Python Onramp

Stephen D. Boyles

Spring 2021

1 Introduction

This document aims to transition you into Python, assuming that you already have experience programming in another language (e.g., C, Fortran, Matlab). It is not intended to be a full tutorial in and of itself; there are many of those that already exist (see Section 9). Rather, the goal is to point you to some of the distinctive features of Python, similarities and differences with another language you may have learned. I try to highlight the specific differences that arise between Python and other languages you have learned — for instance, if you are mainly familiar with Matlab, pay particular attention whenever I use the word 'Matlab.'

This document is a companion to the Python Tutorial at `http://avinashu.com/tutorial/indexPythonlong.html`, which provides more instructions and examples. I also make a number of references to Learn X in Y Minutes[1] which lists many examples concisely. I recommend having both of these open as you go through this document.

2 Getting a Python environment

For the purposes of CE 367R, there are three options for working with Python:

1. The simplest option is to use Repl.it, which provides a Python environment you can access from any web browser: just browse to `http://repl.it/languages/python3` and start typing. We will be using this environment for the in-class programming activities, to ensure that everybody is using a common interface. You have the option of using it for the assignments in this course as well, and you can register an account for free that will let you save your work.

   However, if you intend to do serious Python programming after this class, or if you are involved in more complex projects, I strongly recommend finding a way to run Python on your own machine. As with any cloud service, **please keep a local copy of your work as a backup** (e.g., downloading your project at regular intervals) in case the service is interrupted or you have connection difficulties.

2. You can also install Python on your own computer. This is the most convenient option if you will be using Python intensively, especially if you have a powerful computer and need to do complex calculations or work with large data sets.

   The downside is that there are many Python environments, and installation instructions depend on your operating system and hardware. I personally use Linux, and most Linux distributions come pre-installed with Python interpreters (type `python3` from a command prompt). For instructions to install the Anaconda environment on Windows, see `http://avinashu.com/tutorial/indexPython.html`. If you have a different operating system, you can search online for instructions.

[1] `https://learnxinyminutes.com/docs/python/`

3. A third option is to use the CAEE virtual desktops, which have the Anaconda environment pre-installed. For instructions on how to access the virtual desktops, see `https://www.caee.utexas.edu/students/itss/43-students/it/386-virtualdesktops`. If you are off-campus, you will need to connect via VPN first; if you need instructions see `https://wikis.utexas.edu/display/engritgpublic/Connecting+to+the+University+of+Texas+VPN`

   Your files should be saved to your Austin Disk allocation (4 GB space) at `\\austin.utexas.edu\disk\engrstu\caee\your_eid_here`. I recommend that you check that this is in fact the case (outside the virtual desktop environment) before logging off. For more information on accessing this storage, see `https://students.engr.utexas.edu/it-resources/data-storage`

Regardless of which option you use, please regularly back up your work so it is not lost. It is a good idea to have your backup be in a different place than your main work. For instance, if you are mainly working on your own computer, storing your backup on Austin Disk, Box, or another cloud server; or if you are mainly working on repl.it or a virtual desktop, storing your backup on your own computer. Keeping your backup on the same hard drive as the main copy is a bad idea, since if the hard drive fails, you've lost both.

The rest of this document assumes that you already have Python up and running. I recommend you open up a Python interpreter so that you can try out things as you read through the document. I also recommend you look through some of the Python code I have already provided to you for Homework 0, to see what "real-life" code looks like and not just artificial examples.

3 Comments, variables, and arithmetic

Two main issues that arise when people are moving to Python from another language:

- All of Python is case-sensitive, unlike Fortran or some parts of Matlab. For example, `variable`, `Variable`, and `VARIABLE` are all treated as different.
- Languages use different ways to show where blocks of code start and end (e.g., what is inside a loop or 'if/else'). For example, languages like C use braces { } to show where code blocks start and end. Matlab and Fortran use `end` statements at the bottom of code blocks.

  Python does not have any special statement or character for this. Instead, **indentation** is used to mark blocks of code: everything which is inside a loop needs to be indented by exactly the same amount. You can use either tabs or spaces to indent, but you have to be consistent. The official Python style guide recommends using 4 spaces to indent, and this is what I use in all of the code I will give you in the class. This is a common source of frustration for new programmers, but it becomes intuitive after a while.

Good code has comments to explain what is going on in an intuitive manner. In Python, there are two ways to comment: the `#` character creates a single-line comment, the equivalent of `//` in C, `!` in Fortran, or `%` in Matlab. Long comments which span several lines are marked with three quotation marks at the start and end. For examples of both of these, see `studentGrades.py` in Homework 0.

Unlike C or Fortran, you do not have to declare variables before using them. Python will automatically detect what kind of variable it is (integer, float, etc.) when it is created. Variable names can only include letters, numbers, and underscores. They cannot start with a number.

The normal operators are used for addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). For exponents, use `**`. If you want to divide two integers and obtain an (integer) quotient and remainder, use `//` and `%`, respectively. For example, 8 divided by 5 is 1 with a remainder of 3; `8 // 5` gives `1`, and `8 % 5` gives `3`.

To display text or the value of a variable, use the `print` statement. For examples, see `http://avinashu.com/tutorial/pythontutorialnew/DataStructures.html`. These are helpful as a quick way to debug code. Work out an example by hand to see what the values of variables should be at each point during the code, and use `print` statements to check that this is true, and to find the first place where things go wrong.

4 Basic data structures

One of the advantages of Python is its built-in data structures for managing groups of variables. C and Fortran have built-in array types, and Matlab has built-in vector matrix types. In Python, the most commonly used data structures are **lists**, **tuples**, **sets**, and **dictionaries**.

If you are used to working with arrays or vectors, the list is the most similar. For examples of lists, search for "Lists store sequences" in Learn X in Y Minutes or see the Lists section in `http://avinashu.com/tutorial/pythontutorialnew/DataStructures.html`. Importantly, it is easy to add or remove elements from a list — this is a major difference from arrays in C or Fortran. One major difference from Matlab is that lists are indexed starting from 0, and not from 1. So if `x` is a list, `x[1]` in Python will give you the *second* element of the list and not the first (which is accessed with `x[0]`). I recommend using lists if you need to add or remove things from a collection, and if the order is important.

A tuple is like a list, except that it cannot be changed after being created (it is "immutable"). For examples, search for "Tuples are like lists but are immutable." in Learn X in Y Minutes or see Tuples and Dictionaries in `http://avinashu.com/tutorial/pythontutorialnew/DataStructures.html` It is helpful if you have a collection of data where the order is important, but that won't be changed.

A set is also like a list in that it collects data, but order doesn't matter (there is no concept of "first" or "second" element), and you can't repeat elements. For examples, search for "Sets store" in Learn X in Y Minutes. I use sets relatively rarely in Python.

A dictionary is a collection of data that associates a "key" with each "value." The name is meant to be suggestive; each word in a dictionary is associated with its definition and other information. In this case, the "key" is the word you look up, and the "value" is the definition that you are trying to find.[2] In Python, dictionaries are often used when you want to associate the values in your collection with some kind of label or keyword. For examples, search for "Dictionaries store mappings from keys to values" in Learn X in Y Minutes or see Tuples and Dictionaries in `http://avinashu.com/tutorial/pythontutorialnew/DataStructures.html`. One potential source of trouble is that dictionary keys must be immutable (unchangeable). So text strings, tuples, and specific numbers work. A list will not.

5 Basic control structures

Every programming language has control structures representing *conditional* behavior (code which is only executed if something is true or false) and *loops* for repeating a certain block of code. As mentioned above, these "blocks" of code are represented in Python by indenting, and Python is strict about your indents being consistent in size and type (don't mix spaces and tabs).

In Python, the main conditional statements are `if`, `else`, and `elif`. If you are coming from C, Fortran, or Matlab, these are analogous to the `if`, `else`, and `else if`/`elseif` constructs you have seen before, but you do not need to put an `end`/`endif`/close brace at the end — indentation handles this in Python. For examples, search for "Here is an if statement" in Learn X in Y Minutes or see "Conditional Statements" in

[2]By the way, did you know that the word "gullible" has been removed from most dictionaries?

`http://avinashu.com/tutorial/pythontutorialnew/PythonBasics.html`. Notice that your conditional statements must end with a colon.

Greater than, less than, and equals are exactly as they are in C, Fortran, or Matlab(`>`, `<`, `==`, `<=`, `>=`). Inequality ("not equals") in Python is written as `!=`. This is the same as in C, but different from Fortran (which uses `/=`) or Matlab (which uses ∼`=`).

As in all of these languages, you need to distinguish a single equal sign (assignment; set the variable on the left hand-side to whatever is on the right) with a double equals sign (comparison; are the left and right hand sides currently the same?) You can combine conditions using `and`, `or`, and `not`, and use parentheses to determine the "order of comparison." Search for "Boolean Operators" in Learn X in Y Minutes to see examples.

Python has two kinds of loops: a `for` loop is used when you know in advance how many times you want to go through the loop, and a `while` loop is used when you want to keep looping until some particular thing occurs, regardless of how many times it takes.

If you know C or Matlab, you have seen `for` loops already. In Fortran, this corresponds to the `do` construct. You do not need an `end` statement at the end of the loop, the indentation marks what code is repeated. Two common uses of `for` loops are (1) do something for every element in a list or dictionary, and (2) do something a given number of times. For example, if `li` is a list, the statement `for x in li:` repeats a block of code once for every element in `li`; and the variable `x` rotates between all the elements of the list, in sequence. In the second case, the `range` statement is commonly used: the statement `for x in range(5, 8):` repeats the loop three times, with `x` taking the values 5, 6, and 7. Be careful: `x` starts at the first value in the range, and stops one before the last one. See some other ways to use the `range` statement by searching "returns an iterable of numbers" in Learn X in Y Minutes.

A `while` loop is simpler, just put a condition (like you would use in an `if` statement) after the word `while` and add a colon at the end. If you know C or Matlab, you have seen the `while` statement already. In Fortran, this corresponds to the `do while` construct. Search "While loops go" in Learn X in Y Minutes, or search "While loops execute" in `http://avinashu.com/tutorial/pythontutorialnew/PythonBasics.html`, for examples.

Loops and ifs can be nested (placed one inside another), using additional indents as needed.

6 Projects with multiple files

Large programs are often divided into multiple files. This improves organization, and makes it easier to find certain pieces of code. To use functions or classes defined in another file, you need to use an `import` statement. This statement is also used to use "libraries" that provide additional functionality beyond basic Python. For instance, `import math` gives you access to math functions for square roots, exponentials, trigonometry, and so on. Search for "You can import modules" in Learn X in Y Minutes for examples of this statement and how to use the imported functions or classes. Notice that if you just type `import math`, then you need to precede each math function with `math` and a period.

7 Functions and building complete programs

To keep code organized, you are encouraged to split long sections of code into smaller functions. Each function should do one specific thing. Functions also help avoid duplication — if you need to do the same thing at several different places in a program, section that code off into a separate function. You can then call that function with a single line of code, and if you have to make any changes you only have to do it in one place. Functions are marked with the `def` statement, followed by the name of the function, its arguments, and a colon; the body of the function is indented.

The best way to get familiar with Python functions is to look at some existing code. Look at the `grader.py` file from Homework 0. This is the file that you will run to check whether your code is working correctly. It does this by comparing the output of your code to what is expected, given the inputs in the `tests` folder. This file has three functions: `runTests`, which manages running all the tests of a given type; `displayScores`, which shows the results at the end, and a special "main" function which is where the program starts. This is indicated by the (admittedly awkward) command `if __name__ == '__main__':`. You will see that the main function calls `runTests` three times — rather than repeat the same commands three times, it's easier to split it off into a separate function.

8 Classes and organizing complex programs

Python is an *object-oriented* language, which means there is yet another way to organize code.[3] An *object* can have multiple variables and functions associated with it; these variables are usually called *attributes*, and these functions are usually called *methods*. To describe a type of object, you create a `class` which provides its attributes and methods. It is common to create a separate file for each class.

Look at the `student.py` file in Homework 0, which defines the `Student` class, and notice the following:

- This class has four methods: an "init" method, a "str" method, `displayStudentCount`, and `examAverage`. Like regular functions, each method is indicated with a `def` and a block of indented code.
- Like functions, methods have a list of arguments in parentheses. *Every method should have `self` as its first argument.*
- The init method defines three attributes for each `Student`: their name, and their scores on two exams. Within the class methods, you refer to these attributes by `self.name`, `self.exam1`, and `self.exam2`. For example, the `examAverage` method computes and returns the average of the two exam scores.
- The init method is what is used to create a `Student` instance. For example, the line of code

  ```
  s = Student("Steve", 70, 80)
  ```

  creates a new `Student` variable `s`, whose `name` is "Steve," and whose `exam1` and `exam2` scores are 70 and 80, respectively. You can access these attributes with `s.name`, `s.exam1`, and `s.exam2`, and can compute the average score with `s.examAverage()`. That is, all of the following statements are true:

  ```
  s.name == "Steve"
  s.exam1 == 70
  s.exam2 == 80
  s.examAverage() == 75
  ```

- The str method is used to create human-readable output you can use with a `print` statement. Continuing the above example, the code `print(s)` would output

  ```
  Student with name Steve and exam scores 70 and 80.
  ```

[3]You may have noticed that a lot of language features are based around organization. This becomes more and more important as you work with larger and more complex programs, especially if you are working with other people. Take advantage of all of these features, it will make your life easier now, and it will make it easier for anyone who might look at your code in the future — even your future self!

Both the init and str method have two underscores before and after their names, to indicate that these are special methods.

- In the above examples, notice that when calling any method — including the init method and `examAverage` — you do not include the `self` argument which appears ih the class file. The init method is called with three arguments, not the four listed in `student.py`; and the `examAverage` method is called with zero arguments, not one. This is a common source of confusion: `self` appears in the code *defining* the class, never in other code which is *using* the class.
- Line 2 of this file introduces a variable called `studentCount`. By defining this variable here, outside of any methods, it creates a *class variable* which is common to *all* `Student` objects you create. Here, the intent is for this variable to store the total number of students. For more on the difference between class and instance variable, see `https://www.digitalocean.com/community/tutorials/understanding-class-and-instance-variables-in-python-3`.

Now, look at the `utstudent.py` file and read its documentation. In line 31, notice that the name of `UTStudent` class is followed by `Student` in parentheses. This is called *inheritance* — the `UTStudent` class includes everything in the `Student` class, along with additional attributes and methods. In this case, the only difference is the addition of an attribute to store a student's EID.

For more information about Python classes, or object-oriented programming in general, see `https://realpython.com/python3-object-oriented-programming/`.

9 Other tips

- Python is a very commonly used language today. A good Web search can answer many of your questions. For example, if you are looking at the `test.py` file and don't know what the `pass` statement does, a quick Google search for "python pass" gives a number of explanations.
- I also encourage you to experiment within Python. If you are curious what something does, or what happens if you change something, just try it out. You aren't going to wreck anything permanently (but as always, make sure you keep a spare copy of any important code so it is easy to revert your changes when you're done experimenting).
- When you are first learning, the easiest way to debug is to scatter `print` statements through your code. Work out an example by hand to figure out what the values of the different variables should be as your code progresses, and use `print` statements to check that this is happening. You can use a "binary search" to quickly narrow down where things go wrong. For instance, if you have 100 lines of code, try `print`-ing values at line 50; if things are wrong, you know there is an error somewhere in the first 50 lines; if not, then the error happens somewhere in the last 50 lines. Wherever the problem is, put another `print` statement in the middle of that block, and repeat again. When you are done debugging, you can either remove the `print` statements (best for final submission to keep your code clean) or "comment them out" by putting a `#` character at the start of the line (useful if you may want to print things again, but eventually this will clutter up the code).
- As you get more comfortable with Python, you may want to look at using a proper "debugger," which has more features for tracing what your code is doing to find errors. For example, Spyder has a debugger. See this tutorial for an example of how to use it: `https://wiki.math.ntnu.no/anaconda/debugging`.
- I also recommend "pair programming," where you work together with a partner. Just like proofreading, it is much easier to see problems in someone else's code than in your own. For this to work, both of you should be looking at the same code at the same time, and working together. Usually one person is typing code, and the other person is making suggestions or looking for errors. Switch roles frequently.