LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 1 INTRODUCTION

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

1. INTRODUCTION TO SOFTWARE DEVELOPMENT IN PYTHON

BY FRED ANNEXSTEIN

In this chapter we will cover the basic tools and terminology used in Python programming. We will first examine installing a Python Programming platform, which is used throughout this book. We then turn to look at the Python interpreter and the dynamic typing that underlies the Python execution model.

CHAPTER 1 OBJECTIVES

By the end of this chapter, you will be able to...

- Install and run the Anaconda version of the Python Interpreter.
- Recognize and describe the Python Interpreter employing common Python terminology.
- Practice running small Python scripts. Understand the Python software development process.
- Understand the Python runtime model including, dynamic typing, statements and expressions, and basic control.

- Understand the syntax and semantics of Python doctests. Formulate and execute a Python program with a variety of doctests.

THE ANACONDA PYTHON DISTRIBUTION

In this book we use the Anaconda Python distribution because it is easy to install on the most platforms such as Windows, macOS and Linux. Anaconda supports the latest versions of Python, the IPython interpreter, and Jupyter Notebooks. Anaconda also includes other software packages, libraries, and tools commonly used in Python programming and data science, allowing students to focus on learning the Python language and computing skills. In this book we will use the Spyder application, which is packaged with Anaconda, as the default software development environment. This environment will allow you to easily experiment and test for your Python programming exercises.

Download the Anaconda3 installer for either Windows, macOS or Linux from:
`https://www.anaconda.com/download/`
When the download completes, run the installer and follow the on-screen

instructions. To ensure that Anaconda runs correctly, do not move any of the files after you install it.

UPDATING ANACONDA

The next task is to ensure that Anaconda is up to date. Open a command-line window on your system as follows:

On macOS or Linux, open a Terminal from the Applications folder's Utilities subfolder.

On Windows, open the Anaconda Prompt from the start menu. Execute the Anaconda Prompt as an administrator by right-clicking, then selecting More > Run as administrator. If you cannot find the Anaconda Prompt in the start menu, simply search for it at the bottom of your screen.

In your system's command-line window, execute the following two commands to update Anaconda's installed packages to their latest versions:

```
conda update conda
conda update --all
```

INSTALLING THE OTHER PACKAGES

Anaconda comes with approximately 300 popular packages, such as NumPy, Matplotlib, Pandas, Regex, SciKit-Learn, Seaborn, and many more. The number of additional packages you'll need to install throughout the book will be small and we'll provide installation instructions as necessary. As you discover new packages, their documentation will explain how to install them. Generally, you should not use the pip program, rather rely on the %conda install command line option to install additional packages.

A CALCULATOR

Let us now use the Python interactive mode to evaluate simple arithmetic expressions. First, open a command-line window on your system: In the command-line window, type python, then press Enter (or Return). You should see text like the following, but this varies by platform and by IPython version:

```
$ python
Python 3.9.12 (main, Jun  1 2022, 06:36:29)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
```

```
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

The text ">>>" is a prompt, indicating that Python is waiting for your input. So now enter text to evaluate expressions:

```
>>>  25 * 4
100

>>> 25 / 4
6.25

>>> 25 // 4
6

>>> 25 %  4
1

>>> 25 **  4
390625
```

Python uses the asterisk (*) for multiplication, forward slash (/) for division, double slash (//) for integer division, percent (%) for modular arithmetic, and double star (**) for exponentiation. Parentheses should be used to force the evaluation order. Numbers with decimal points, like 12.7, 43.5 and 21.75, are called floating-point numbers.

EXITING INTERACTIVE MODE

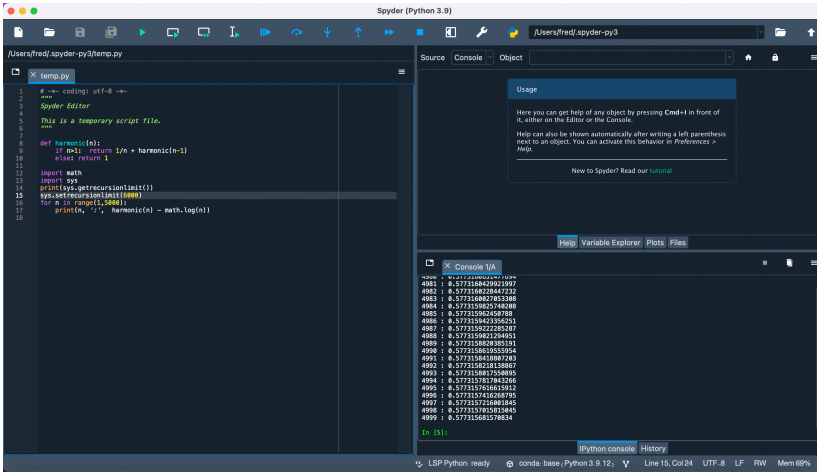
To leave interactive mode, you can: Type the exit command at the current prompt and press Enter to exit immediately.

Type the key sequence <Ctrl> + d (or <control> + d) . This displays the prompt "Do you really want to exit ([y]/n)?". The square brackets around [y] indicate the default response—pressing Enter submits the default response and exits.

Type <Ctrl> + d (or <control> + d) twice (macOS and Linux only).

CREATING SCRIPTS

Typically, you create your Python source code in an editor that enables you to type



text. We will use the Spyder application (which comes with Anaconda), which you can run from the terminal as follows:

```
fred$ spyder
```

The interactive terminal is called a repl for “read eval print loop” - and found in the lower right pane. The editor on the left side pane is used to type and save programs. Spyder provides many tools that support the entire software-development process, such as debuggers. In the upper right pane, Spyder has a set of linting tools for code analysis that detects style issues indicating potential bugs, and can give your code an overall quality score.

THE PYTHON INTERPRETER

Python code is usually saved as a text file with a name like 'lab1.py'. To run the code there is application program installed on your named "ipython". Other installations may call this "python3" or simply "python", and its job is reading and running your Python code line by line. This type of program is called an "interpreter". We will run the interpreter

directly within the Spyder application found in lower right pane.

TALKING TO THE INTERPRETER

We can use the interpreter to look up various information about the programs we are developing, including the formal descriptions built-in python functions. We find that just typing an experiment in the interpreter to see what it does is quick and satisfying way to get the information you need. You should replicate and then play with the examples in the book. Experiments will help you learn the material.

For simplicity we will denote a generic python interpreter prompt with >>>. This prompt will also be used to construct doctests. The symbol means the interpreter is waiting for you to type a line of Python code.

INDENTATION FOR THE INTERPRETER

It is possible to write multi-line indented code in the interpreter. If a line is not finished, such as ending with ":", typing the

return key does not run the code right away. You can write further indented lines. When you enter a blank line, the interpreter runs the whole thing. The python standard is to indent 4 spaces. This indentation should be done automatically when you type return. Hitting return on a blank line ends the block and runs the script. Try this example in your interpreter:

```
>>>  for i in range(5):
   ...:    mult = i * 100
   ...:    print(i, mult)
   ...:
0 0
1 100
2 200
3 300
4 400
```

HELP AT THE COMMAND PROMPT

When you type a question mark ? at the prompt you will get some basic help information. What about Python built-in data types? The built-in string data type known as <str> has many associated function or methods. We can pull down a list of these

function directly from the interpreter as follows.

```
>>> dir(str)
['__add__',
 '__class__',
 '__contains__',,....
```

With the built-in help function, you can pull down the docs for each function. To refer to a function associated with the str type, we use the form str.find. Note that the function name is not followed by parenthesis.

```
>>> help(str.find)
Help on method_descriptor:

find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring
sub is found,such that sub is contained within
S[start:end].  Optional arguments start and end
are interpreted as in slice notation. Return -1
on failure.
```

USAGE OF NAMES

Variables and simply *names* in Python refer to values which are python objects. Names are like pointers in that they don't refer to specific locations in memory. Assignment

statements are used to make names refer to particular values. Many names can refer to one value or object, and in such a case are called *aliases*. Assignment statements never copy data, since they only reassign names to new values. Users of Python should be consciously aware when a name refers to *mutable or immutable data*. When a name refers to a value that is mutable -- the value can be changed without changing the name which is an object reference. Therefore all names that refer to that object will reflect the change to the value -- updates are therefore visible via all aliased names. When an object is immutable, no visible change is possible. Errors are raised when attempts at modifying an immutable value are executed.

PYTHON PACKAGE TERMINOLOGY

Software is complex and spans many layers of abstraction. Python simplifies these abstraction layers. When speaking about Python programming we say: Applications are collections of programs --- Programs are collections of modules --- Modules are collections of statements --- Statements are collections of expressions --- Expressions

create new objects or process existing objects.

Modules are the basic unit of code reusability in Python, and represents a block of code that may be imported by some other program. Three types of modules concern us here: pure Python modules, extension modules, and packages.

A pure module is a program that is a single .py file. An extention module may be written in the low-level language (e.g. C) of the Python implementation. A package is a special module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file named __init__.py.

PYTHON AND BIG NUMBERS

Python has the ability to handle very large numbers, both integers and decimal numbers. In Python, the maximum value for an integer and the number of decimal digits is essentially unlimited. This means you can perform arithmetic operations on numbers, as long as you have enough memory to store them.

Python supports different numerical types –

int (signed integers) – They are often called just integers or ints. They are positive or negative whole numbers with no decimal point. Integers in Python are of unlimited size and there is no special 'long integer' type such as found in C++.

float (floating point real values) – Also called floats, they represent real numbers and are written with a decimal point which splits the integer and the fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (for example, 2.5e2 = 2.5 x 10^2 = 250). Typically, the default is for floats to have about 16 decimal digits of precision.

Consider the following number with many decimal digits and equal to 10^400 (that is 1 with 400 zeros) stored as an Python integer and float.

```
>>> print(int(10**400))
100000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
00000000000000000
```

```
>>> print(float(10**400))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

Oh no. Floats can cause overflow errors, even for integer values. Python has specialized libraries that are optimized for large-scale numerical computing. Let us now look at using one such library, called mpmath, for handling floats with unlimited precision.

IMPORT AND FROM

Generally, when you want to add a package you will prefer to use the import command.

```
>>> import mpmath
```

Most Python programs are composed of multiple module files linked together by import statements, and each module has attributes -- a collection of variable names—usually called a namespace.

Importing everything can be convenient and interactive, but be careful when mixing with other libraries. To avoid inadvertently overriding other functions or objects,

explicitly, it is usually best to import only the needed objects. We are importing the module mpmath which is a library for high-precision decimal numbers. We can import the entire module, and access the sin and cos methods, which are defined in mpmath, as follows:

```
>>> import mpmath
>>> y = mpmath.sin(1)
>>> x = mpmath.cos(1)
```

Mpmath uses global values for working with precision; that is, it does not keep track of the precision or accuracy of individual numbers. Performing an arithmetic operation or calling mpf() rounds the result to the current working precision. The working precision is controlled by a context object called mp, which we can view as follows:

```
>>> from mpmath import mp
>>> print(mp)
Mpmath settings:
  mp.prec = 53                [default: 53]
  mp.dps = 15                 [default: 15]
  mp.trap_complex = False     [default: False]
```

This shows that the default precision is 15 decimal digits. We can change this setting the number of digits of precision or accuracy, and produce a number requiring many digits, before repeating, as follows:

```
>>> mp.dps = 400
```

Here we are setting the precision to 400 digits. Let us try the example from above, which failed before:

```
>>> mp.mpf(10**400)
mpf(‘1000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
0000000000000000000000.0')
```

Now it works. Python can handle number objects with unlimited number of decimal digits with the mpmath library. Let us stress test it with the following command to compute the decimal fraction 1/(999**2). If you run the expression the interpreter should produce an interesting result with 3000 digits.

```
>>> mp.dps = 3000
>>> mp.mpf(1) / mp.mpf(999**2)
```

PYTHON WORKFLOWS: DEBUGGING AND TESTING

Much of the effort in programming is focused on testing and debugging code, which involves learning to interpret errors and diagnose the cause of unexpected errors. Dividing by zero is not allowed and results in the raising of an exception—which is a sign that a problem occurred:

```
>>> 1/(1-1)
Traceback (most recent call last):
  Input In [] in <cell line: 1>
    1/(1-1)
ZeroDivisionError: division by zero
```

In the above example the execution of the interpreter is said to *raise an exception* of type ZeroDivisionError. When an exception is raised in Python it will terminate the snippet, then display the exception's traceback, and finally shows the next prompt so that you can input the next snippet.

Learning and working with the exception tracebacks that Python generates is good

practice for debugging. Here are some more guiding principles.

Test incrementally: Every well-written program is composed of small, modular components that can be tested individually. Try to test everything you write as soon as possible to identify problems early and gain confidence in your components.

Isolate errors: An error in the output of a statement can typically be attributed to a particular modular component. When trying to diagnose a problem, trace the error to the smallest fragment of code you can before trying to correct it.

Check your assumptions: Interpreters do carry out your instructions to the letter — no more and no less. Their output is unexpected when the behavior of some code does not match what the programmer believes (or assumes) that behavior to be. Know your assumptions, then focus your debugging effort on verifying that your assumptions actually hold.

Consult others: You are not alone! If you don't understand an error message, ask a friend, instructor, or search engine. If you

have isolated an error, but can't figure out how to correct it, ask someone else to take a look. A lot of valuable programming knowledge is shared in the process of group problem solving.

In this book we will use the process of test driven development (TDD), which is simply software development process that puts emphasis on clearly documented code tests. Your goal is to place readable tests right into your everyday coding practice. For this course we will use a python package called the *doctest module*. This module allows us to run testing comments just like we run code.

COMMENTS IN PYTHON

All comments begin with a hash mark (#) and continue to the end of the line. Because comments do not execute, when you run a program you will not see any indication of the comment there.

Comments are in the source code for humans to read, not for computers to execute. Python does not have a syntax for multi line comments, however your IDE should allow insertion and removal of block

comments. Strings can also be used for comments. If we do not assign strings to any variable name, then they play a role and act as comments.

DOCSTRINGS IN PYTHON

Docstrings are simply strings that occur as the first statement in a module, function, class, or method definition. Any such string is called a docstring and becomes the __doc__ special attribute of that object.

There is a standard python module called “doctest” that is useful for setting up a TDD testing framework. The doctest module searches for pieces of text that look like interactive Python sessions inside of the documentation parts of a module, and then executes (or re-executes) the commands of those sessions to verify that they work exactly as shown, i.e. that the same results can be achieved. In other words: The comments or help text of the module is parsed and run through the python interpreter sessions. These slices of example code are run and the results are compared against expected values indicated in the tests.

HOW TO USE DOCTESTS

To use the “doctest module” it has to be first imported. The part of an interactive sessions with the example test code and the associated expected outputs must be copied inside of the docstring of the corresponding function.

SAMPLE RUN OF DOCTEST

Here is a small function with several doctests. Each test checks whether the code for the factorial function produces expected results. Try to load this code and run it to see the output of the doctests. The last if statement is boilerplate code, and the typical way we will execute the doctests when running any code.

```
def factorial(n):
    """Return the factorial of n >= 0.
    Here are three doctests:
    >>> factorial(3)
    6
    >>> factorial(30)
    265252859812191058636308480000000
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    """

    f = 1
    for i in range(2,n+1):
        f *= i
```

```
    return f

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

CHAPTER 1 EXERCISES

1. Copy the code with the three doctests from above, and paste and load it into the Spyder IDE (left pane). Run the code to show the results of passing doctests. Your output should be as follows:

```
Trying:
    factorial(3)
Expecting:
    6
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.factorial
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

2. Download the expanded doctest example from the following website: https://repl.it/@FredAnnexstein/doctest-example#main.py.

Consider this code and see that it contains examples with logic to raise exceptions based on potential input values to the function. Exceptions may be raised by the system, as in the ZeroDivisionError example above, or the user can raise exceptions based on the value of an object. For example, consider:

```
if math.floor(n) != n:
        raise ValueError("n must be exact integer”)
```

The code snippet above is used to test if a value n is an integer or not. If it is not, then an exception is raised as a ValueError. We will in a subsequent chapter that when an exception is raised it can be handled in a way that does not terminate the processing.

Modify the factorial() function by adding a snippet of code, such as

```
if “n is very large”:
        raise ValueError("n must not be so large!”)
```

so that when n is very large a ValueError exception is raised. You should determine the logic of the “n is very large” trigger

condition. Run the doctest to test your results.

3. From Canvas download the the folder for the Lab 1 on Expressions and Control. Extract the files in the folder if needed and open the index.html file in you browser. The Lab sheet should be formatted with a Red Banner on top. Complete the worksheet.

4. Copy the code for Lab 1 Required Questions. Your submission for Lab 1 will be a modification of the file. You should write an implementation that passes all of the doctests. Once your code passes all the doctests, you should upload this and be confident you will receive full marks.