://virtualenv.pypa.io/en/stable/.

conda If you are using Anaconda or Miniconda, it is recommended to use the conda package managing tool that comes with it. With conda, we can create and manage virtual environments similar to `virtualenv`, which is the commonly used virtual environment tool for Python. For example, below is some code for creating a virtual environment that we call "stat479" just for this class:

```
$ conda create -n stat479 python=3.7
```

Then, to activate this environment, we execute

```
$ source activate stat479
```

Note that we have to execute `source activate stat479` each time we open a new command line terminal. Otherwise, the default environment will be used.

You can see that a virtual environment is active based on your command line prompt as illustrated below.

Before:

```
sebastian@Sebastians-MacBook-Pro:~$ source activate stat479
```

After:

```
(stat479) sebastian@Sebastians-MacBook-Pro:~$
```

For more information about `conda` and managing virtual environments, please see https://conda.io/docs/user-guide/tasks/manage-environments.html.

3.2.3 Installing and Updating packages

There are two recommended ways for installing Python packages, which will be introduced in the next subsections.

Pip Pip is the official Python package installer. While `pip` is a Python library or package itself, you can directly use it from the command line as a standalone program. For example, to install the NumPy package that we will be using next lecture, you can use the following command:

```
$ pip install numpy
```

A specific version of a package can be specified in the installation command as follows:

```
$ pip install numpy=1.15
```

To upgrade a package that is already installed, use the `--upgrade` flag:

```
$ pip install --upgrade numpy
```

To update `pip` itself, use

```
$ pip install --upgrade pip
```

Uninstalling a package is also simple:

```
$ pip uninstall numpy
```

Note that since `pip` is also a Python package, entering `pip` on the command line terminal is the same as running `python -m pip`, that is, running `pip` as a module via the `-m` flag. This may be useful of your command line terminal does not recognize the `pip` command. For example, NumPy can be installed this way as follows:

```
$ python -m pip install numpy
```

For more information about Pip, please see https://pip.pypa.io/en/stable/ and refer to https://pip.pypa.io/en/stable/installing/ in case you encounter problems with using `pip` on your command line.

Conda If you are using Anaconda or Miniconda, it is highly recommended to use the `conda` package manager to install Python packages instead of `pip`. The Anaconda team provides pre-compiled versions of Python packages to ensure the best compatibility with your environment and operating system, and it handles complex dependencies between different packages if necessary.

The usage is very similar to `pip`. To install the NumPy package that we will be using the next lecture, you can use the following command:

```
$ conda install numpy
```

A specific version of a package can be specified in the installation command as follows:

```
$ conda install numpy=1.15
```

To upgrade a package that is already installed, use the `update` command:

```
$ conda update numpy
```

To update `conda` itself, use

```
$ conda update conda
```

Uninstalling a package is also simple:

```
$ conda uninstall numpy
```

While most major packages for scientific computing are available via `conda`, you may find that you need packages that are not available through the `conda` installer. Note that if you are typing `conda install` , the package is fetched from the official Anaconda website. However, `conda` also allows us to specify channels to download packages from other sources. One of these is the community project `conda-forge`, which provides additional packages for conda that are not available via the official Anaconda channel. One such example is the `mlxtend` package, which we will be using for one of the homework exercises to visualize 2D decision regions of scikit-learn classifiers. To install `mlxtend` from the conda-forge channel, you can use the `-c` (channel) flag as follows:

```
$ conda install mlxtend -c conda-forge
```

Even if you are primarily using `conda`, you can still install packages via `pip`. Below, I listed my recommended order of approaches to try when installing a new package:

1. directly via `conda`;
2. via `conda` from the `conda-forge` channel;
3. using `pip`.

3.3 Running Python Code

There are many different ways how we can execute Python code. The following subsections list some of these. Note that you are welcome to use any approach you prefer, the problem sets (homeworks) will be handed out as Jupyter Notebooks [13]. Also, you are expected to hand in your homework in the form of Jupyter Notebooks + an HTML file created from the Jupyter Notebook. The overall procedure will be discussed in more detail class when I hand out/provide the first problem set.

3.3.1 Interpreter/REPL

The simplest way to use Python is via the so-called “Read-eval-print loop” (REPL). The REPL essentially means that we are executing Python code in an interactive session:

```
$ python
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

[13]Please see the official documentation for more details: https://jupyter-notebook.readthedocs.io/en/stable/

```
>>> print(1 + 2)
3
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Note that in this document, we use the following notational convention:

- Starting a line with a "$" character refers to a command line prompt in a terminal (for example, a Linux or Unix shell).
- An ">>>" at the line start refers to a prompt in a Python interpreter.
- An "..." at the beginning of a line indicates the continuation of the input command that was initiated by the previous >>> prompt.

The REPL is useful if we want to evaluate a small number of expressions, for example, but it is not recommended for doing "heavy lifting," that is, running more extensive code examples or programs.

3.3.2 IPython

IPython stands for "interactive" Python, and using Python over the standard Python REPL has many advantages – for example, the so-called "magics," which are some extra commands for our convenience[14]. Another of my favorite IPython features is that we can use the `Tab` key to autocomplete function and variable names.

IPython can be installed using `conda`. For example,

```
conda install ipython
```

For more information and installation instruction, please refer to the official documentation at https://ipython.org/install.html.

Once installed, we can start an IPython session by evoking the `ipython` command from the command line:

```
$ ipython
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.7.0 -- An enhanced Interactive Python. Type '?' for help.
```

While all default Python language and interpreter features also work in IPython, IPython has a nicer "help" documentation compared to the `help()` function in Python, which we can evoke via the `?` command. For example, if we want to find out more about Python's `sorted()` function, we can simply type `sorted?`, as shown below:

[14] https://ipython.org/ipython-doc/3/interactive/magics.html

```
In [1]: sorted?
Signature: sorted(iterable, /, *, key=None, reverse=False)
Docstring:
Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order and the
the reverse flag can be set to request the result in descending order.
Type:      builtin_function_or_method
```

Magic commands are preceded by an `%` symbol in IPython. One of my favorites is the `%timeit` magic command, which is very useful for "benchmarking" functions:

```
In [11]: def reverse_string_1(my_str):
    ...:     return ''.join(reversed(my_str))

In [12]: def reverse_string_2(my_str):
    ...:     return my_str[::-1]

In [13]: %timeit reverse_string_1(very_long_string)
1.74 ms +- 20.5 microsec per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [14]: %timeit reverse_string_2(very_long_string)
42.4 microsec +- 608 nanosec per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

Also, IPython allows us to use Linux/Unix commands directly within the Python session – if we start a command with "`!`" IPython interprets the command after the `!` as a shell (Linux/Unix) command. For example, `!ls` lists the current subdirectories and files in the current working directory:

```
In [15]: !ls
Creative Cloud Files OneDrive             code
Desktop              Pictures             custom-settings
Documents            Public               miniconda3
Downloads            ...
```

3.3.3 Scripts (.py files)

If we are developing more extensive analyses or programs, we want to keep most of our code in some sort of file, which has certain advantages. For example, if we execute a long series of commands in the REPL or in IPython to produce certain results, it would be cumbersome to reproduce the results if we want to perform the same or a similar analysis again (for example, imagine you want to rerun all the previous code after adding a few extra entries to the dataset). If we kept the code in a file instead, it is easier to

- modify and adjust code;
- backup our code;
- share our code.

If you are creating Python (.py) files, you can, of course, use any text editor you like. However, it is highly recommended to use a text editor that at least offers programming-language specific syntax highlighting to make your life easier. Common and good choices for text editors are Visual Studio Code[15], Atom[16], and Sublime Text[17], for example. My personal preference is Visual Studio Code.

Of course, there are also specific IDEs (Integrated Developer Environments) for Python that provide additional convenience functions. Commonly used Python IDEs are PyCharm[18] and Spyder[19]. However, for many tasks in scientific computing, IDEs are considered "overkill" and would be something to consider for more advanced Python-based software development.

3.3.4 Jupyter Notebooks

Using Jupyter Notebooks for writing code is like using Microsoft Word documents for writing text: it is very convenient if you want to have and view everything in one file.

Jupyter Notebooks are particularly hand for conducting data analyses (because it allows you to add notes, figures, and plots).

Originally, Jupyter Notebook[20] was developed as an interactive document on top of IPython – back then, it was called IPython Notebook[21]. However, over the years, the developers extended the "Notebook" concept to also support other programming languages, such as Julia and R[22].

You can think of Jupyter Notebooks as an interactive environment similar to IPython. However, in addition to having an interactive IPython session, Jupyter Notebooks are also "documents" that allow us to add text, LaTeX equations, figures, and so forth.

The reason why Jupyter notebooks are so popular within the scientific computing community is that they make it easy to save, present, and share a data analysis in a single, executable file.

We discussed how to set up and use notebooks live in class. Since Jupyter notebooks are such an interactive concept, it is probably most effective if you consider a video tutorial as a reference rather than text. For example, Corey Schafer is sharing a good video tutorial for setting up Jupyter Notebook on YouTube at https://www.youtube.com/watch?v=HW29067qVWk.

3.3.5 JupyterLab

While Jupyter Notebook is the "original" application for working with Jupyter notebooks, it is not the only one. Analogously, Microsoft Word is a program to open Word (.doc, .docx) files, but those files can also be opened by other applications (like OpenOffice or LibreOffice).

Recently, a new, officially supported interface for Jupyter notebooks was released called JupyterLab[23]. JupyterLab is a modernized version of Jupyter Notebook that adds some more convenience features on top of it. You are welcome to use it since `.ipynb` files are compatible with both Jupyter Notebook and JupyterLab.

[15] https://code.visualstudio.com
[16] https://atom.io
[17] https://www.sublimetext.com
[18] https://www.jetbrains.com/pycharm/
[19] https://www.spyder-ide.org
[20] https://jupyter-notebook.readthedocs.io/en/stable/
[21] This is the reason why we still use the file ending '.ipynb' for Jupyter notebooks.
[22] The term Jupyter is basically something like an acronym of the terms Julia, Python, and R
[23] https://jupyterlab.readthedocs.io/en/latest/

3.3.6 Jupyter Notebooks and Homework Submissions

As discussed in the lecture, when it comes to computing, we will be mostly working with Jupyter notebooks in this course. You will receive the homework assignments, questions, and starter code as Jupyter notebooks (`.ipynb` files). Also, you are then expected to hand us back the solutions in the form of Jupyter notebooks, which are the original notebook files but modified with your solutions.

We (the TA and I) will be viewing your answers to the homework questions assignments using Jupyter Notebook, too. Also, we will run your code on our computers to make sure that the code you provide in certain assignments actually works. Thus, please make sure that your notebooks can be executed sequentially.

For working on the homework and submitting it, please follow the following steps:

- Do not modify the cells that contain the original question/assignment text or code cells that start with the line "`# DO NOT MODIFY THIS CELL`.
- After you finished working on your notebook, make a copy of the notebook, and click on the "Restart & Run All Cells" button under the "Cells" tab in the menu bar, to check that all your code can be executed in sequential order.
- If everything works as expected, export the notebook as HTML file (click on `File` -> `Download As` -> `HTML (.html)`).
- Send both the `.ipynb` and the `.html` file of the notebook with your homework solutions to Shan Lu (the TA) and CC me on the email.

3.4 Relevant Python Topics

There are many really good Python resources out there as we discussed. Also, since this is a machine learning course, we cannot spend to much time on learning Python in this course. Below is a list of Python concepts that I would consider as most relevant.

If you are already familiar with Python, my recommendation is to read through the list below and check for yourself that the subjects make any sense to you. If the majority of these do, you can read up on individual concepts using the Python documentation, for example.

If most of this is new to you, you should consider spending a few hours working through a Python learning resource – consider these listed on the course website.

Also, consider the excellent, official, and free Python tutorial as a reference resource, as well as the official Python documentation for learning about Python:

- Official Python Tutorial: https://docs.python.org/3/tutorial/index.html.
- Official Python Documentation: https://docs.python.org/3/

3.4.1 Basic Types

- `bool`, `float`, `int`, `str`, . . .

3.4.2 Operators

Arithmetic Operators

- `+`, `-`, `*`, `/`, `//`, `**`, . . .

3.4.3 Strings

Basics

- single quote, double quote, escape characters, strings that span multiple lines
- strings are immutable objects
- string indexing and slicing
- different ways to print a string

Basic string methods

- `.upper()`, `.lower()`, `.replace()`, `.startswith()`, ...

3.5 Data Structures

3.5.1 List

- `list` type
- mutable
- sorting a list
- variable-size
- slicing and indexing

3.5.2 Dictionary

- `dict` type
- key-value pairs
- keys must be immutable types
- fast look-up ("hash table")

3.5.3 Set

- unique values (stores immutable objects)
- fast look-up ("hash table")

3.5.4 Tuple

- like list but fixed size
- comma creates tuple, not the parenthesis

3.6 Conditionals

- if / elif / else

3.7 Iteration

- `while`-loop
- `for`-loop
- useful keywords: `continue`, `break`
- useful objects: `range`, `enum`, `zip`

3.7.1 Generators

- difference between looping over a generator vs. a list

3.7.2 Comprehensions

- list comprehension
- set comprehension
- dictionary comprehension

3.8 Functions

3.9 Classes

3.10 Command Line Arguments via Scripts

- Using `sys`

```
import sys

first_value = sys.argv[1]
second_value = sys.argv[2]

print("A:", first_value)
print("B:", second_value)
```

- more sophisticated command line argument parsing via `argparse` library

3.11 Reading and Writing files

- `f = open('file.txt', 'r')` + `f.close()`
- better: `with open('file.txt', 'r')`
- `r` for read mode, `w` for write mode

3.12 Importing Libraries

- `import numpy`
- `import numpy as np`
- `from numpy import some_function`

3.13 Standard Library

3.14 GIL and Multiprocessing

- GIL = Global Interpreter Lock

3.15 Subprocesses

3.16 Exceptions

3.17 Debugging

3.18 Resources

- Really good, official, and free Python tutorial: https://docs.python.org/3/tutorial/index.html
- PEP8, Python Style Guide: https://www.python.org/dev/peps/pep-0008/
- Additional Python resources you should consider working through if you are new to Python: http://pages.stat.wisc.edu/~sraschka/teaching/stat479-fs2019/#resources