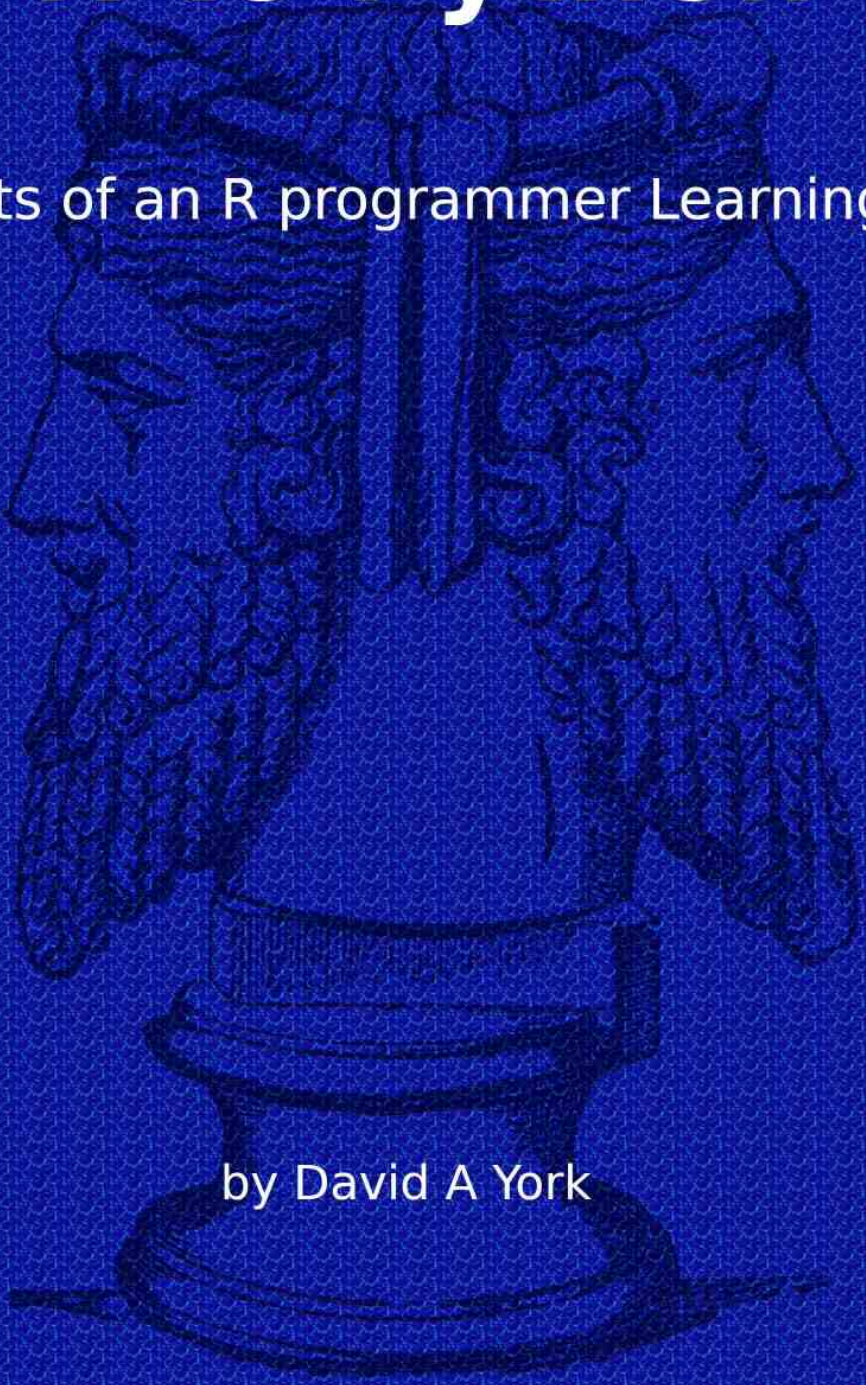
R to Python

Thoughts of an R programmer Learning Python



by David A York

R to Python

Contents

I Thoughts of an R programmer Learning Python **5**

1 Front Material **7**
1.1 Colophon . . . 7
1.2 Preface . . . 7

2 Introduction **11**

II R and Python, Side by Side **13**

3 A Comparative Look **15**
3.1 Functional Programming Commonalities. . . . 15
3.2 Building Blocks of Functions as Procedures . . . 16

4 Set-up for R and Python Exercises **19**
4.1 Package Libraries . . . 19
4.2 Pulling it Together . . . 20
4.3 Using R and Python Together . . . 21

5 Basic Mathematics in R and Python **25**
5.1 Add, Subtract, Multiply and Divide . . . 25
5.2 Other basic algebraic operators in R and Python . . . 26

6 Functional Programming with R and Python **29**
6.1 Defining Functions . . . 29
6.2 Calling and Using Functions . . . 30
6.3 Writing Function Examples in R and Python . . . 30
6.4 Using the Core R and Python Standard Libraries . . . 32

7 Object-Oriented Programming with R and Python **37**
7.1 Defining Classes in R and Python . . . 38
7.2 Using Objects and Classes . . . 40

III Data Science Topics in Python Compared to R **43**

8 Clean and Tidy Data **45**
8.1 Reproducibility . . . 45
8.2 R Data Munging . . . 45
8.3 Python Data Munging . . . 45

9 Using Probability Distributions with R and Python **47**
9.1 Basic Probability Issues . . . 47

9.2 Using the Distrbutions . . . 47
9.3 Other Libraries with Probability and Statistical Packages . . . 47

10 Descriptive Statistics and Data Exploration 49
10.1 Defining Functions . . . 49
10.2 Calling and Using Functions . . . 49
10.3 The Core or Standard Libraries . . . 49

11 Statistical Analysis and Modeling 51
11.1 Defining the Available Functions . . . 51
11.2 Calling and Using Functions . . . 51

12 Non-Stochastic Models 53

13 Reproducible Research 55

14 References 57

15 Bibliography 59

IV Appendices 61

16 Appendix 1 63
16.1 Comparative Syntax for Programming Constructs of R and Python . . . 63
16.2 Extended Structures . . . 63

17 Appendix 2 65
17.1 Mixed R and Python Examples with rMarkdown and Jupyter-SOS Notebooks . . . 65

18 Appendix 3, R and Python Packages 67
18.1 R Core Package List . . . 67
18.2 Python Standard Library[2] List . . . 68
18.3 Other Package Sources (Third Party) . . . 77

19 Appendix 4 Selected Tutorials and Learning Resources 79

Part I

Thoughts of an R programmer Learning Python

Chapter 1

Front Material

1.1 Colophon

R to Python

Thoughts of an R programmer Learning Python

by David A York

http://crunches-data.appspot.com/contact.html

self-published, on Toth-York Imprint, Calhoun GA USA, November 27, 2018

https://github.com/medmatix/RPythonBook

> "In ancient Roman religion and myth, Janus (/ d e nəs/; Latin: IANVS (Iānus), pronounced [ja .nus]) is the god of beginnings, gates, transitions, time, duality, doorways,[1] passages, and endings. He is usually depicted as having two faces, since he looks to the future and to the past."
>
> from Wikipedia, https://en.wikipedia.org/wiki/Janus, accessed 27 November, 2018 at 4:17PM

1.2 Preface

This book is a work in progress on the thoughts of an R programmer (the author) moving to (or learning) Python. It is intended as a comparison of R and Python for those already using R. The hop is to ease the way for those in particular newer to programming as opposed to the quick task oriented scripting which R lends itself so well to.

As with any language the most visible presence on line are the hardened champions of the languages with the pressure to standardization (not a bad thing per-se) and dogmatic adherence to the "blah blah" way of doing things (which is less desirable I think). For those of us who are not playing at being computer scientist but just scientists with a need for tools to do our work mose effectively an approachable treatment of scripting and programming is helpful. There are better computer science texts and better programming texts than this book would ever purport to be. I will often be reminding the reader that this is not a beginning programming book. Neither is it a Computer Science text book. It is a book by a applied programmers for applied programmers who operate in other research domains.

Figure 1.1: Janus

Having started as an R user, then as an R programmer, I have relatively recently embarked on learning python as well. Why I would do this is rather complicated to explain, particularly as I had knowledge of other general programming languages, which would conceivably serve such a role, particularly BASIC and Java. Suffice to say that as a budding Data Scientist I see lots of reasons to know both R and Python. BASIC, though still extant it is very limited in support and relevant libraries by comparison. Java . . . well, as a strong open source advocate I feel Java is bound to Oracle in many ways for all time. It is likely they have no intention of truly releasing it into the public domain, ever. There is far more rationel to learn C++ these days than Java for the sake of what is arguably only minor difference in learning curve.

I absolutely love R! When I returned to school to study Applied Math and Statistics it was an ever present partner for me. It could do most things I needed, though matrix math was less smooth than Matlab the price was right (not withstanding Octave or Scilab). However, I increasingly found the need for a general purpose programming language and Python was there growing in the guise of 'the' data science alternative.

But while R was great with vectorized functions python, at it's core, was weak in this regard. The libraries for python have been growing steadily in number and variety, and the functionality gap between it and R narrows in cluding for vectorized application. I still cannot see a Data Scientist being as productive not knowing R at this point. However, I see no reason not to also know Python - ergo, I think any serious Data Science needs to know both; AND, there is the added incentive of the interoperability of R and Python (and C++)! From R we have XRPython and reticulate and from Python we have Rpy2.

There are far more choices for languages to do Data Science with than I am dealing with here. Arguably, at time point, R and Python seem the most suitable overall to "out of the gate" data analysis. Julia is comming on strong, and C++ is everyones speed oriented fall back. However speed is loosing place in the decision criteria as benchmark comparisons increasingly are show. Most new packages are written in C or C++ anyway and Cython makes this fairly approachable for python package developers. Bottlenecks in Big Data analysis are rarely CPU limited where out of RAM processing, large files and disk access become the limiting factors.

The Organization of the book is in 3 parts. A close Comparison and discussion of R and Python unburdened by a mandate to comprehensively teach beginning programming in either. The first is the overviews of part 1, I only touch quite generally on the nature of the usual The procedural elements of programming structures, Variables, datatypes and data structures, binary decisions, repeating code blocks are all internals in declarative or functional programming - they are just details. The Same can be said for languages introducing object-oriented programming models.

In the second part of the book, an opportunity is taken to compare how data science can be done in R and Python in some of the various domains we work. The distinction between function and object is quite blurred here.

In part 3, a collection of syntax summaries, library reference materials and resources are compiled in Appendices.

In a sense one could consider this book as a second or third source for applied R users learning python - perhaps after the beginner and intermediate level courses to get the necessary python syntax vocabulary and programming structures.

Chapter 2

Introduction

This is not a book on learning to program with R or python. Consider it the companion for non-programmers who are learning python with some knowledge of R.

Learning a new programming or scripting language begs comparison. Often this is not overly helpful to the learner who, coming from a zone of comfort is often frustrated by the old subtleties not consciously recognized, being yet unknown in the new. I recall a family member, completely new to computers, saying why doesn't it know what I mean expecting the implied parts of language to transfer from English (or German, or Chinese etc.) to computer language.

R writers early on recognize the absence of vectorization of functions in core python. It's available though in numpy and pandas, but base python doesn't have it and they miss it. You'll get there, really. Object Orientation is also not natural to either R or Python. It came later to both, but it seems more smoothly accomplished by python than R.

I soon realized, though that you had to know the programming commonalities of python first; R comparisons really should start with the libraries of python. R in someways has cut right to the data science chase often at the expense of strong programming constructs. This is the nod to interactive coding and scripting rather than the need to undertake formal programming. Python began early on with a scripting functionality but it wasn't until the development of iPython that interactive python could be realized in the way that R users were used to.

All this is not to say that R was lacking in basic programming constructs from the start, it wasn't; but the proportion of users who used R in a quick-and-dirty small problem focused way was quite high. This suited it's use as a learning environment for introductory statistics[7]. I should think that those who try to start python at the same time as their first statistics course will find the learning curve too steep to help with their need to learn statistics.

Dalgaard[7] was the first source I used to learn R during my second semester of Statistics. I similarly used Lutz[8] to learn Python well after I was comfortable with R and Probability and Statistics. One needs to get past the programming knowledge of python to affectively get up to speed with matrix algebra, probability and statistics and data munging using the available libraries outside of the basic and Standard Library[2] functions of python. There are books to help you cut some corners in applying python to data science[9, 10], but it's very easy to find oneself confused without a good level of comfort with the ideas and the pitfalls of importing modules in python. Namespaces and variable visibility are less forgiving in python than R.

Once you get comfortable with the math and stats functions available in numpy and/or scipy modules, the data structures and munging functions of Panda and the grphics and plotting functions of Matplotlib you are up to the speed you had gotten to much earlier with R.

As a final word, it may seem like I distain style, in favor of substance. I recall in a list of interview questions somewhere whether software that worked was more important than well documented code or vis-a-versa.

It is a trick question. Though software that doesn't work is useless, it may be worse if it is unfixable too. You can't debug what is not understandable and spagetti code that is undocumented and disappearing into thin air never to be seen again is impossible to work on. Not only will others who want to adapt or debug our old code be lost, but given a month away from it, we probably won't know whatwe mwere doing either. Comment obscessively from the start.

There is a plethora of free open source code and applications out there. Poor documention for users is lazy and sloppy and betrays our responsibility to tell our clients how to use the solutions we offer. Sometimes those clients are us and we deserve better too. Writing user documentation helps us think about our projects in a new perspective - making our work more valuable.

Reproducibility is a necessity for any modern endeavor. In the face of the reseach fraud resurfacing more and more, we can no longer ask people to "just trust us". Data Scientists must know this from the outset. markdown with integrated code, Rstudio and Jupyter addresses this issue superbly well and so their use is central to what I am writing. The fact that they allow multi-language processing in the documents is the bonus. See the References and Resources at the end of the book for other books and articles on Documentation, Style and Reproducibility.

So now we can forge ahead and look at R and python side-by-side and perhaps reduce the learning curve for python Data Science libraries.

Part II

R and Python, Side by Side

Chapter 3

A Comparative Look

As already mentioned, it is not the intent of this treatise to teach either beginning R or python scripting. There are many good sources for this and yet another beginners book or tutorial I believe is not needed. For ease of reference, I include in the Appendix1 an overview of relevant R and python syntax and the general programming constructs common to any functional language. In Part 1, at the risk of being pedantic, I am giving a generic overview of the components of the typical script or program to be used as a framework in organizing our R and Python comparisons.

3.1 Functional Programming Commonalities.

In functional programming, a kind of declarative paradigm, operational code is incorporated by task and purpose into blocks of code which are called by name in the body of a script. In practice it is most often a hybride rather than any one pure paradigm incolved. As a result some connecting statements and assignments in the program of script set up a milieu in which the functions operate together to accomplish some larger task.

Historically, Fortran and Basic code in common academic and educational use, became unwieldy as users became increasingly prone to the writing of rampant spagetti code. The introduction of the concept of 'structured programming' and the uniform reliance on subroutine and function calls mostly tamed the spagetti-monster within a procedural programming paradigm. Growth of functional programming with pascal and to some extent modern C was a natural evolution of this trend and object oriented extensions to most legacy languages exist as we'll see later on.

So the perspective of R and Python as relying on script collections of functions calling each other is what we are stressing now. Purely Functional programs use functions and flow is linear through the collection which makes up the program. In a strict sense this would be done without regard to flow control, but functions are just in a sense subprograms with all the statements of procedural tasks and flow control within, common to the usual conception of programs.

I will deal, in a general way with the, statement types which make up the functions and include comparsions between how R and Python each express the tasks. As I've already stressed,this book not aimed at teaching beginning programming. Consult the bibliography for some suggested sources for beginning programmers^7, 8^.

The later innovation of object oriented programming concepts evolving from this will be touched on in a comparative fashion in another chapter.

3.2 Building Blocks of Functions as Procedures

3.2.1 Statements and Expressions

A program or script is of course considered as a series of instructions grouped together and run as a unit. These statements constitute expressions of mathemetical equations (see specifically Chapter 5), choices, and other orders to be carried out by the machine.

With any program or script statements tasks and values stored are executed or accessed from memory locations. This was the innovation conceived of by John Von Neumann which moved programming from the moving wire jumpers to the input to the machine from various codes punched into tapes or cards to be held in memory while being used.

3.2.2 Assignment of Values to Memory and the Variable

Variables are memory locations assigned values as the computing process goes on. The classification of specific locations determines a type of data held there.

The codes stored in the machine include a specific operator code to direct the move of a value received for input or a statement's result to a memory location. This is called and assignment.

Traditionally R has used the combined dash-less-than symbols '->' for this purpose but the simple equals sign '=' now also works in this fashion. Python uses the same equals sign for assignments.

Variables are named in both R and Python and neither may begin with a digit. There are other restrictions and conventions which are not gone into here.

3.2.3 Decisions and Choices

As currently implemented all choices by a computer resolve to yes or no results. Some semblance or the greys we think in are a result of a collection or range of yes-no questions. These choices are explicit or implicit 'if' queries.

3.2.4 Doing it Over Again (and again ...)

Loops are created within a program logic, usually involving a choice test to be executed which cause tasks to be repeated zero or more times. It is considered bad parctice to create a loop without some form of exit (short of having to power down the machine).

3.2.5 Functions

Functions in fortran were short program clips that did a task and immediately returned a value . Subroutines are more like what functions in R and Python are, though returning to the calling place with some value is a frequent option with these functions as well. Functions may provide a new programming element like a square root (python, math.sqrt(x)) extending the language, or functions may carry out a whole task like carring out a liner regression, storing objects of the answer in memory for retrieval, or return as an object explicitly. Often a function is called that builds a whole graphic plot for display or other output when called.

In R,

```
function.name <- function(arguments)
{
    computations on the arguments
```

```
    maybe other code too
}
```

and in Python indents take the place of brackets to define code blocks. Self in the function spec is the calling code block, and if omitted, is created by python implicitly. In the functional paradigm, self is always present but implicitly. It becomes important in object-oriented programming in python where, your first argument implicitly or explicitly is the calling namespace. 'Self' in the indented code block would be the function itself.

```
def aroutine(args):
    computations on args
    maybe other code too
```

The notation of R is C (and Java) - like with bracketed blocks, and Python is Pascal-like with indented blocks. Unlike pascal there is no 'begin:' keyword required to start the script but when named code blocks are used such as functions the colon is needed at the end of definition.

Functional programming with R and Python, will be expanded on in Chapter 6 later on.

3.2.6 Objects and Classes anticipating Object-Oriented Programming

In general terms we will come to consider all program elements as objects. A Class will be considered as a pattern definition for an object in the same fashion as a function definition. All objects are used as instances of some previously defined class created by an assignment statement.

```
AnInstance=ClassName() # call ClassName constructor to create an instance "AnInstance"
```

More later in Chapter 12

Chapter 4

Set-up for R and Python Exercises

Clearly you have to get R and python onto your system. I have a bias about dependance on any en block installed software stacks, like Anaconda etc. Anaconda is a convenient way to install programs together. However it is easy to loose locations of programs and tools and have conflicts with installs from outside of the group installers, such as different Python destributions, and difficult to manage path complexity.

Just the same, Anaconda is a good choice if you are doing a clean install and intend to do everthing inside that framework.Then installing Anaconda,Python Data Science Platform can make things much easier in the long run for integrating R and Python and Reproducible Research work.

All of these sites will help you approach installs and configurations in a variety of ways. As an R programmer, first I installed R, then RStudio[4]. Once installed, RStudio is a convenient way to install the R packages as well.

Then I installed Python 3 from Python Software Foundation. From that point on I used PIP package manager for python and RGUI for package installs for python and R respectively and install jupyter see Project Jupyter. This gave the maximum flexibility for R and Python in their own environments. I also installed the SciPy (which includes numpy), Pandas, and Matplotlib all using PIP.

I did do a parallel Anaconda install afterwards with R and RStudio and iPython for Jupyter Notebooks but as mentioned there was some path confusion with this approach. It does set you up nicely for any approach to data science and development you could choose down the road.

You really are going to need an IDE for Python. Spyder, available separately installed with PIP, or installed through Anaconda. It is a solid python focused choice. I personally use Eclipse as it is open sourced and you can use it for multiple languages including R (not as strong yet) and python, HTML/CSS/javascript, C/C++, Java and others.

4.1 Package Libraries

The strength of both R and Python is from their libraries. CRAN Package Repository and Bioconnector are both well stocked with general and special purposed functions for R. RStudio develops R Packages integrating well with the RStudio environment. The are lost of R packages in development awaiting addition to CRAN or hosted outside of CRAN such as on Github.

If the reader has spent any time with R at all you have likely made inroads into these package sources. So how does Python compare? Python comes with a large standard library providing “out of the gate” functionality to compare with R-core. The best handing of the intricacies of the python standard library I’ve found is Doug Helmann’s The Python 3 Standard Library by example. Python Software foundation hosts the Python Package Index , PyPi analygous to the CRAN repositories and Bioconnector. Also, the most

important resource to get you up to speed is SciPy tools consisting of SciPy, Numpy, Pandas and Matplotlib. These are the essential minimum to do any R-like work in python. These and more can be accessed as well through the Numfocus Projects. Finally, you should check out the StatsModels[11] package site.

4.2 Pulling it Together

Having apprised ourselves of the breadth and availability of these resources we will go forward and organize some of the ideas. R packages can be installed with the R GUI but the RStudio environment is much easier for this. Once a package is installed it is available in all environments I've alluded to. Compiling and installing is always best and I thing RStudio makes this fairly painless. Remember to start the GUI, Anaconda or RStudio environment as administrator to keep things together.

4.2.1 The Prompts

When accessing the R and Python interpreters from the command shell (or bash) prompt,

```
$
```

after the version and copyright banners, if present, you are at an interpreter prompt. For R this is:

```
>
```

for python,

```
>>>
```

for iPython

```
In [1]:
```

from this point you enter what ever commands or statements you wish to execute. These are executed immediately, or continued with continuation prompts until executable. Once executed any declarations and output are retained if the were assigned to some object to be held in memory. Immediate answers are generally lost.

Otherwise, one groups statements into a file to be read all at once and executed as a group. Objects created are again retained in memory by default. This is a script of program. Script file extensions are standardized for the operating system to recognize, eg.(.R for R, and .py or pyc for python). Python scripts intended to be called by other scripts for library code to be incorporated are called modules and end in the same .py extension. A package can be considered, in both R and python) as a collection of declarations, functions, classes, or modules for library purposes.

4.2.2 Installing Packages

From the R GUI prompt we have,

```
package-install()
```

Python packages from any source mentioned above can be installed with the PIP utility. This can be done from the shell prompt (remember to start as administrator or root):

```
$ pip install matplotlib
  # OR
$ python -m pip install matplotlib
```

This can be done as root with the anaconda prompt as well.

4.2.3 Using Packages

In R we must make sure the package (once installed) is loaded for our session. This is done with the library() function,

```
library("reticulate")
```

while in python we import the package previously installed as,

```
import numpy
```

Once imported or loaded into either, we can access the functions of reticulate directly by name. If we have a previous function of the same name it gets overwritten unless in python we keep the name spaces separate. This can be done by the form of the import statement. We can also import only slected functions from a package module.

```
import numpy as np             # usual format to use namespace segregation of functions
from numpy import sum, matrix  # economical import,still can conflict without renaming
                               #   using 'as', such as where base python has a sum()
                               #   function and math has a different one called sum()
```

4.3 Using R and Python Together

There are several ways that one could find helpful to use R and Python together, to take best advantage of their respective strengths

4.3.1 Working in R and Calling Python

In this instance one would find working in R the necessary starting place and find the need to employ python functions or packages helpful to your needs. A call would be made to the python function from R to have a task completed by python and control returned to the R program.

```
cat("PI in R is", pi)
```

```
## PI in R is 3.141593
```

```
import math
print(math.pi)
```

```
## 3.141592653589793
```

4.3.2 Working in Python and Calling R

Working in python and calling an R object would look like,

```
# Accessing an R object from from python
# (example from: http://rpy.sourceforge.net/rpy2/doc-dev/html/introduction.html)
import rpy2.robjects as robjects
rpi = robjects.r['pi']
print("PI from R = ", rpi)
```

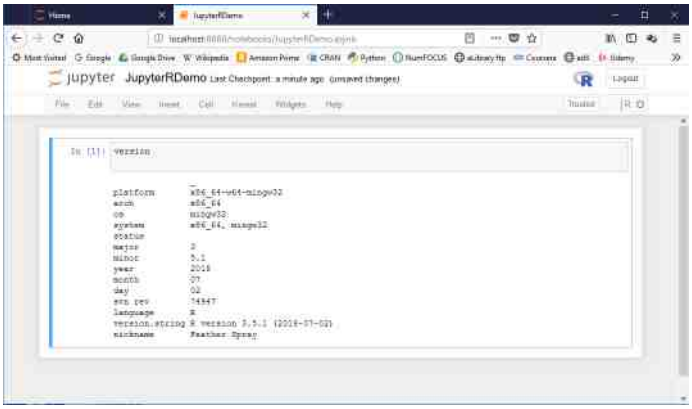


Figure 4.1: Jupyter R Demo

Figure 4.2: Jupyter Python Demo

4.3.3 Working in Jupyter and Writing in R, Python or Both in the Same Notebook

There are multiple kernel plugins available for jupyter notebooks. As A rule, the notebooks only use one language kernel at a time. An exception is the [SOS kernel](https://vatlab.github.io/sos-docs/.

Jupyter is running first level in python. It is afterall a python module. When running the R kernel (irkernel) remmber you are running R in python. See the caution below in the contest of rMarkdown code chunks.

Consider this Jupyter notebook using the R kernel,

and this Jupyter notebook using the 'Python kernel',

4.3.4 Working in rmarkdown and Using R and Python Code Chunks

Now consider an rmarkdown session running primarily in R. The T code chunks are executed in the same R session as the rMarkdown. On the otherhand, a python code chunk is running in a python guest environment in the R level host.

Caution: I have found behavior is unpredictible to call R objects back with Python chunks run in rmarkdown (ie. inside another instance of R). The identity of R_User for rpy2 gets lost. There are limits to how deeply you should nest languages within languages.

That said, the rmarkdown chunks look like this.

```
'''{r}
```

````
cat("PI in R is", pi)
```

```{python}
import math
print(math.pi)
```
````
````

Chapter 5

Basic Mathematics in R and Python

This is pretty straight forward and intuitive stuff. Just the same, for completeness here it is. The Standard libraries of python have the function beyond the four arithmetic operations.

5.1 Add, Subtract, Multiply and Divide

The usual quick calculation would be done in immediate or interactive mode without relying on print statements. There is outwardly no difference if the print is used as would be the case in a script. However Printing a set of values reqires the set be organized into an object in itself, a vector.

```
# Loading reticulate package to bring python interpreter on line.
library("reticulate")
```

5.1.1 R Scripting

```
# Some immediate calculations, ie. using R like a calculator
2 + 3           # (simple interactive) addition
```

```
## [1] 5
```

```
5 - 2           # subtraction
```

```
## [1] 3
```

```
6 * 2           # multiplication
```

```
## [1] 12
```

```
4 / 3           # division
```

```
## [1] 1.333333
```

```
print(4 / 3)    # calculation in a script requires print to display an answer
```

```
## [1] 1.333333
```

```
# Variable assignment and printing grouped variables as a vector
a <- (2 + 3)           # addition
b <- (5 - 2)           # subtraction
c <- (6 * 2)           # multiplication
```

```
d <- (4 / 3)            # division
print (c(a, b, c, d))   # printing all results together one statement
```

```
## [1]  5.000000  3.000000 12.000000  1.333333
```

Python, on the otherhand considers literal expressions and variables individually (though we'll see they can be groups as well.) This thr puthon print statement is by default printing one or more individual objects as scalars whereas R print function prints object and there is no scalar. The simplest object in R is the vector.

5.1.2 Python Scripting

The simplest object in python is the scalar, having one of five **basic datatypes**,

- Integers
- Floating-Point Numbers
- Complex Numbers
- Strings
- Boolean Type

```
# The same immediate calculations, ie. using iPython like a calculator
print(2 + 5, 3 * 8, 5 - 1, 67 / 3) # printing all immediate results together, one statement
```

```
## 7 24 4 22.333333333333332
```

```
# OR assignment and print
a = 2 + 5           # assigned addition
b = 5 - 1           # assigned subtraction
c = 3 * 8           # assigned multiplication
d = 67 / 3          # assigned division
print(a,b,c,d)      # printing all results together one statement
```

```
## 7 4 24 22.333333333333332
```

```
print('a =', a, 'b =',b, 'c =',c, 'd =',d) # annotating print is just using 4 literals and 4 variables
```

```
## a = 7 b = 4 c = 24 d = 22.333333333333332
```

This differs from R where a single object is printed but that object may be a group of objects combined.

5.2 Other basic algebraic operators in R and Python

Next lets compare modulus, powers and interger division in R and python,

```
35 %% 2     # modulo operator
```

```
## [1] 1
```

```
35 %/% 2    # integer division
```

```
## [1] 17
```

```
35^2        # powers in R\
```

```
## [1] 1225
```

```
print(35%2)       # modulo operator
```

```
## 1
```

```
print(35 // 2)    # integer division       * note difference from R
```

```
## 17
```

```
print(35**2)      # powers in python     * note difference from R
```

```
## 1225
```

Note here with python the print statements are required otherwise only the last immediate calculation would get output to text.

When we extend our algebraic calculations in R the usual functions of square root, absolute value and exponentiation we still haven't had to load any library packages. All these are part of the core R.

```
abs(-5)          # absolute value
```

```
## [1] 5
```

```
sqrt(16)         # square root function
```

```
## [1] 4
```

```
exp(0)           # exponent (e^0 etc)
```

```
## [1] 1
```

Not necessarily so for python. We still have the basic functions for algebraic tasks but from sqrt on ward, we now need to go to the standard math library for these. As an aside we could have got more powerful versions of these functions (and more) importing numpy or numba or scipy library packages. This will come up in more detail later.

```
print(abs(-5))
```

```
## 5
```

```
print(divmod(10,3))             # Returns quotient and remainder of integer division
```

```
## (3, 1)
```

```
print(max(2,10,3,14,4,28,7))    # Returns the largest of given arguments or items in an iterable
```

```
## 28
```

```
print(min(9,2,10,3,14,4,28,7))  # Returns the smallest of the given arguments or items in an) iterable
```

```
## 2
```

```
print(pow(3,4))                   # Raises a number to a power
```

```
## 81
```

```
print(round(3.1415962,3))         # Rounds a floating-point value
```

```
## 3.142
```

```
print(sum([2,3,4,5,6]))
```

```
## 20
```

```
import math                     # Import math package/module from Standard Library
print(math.sqrt(16))            # square root function
```

```
## 4.0
```

```
print(math.exp(0))              # exponentiation function (powers of e)
```

```
## 1.0
```

```
print(math.log10(5))            # base 10 logarithm
```

```
## 0.6989700043360189
```

```
print(math.log(5))              # natural logarithm
```

```
## 1.6094379124341003
```

I suggest, choosing your favorite R introduction and work through it's early examples using both R and Python - a good chance to try out Jupyter SOS kernel. It will also create a record of the basics of R and python side-by-side that you can refer back to.

A full list of the built-in (as opposed to Standard Library Funtions) is displayed, from the python documentation, below.

5.2.0.1 Python Built-in Functions

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	**import**()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

R has a very large set of built-in functions available before the need to load any libraries arises. The R Language Reference provides a comprehensive background on built in fucntions as well as the rest of the language.

So R is centered around the mandate of being a complete mathematical system, much like Matlab,(Octave or SciLab) which Python's manadate is that of a full featured scripting and programming environment.

However, calling standard library modules in python puts it quite easily and effectively on a even par with R. Going into more complex activites requires thar both load or import fucther libraries. Like **Python's Standard Library**, **the R Core Library** shipping with the R install is described in the Full R Reference Manaual and covers a wide range of mathematical, statistical and programming solutions.

We will go deeper into these and the other external Libraries and modules in the next chapter, covering functional programming in more detail. As well focused discussions on specific Data Science domains form a block of Chapters in Part II of the book.

Chapter 6

Functional Programming with R and Python

At this point we are ready to discuss the functional programming paradigm in R and Python. The other important paradigm, Object-Oriented, will be discussed in the following Chapter.

In using python in a functional paradigm, a script is built as a series of functions. Data is passed through one of more functions (ie. as series of functions) to carry out steps of a larger task set. When data is passed to a function to carry out some task or analysis on a dataset results may then printed out directly, stored in memory or on disk, or displayed in graphical form by passing the analysis to another set of functions such as those of matplotlib for plotting etc.

A function is a set of stored programmatic statements stored in an accessible form for reuse. The function is called in R or Python by simply inserting it by name in a statement to be executed by one of the interpreters.

6.1 Defining Functions

A function is defined in a specific format for each language.

6.1.1 R Scripting

```
doArea <- function(radius){
  area <- pi * radius^2
}
```

6.1.2 Python Scripting

```
import math
def calcArea(radius):
  area = math.pi * radius**2
  return area
```

6.2 Calling and Using Functions

6.2.1 R Scripting

```
radius = 4
print(doArea(radius))
```

```
## [1] 50.26548
```

```
x = doArea(4) + 100
print(x)
```

```
## [1] 150.2655
```

6.2.2 Python Scripting

```
import math
def calcArea(radius):
  area = math.pi * radius**2
  return area
print(calcArea(4))
```

```
## 50.26548245743669
```

Let's discuss the differences in how the functions were called to be used. COnsider first that the book is set-up as written in rMarkdown which is running in an instance of R. The memory of the code chunk and the underling R which rmarkdown is running are the same. In python terms, the namespace is the same. This is not the case with the python code chunks. We have to explicitly pass to (and retrieve from) the R namespace for it to be accessible to the python chunk, running in it's on (python) namespace. As a result we had to redefine the function in the same namespace in which we were calling it.

With R code chunks the R memory space is open throughout but the python chunks name space is closed each time the python chunk closes. This is a feature of how I am using the R environment to write this book, not of python per se.

Further, recalling we are writing scripts, the print statements are required for both to display the result. We could have used the cat() function of course to display the R output as well.

```
cat(x)       # x is still in R memory here
```

```
## 150.2655
```

Note that the result number [1] is not printed with R using cat() or the python print() functions.

There are more subtlies of memory management in calling one language from inside the other. These will come up below but for now knowing how things behave in the R/rMarkdown environment as above will cover our memory space issues.

6.3 Writing Function Examples in R and Python

6.3.0.1 Example 1 A Surveyor's Function

Lets write a function in R and Python to calculate the height of a tower knowing the angle of elevation at a given distance from the base. This would be something we might need when sighting through a surveyors transit.

Given a distance from the base of 1000 meters, how high is a tower whose top was measured at an angle of 25 degrees at that distance.

```
d = 1000     # distance in meters to base
ad = 25      # angle in degrees

# write a function to convert degrees to radians
deg2rad = function(angledeg){
  angledeg * (pi/180)
}

# write function to calcular height from angle in degrees
ht = function(d, ad){
  h = d*tan(deg2rad(ad))
}

# now call and output function with distance d and angle ad in degrees
cat("Tower Height in meters is: ", ht(d, ad), "\n")
```

```
## Tower Height in meters is:  466.3077
```

So we have written 2 functions, one to convert degrees to radians, that is the expected measure of the angle going into ttigonometric functions of R. Then we wrote a function to accept degrees input and give the height in meters (the input distance units) The function definitions use the keyword 'function, followed by zero or more arguments and then the code block of the function is set out by brace brackets'{ }'. The function is assigned to a name like a variable.

Lets do it with python now,

```
# set variables to problem givens
d = 1000     # distance in meters to base
ad = 25      # angle in degrees
# write a function to convert degrees to radians
def deg2rad(angledeg):
    import math
    return(angledeg * (math.pi/180))

# write function to calcular height from angle in degrees
def ht(d, ad):
  import math
  return (d*math.tan(deg2rad(ad)))
# now call and output function with distance d and angle ad in degrees
print("The height in meters of the tower is :", ht(d, ad))
```

```
## The height in meters of the tower is : 466.3076581549986
```

In the python declarations the definition starts with the keyword 'def' a name and zero or more arguments and there is a colon ':' at the end of the first line. The code block of the function is also indented rather than set out by brace brakets '{ }'. The first line is the signature of the function.

Both functions return a value, typically the last value in the calculation chain for R but as an explicitly called return for python. Note for the two python functions one has a space after return and the other doesn't. This is because the bracket do not contain return parmeters but group the expression to be returned.

Also as previously alluded to pi in R is a core CONSTANT but in python it is a CONSTANT in the math module which must be imported to use it. We could have just explicitly imported only pi at the start and referred to it directly by name as in R.

```
# Python code
 print(pi)
```

```
Gives an error:

 Traceback (most recent call last):
   File "C:\Users\medma\AppData\Local\Temp\RtmpYlYLtl\chunk-code-148c67392482.txt", line 1, in
   <module>
     print(pi)
 NameError: name 'pi' is not defined
```

```
from math import pi
print(pi)
```

```
## 3.141592653589793
```

Once again you must know where to go to get some constants and functions with python which are built-in in R. Check out the math module in the standard python library[3].

6.4 Using the Core R and Python Standard Libraries

Core R distributions currently install with a base set of function packages. The Table in the Appendix 3 lists these.

The python standard libraries contain alot of modules which can be imported. They are grouped in modules by functionality. The functions contained in the modules can be looked up in the Official Documentation online or referring to the Hellmann[2] book.

As done for the R base system in the Appendix, I also list the groups of python modules of the Standard Library in the Appendix 3 by kind as in the Python Documentation[3], but generally including only occasional examples of the many functions each contain. Again a more exhaustive handling is available as indicated.

It's a good time to dive into the functions of the core libraries and try some more sophisticated R and Python tasks. We'll first compare loading and saving datasets in both languages. R has file functions for saving dataframes as comma separated variables, csv, files. We will then load this data back into R and python to compare these tasks in each.

6.4.1 R Scripting

The dataframe airquality is a built in dataset. Rather than display the whole tales head(n) displays the first n lines (default n=6). SImilarly tail(n) would display the last n lines. The named dataframe 'table' is created from airquality data. Then we tell R to write a file names airquality.csv. R automatically closes the file after writing is complete.

```
# loading dataset from base R called
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
table = airquality
write.csv(table,file="airquality.csv")
```

6.4.2 Python Scripting

We need the io module for the file access function. The open() function opens the file "airquality.csv" for reading only ("r") and creates a filehandle, we called file. Now, file is a handle object so here is a first nod to OOP dotted function syntax to read a line from it. Something more functionally intuitive, like `readline(file)` will yield an error. Because calling of R function use dot syntax calls this is not new to us here. Next chapter moves fully to OOP paradigm.

Then we print out what we read - a single line from file (in fact, the first line)

```
# Needs io module functions
import io
#load csv data with Python
file = open("airquality.csv", "r")
aline = file.readline()
print(aline)
```

```
## "","Ozone","Solar.R","Wind","Temp","Month","Day"
```

```
aline = file.readline()
print(aline)
```

```
## "1",41,190,7.4,67,5,1
```

```
file.close
```

When we read a line again, we read the next line. Python readline maintains a pointer it moves as we continue to readlines. The Two lines we printer were the column names as we saved them and a line of comma separated values.

Lets read the airquality.csv file back into R assigning a name "newtable" to the dataframe the read.csv command creates to store the data loaded.

```
# Load csv data with R
newtab = read.csv(file="airquality.csv", header=TRUE)
head(newtab)
```

```
##   X Ozone Solar.R Wind Temp Month Day
## 1 1    41     190  7.4   67     5   1
## 2 2    36     118  8.0   72     5   2
## 3 3    12     149 12.6   74     5   3
## 4 4    18     313 11.5   62     5   4
## 5 5    NA      NA 14.3   56     5   5
## 6 6    28      NA 14.9   66     5   6
```

R automatically loads the csv data back into a dataframe. However, the python readline reads the data as a data stream (a stream of characters) which we told it to store in the aline variable. python does not in this case parse the stream int any special structure - that is left to us to do.

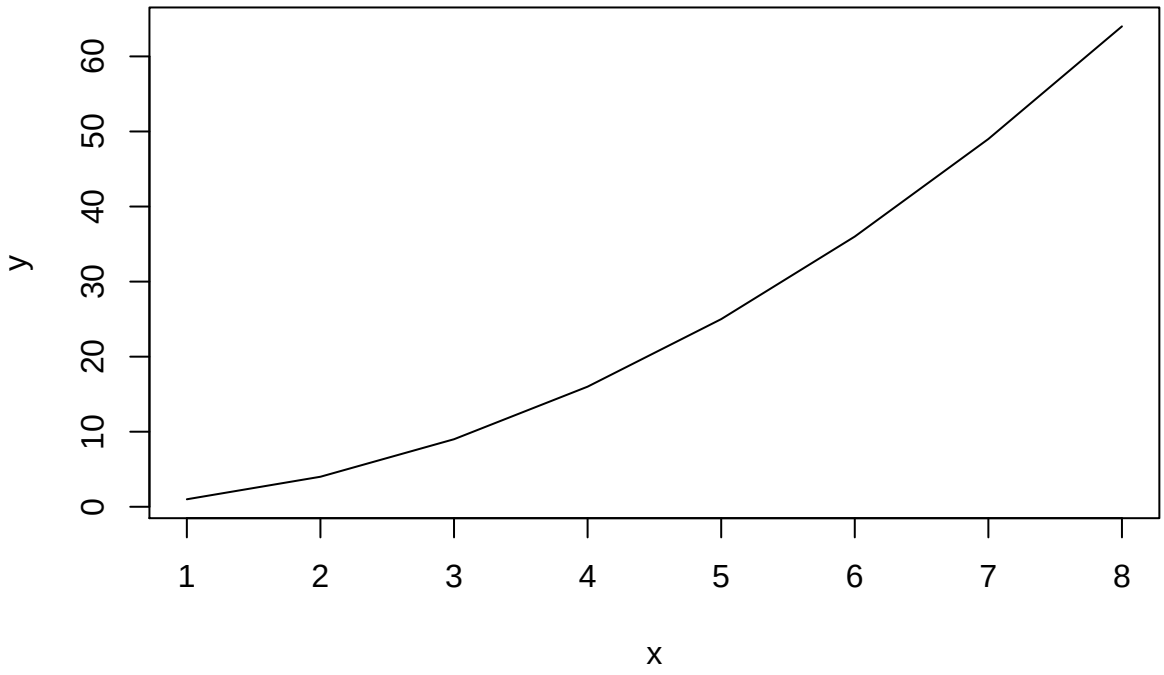
We see something else too. R wrote the row numbers to the file (because we didn't say otherwise) and assigned a new set again when we loaded the data again.

6.4.3 More Function Examples

Basic plotting functions are part of the r-base system we can easily plot a function,

```
# Create a function to make a line plot or squares of x between 1 and 8
linePlotMySquares <-function(type = "l", ...) {
  x = c(1,2,3,4,5,6,7,8)
  y = x**2
  plot(x, y, type = type, ...)
}

# Call our function
linePlotMySquares()
```



Python does not have a basic plotting function and the Standard Library has not provided one either. You casn write your own but there is a popular python library from the SciPy folks called Matplotlib which once installed from the shell with PIP,

```
> pip install matplotlib

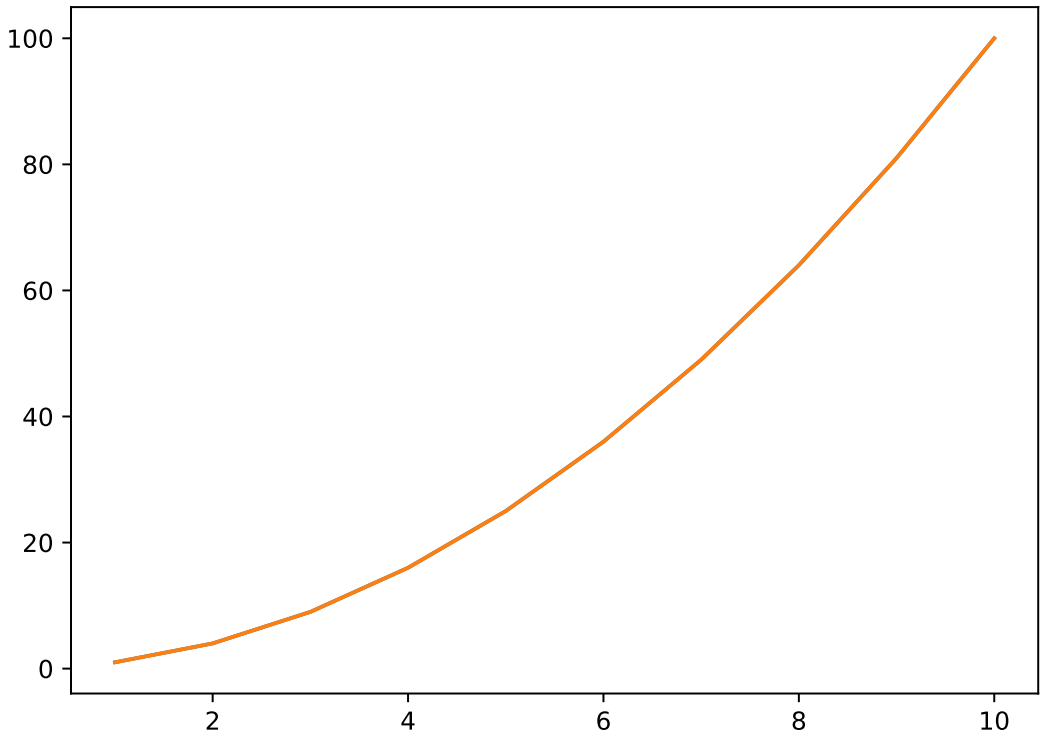
OR

> python -m pip install matplotlib
```

is available for import, and it contains a module called pyplot which has a simple plot function

```
from matplotlib import pyplot as plt  # import pyplot library module from matplotlib
x = list(range(1,11))                  # make a sequence of numbers from 1 through 8, called x
```

```
y = [0,]                                # create a list, must have at least one element to concat to
for val in x:                           # loop through list x to calculate squares
    y = y + [val**2,]                   # concatinate each square to list y as calculated
y = y[1:11]                              # select the values 1 to 8 from y, call them y
plt.plot(x,y)                           # plot the x vs y pairs
plt.show()                              # show the plot
```



There are a few things to discuss here. First is the structure of the matplotlib library. Python code for import are module files (which end in the .py extension like any python scripts or programs).

> A script or program is generally a directly executable piece of code, run by itself. A module is generally a library, imported by other pieces of code.
>
> Note that there's no internal distinction – both are executable and importable, although library code often won't do anything (or will just run its unit tests) when executed directly and importing code designed to be a script will cause it to execute, hence the common if **name** == "**main**" test.[17]

So what we are doing with `from matplotlib import pyplot as plt` is saying: "from the file matplotlib(.py) import the functions in the pyplot class and call the namespace containing them plt"" so that we can call them.

Note: pip is a module, but runs independently from the command line because it contains script with a `__main__` function accessed via pip3.exe to setup and start the process.

It also runs from command line as a module executed by python with the -m switch, by

```
python -m pip install <some package>
```

the -m switch tells python to run the module as a script. Most library modules are not executable as python scripts or if they are, run self-tests called by a main() function.

Chapter 7

Object-Oriented Programming with R and Python

At this point we are ready to expand our exploration with object-oriented programming (OOP) in R and Python. Objects are like (collections of one or more) functions with some added structure. This structure has been defined by the four principles which set apart OOP.

- Abstraction.
- Encapsulation.
- Inheritance.
- Polymorphism.

Additionally the functions an object contains, called it's methods are conceptualized as relating to the actions of the object. The methods are actions which are done by the object for itself - an add method enables the object to add itself. A square object would add something to itself in an internally defined fashion.

Consider a function/method to transparently operates on a variable as would be appropriate to the data-type. This is **polymorphism**, a consequence of the message-passing model. An 'add' method would do some sort of add operation in a fashion appropriate to the object and the input, and it would do it by passing a "do add"" message into the blackbox of the object and the details are inaccessible to the user. Access to the objects characteristics is also by such methods and in no other way. This is **encapsulation** or 'information hiding'.

You can only alter an object itself by extending it's pattern in a new object. All the things of the old object are passed on and exist in the new object by re-inplementation of old characteristics and new characteristics and behaviors are implemented on top of the parent object's pattern. This is **inheritance**.

As above the object has control over how it is seen in the environment external to the object. The picture the object releases is an **abstraction** of it's pattern definition and contents. The totality of object data structure emerges from the outputs of the object methods.

In a generic form (peudocode) a class pattern or definition consists of a class signature (or name) and a contained code block with class details. This is the fundamental abstraction which is an object. It could be represented like this,

```
# Class indentity
class() [ include inheritance]

    #Class details
    # the characteristics
    class constants
```

```
    class variables

    # instance creation
    constructor()

    # class actions
    methods()
```

7.1 Defining Classes in R and Python

A class is a specific object pattern, The automobile class is an object with wheels, some mode of propulsion and driver methods. A class instance is a particular realization of car with specifics values added to characteristics inherited from the class.

R, Functions or Classes

Due to how it evolved, R has at least 3 kinds or levels of objects, developed in a chronologic order. These all, arguably, then strain the limits of what is or is not Object-Oriented. The explanation offered[22] is the origin of R from S as an interactive environment and further development faces compromises to this. It can be used as a scripting language, even as if running programs but at the base of it all the interpreter runs the scripts in the the main over it's interactive nature.

I think in an important sense none of the OOP attempts fit the bill. Witness that there are 3 attempts to provide an R environment for Object-Oriented development and all have different shortcomings in the implementation. Add further what is actually a 4th object-oriented-type, but outside of core-R, with the R6 Package. Summarizing,

- S3 (predominantly genertic functions without formal class definition)
- S4 (Class definitions)
- RC (Reference Classes, or 'euphemistically' S5 or R5)
- R6 (External to Core-R R6 package)

The result of all this is that only the masochistically R obscessed or extreme programmer ventures here. R scripting afficionados even with lots of time proceed with caution.

I learned Object-Oriented Programming with Java and statistics with R. After a look inside I wasn't going there - all the more reason for adding Python to one's Data Science repertoir.

I am not going to explore R OOP beyond the discussion under calling R objects below, really *a syntatic variant* of functional programming. Those readers trully coming from R will understand.

Python Objects

In contrast to R python Object-Oriented Programming meets nearly everybody's understanding of OOP. Python Objects when instantiated look, feel and behave like an object for me. The python system of namespaces and variable visability blend consistently with the idea of encapsulation and data-hiding.

Python objects inherit from parent objects as expected and polymorphism operates behind the object envelope transparently and as one logicall expects for data-types or structures. Abstraction is a conceptualization of the data-structres created as objects.

Lets create a simple python class[8-p627] like,

```
# python code
class SimpleClass:
  # Define the class object

  # Note: Implied constructor is
```

```
  #    that of the base object of
  #    the python environment.

  # Define class methods
  def setdata(self, value):       # a value 'setter' method
    self.value = value            #   self is the instance itself
  def getdata(self):              # a value 'getter' method
    print(self.value)

# create object instance from class definition
x = SimpleClass()
x.setdata(5)                      # x became 'self' in the abstract
x.getdata()                       # what is the value of x
```

```
## 5
```

```
print(x.value)
```

```
## 5
```

Well, now we just accessed the value of x directly - not hidden data as we should expect.

Let's try another, the ubiquitous car example.

```
# what is a car?
class Car:
  _motor = ""
  _style = ""
  _year = ""

  def __init__(self, year, style, motor):      # constructor, always an __init__ function
    self._motor=motor
    self._style=style
    self._year=year

  def setmotor(self, motor):       # setters
    self._motor=motor

  def setstyle(self, style):
    self._style=style

  def getmotor(self):              # getters
    return _self.motor

  def getstyle(self):
    return _self.style

  def toString(self):
    return (self._year+","+self._motor+","+self._style)


# build a car
Newcar = Car('2019','Sedan', '4 cylinder gasoline' )
# What is 'our' Newcar?
print(Newcar)
```

```
## <__main__.Car object at 0x0000000064B65EB8>
```

OOoops I printed out the object info not the instance data!

What happens if I try direct access to an instance variable 'style'?

```
print(Newcar.style)
```

We get an error,

```
Error in py_run_string_impl(code, local, convert) :
  AttributeError: 'Car' object has no attribute 'style'

Detailed traceback:
  File "<string>", line 1, in <module>
Calls: <Anonymous> ... py_capture_output -> force -> py_run_string -> py_run_string_impl
Execution halted
```

Unfortunately, it is still accessible but you need to use the internal name '_style'

```
print(Newcar._style)
```

```
## Sedan
```

Well in terms of encapsulation in python OOP, close but no cigar. But as leandrotk notes in this blog

> Non-public variables are just a convention and should be treated as a non-public part of the API.

That means when we see the leading underscore we 'pretend' the variable is private. The only way to control it at all is to not document the underscore for the object's API.

Now try again using the object's own print method.

```
print(Newcar.toString())
```

```
## 2019,4 cylinder gasoline,Sedan
```

What about inheritance? Let's subclass the car as a truck.

```
class Truck(Car):
    _type = ""
    def __init__(self, year, style, motor, type):
        super().__init__(year, style, motor)
        self._type = type

    def toString(self):
        return super().toString() + "," + self._type

Bigtruck = Truck('2018','Truck','6 cylinder diesel', 'box' )
print(Bigtruck.toString())
```

```
## 2018,6 cylinder diesel,Truck,box
```

We have used the car pattern, referring to it as the superclass super. and we have overridden the tostring function to include the _type variable appended to the car string.

7.2 Using Objects and Classes

An instance of a class is created by the class' constructor. An object constructor is implicitly inherited by all classes, from the language's object definition and implementation. It is often explicitly overridden by a customized contructor.

7.2.1 R Scripting

7.2.2 Python Scripting

Part III

Data Science Topics in Python Compared to R

Chapter 8

Clean and Tidy Data

In an ideal world all data would come organized and in an immediatelyt usable form. Of course, such is unlikely. Datasets are missing values, in the wrong datatype for use, have misleading correlations, such as with time data, and so on. Both R and prython have library and builting functions to clean and tidy or data in a documented reproducible way before proceeding with our studies on it.

8.1 Reproducibility

8.2 R Data Munging

8.3 Python Data Munging

Chapter 9

Using Probability Distributions with R and Python

At this point we are ready to discuss the functional programming paradigm in R and Python. The other important paradigm, Object-Oriented, will be discussed in the following Chapter.

9.1 Basic Probability Issues

9.1.1 R Scripting

9.1.2 Python Scripting

9.2 Using the Distrbutions

9.2.1 R Scripting

9.2.2 Python Scripting

9.3 Other Libraries with Probability and Statistical Packages

9.3.1 R Scripting

9.3.2 Python Scripting

Chapter 10

Descriptive Statistics and Data Exploration

At this point we are ready to discuss the functional programming paradigm in R and Python. The other important paradigm, Object-Oriented, will be discussed in the following Chapter.

10.1 Defining Functions

10.1.1 R Scripting

10.1.2 Python Scripting

10.2 Calling and Using Functions

10.2.1 R Scripting

10.2.2 Python Scripting

10.3 The Core or Standard Libraries

10.3.1 R Scripting

10.3.2 Python Scripting

Chapter 11

Statistical Analysis and Modeling

At this point we are ready to discuss the functional programming paradigm in R and Python. The other important paradigm, Object-Oriented, will be discussed in the following Chapter.

11.1 Defining the Available Functions

11.1.1 R Scripting

11.1.2 Python Scripting

11.2 Calling and Using Functions

11.2.1 R Scripting

11.2.2 Python Scripting

Chapter 12

Non-Stochastic Models

Many mathematical models are deterministic. This includes mathematical programming.

Chapter 13

Reproducible Research

Chapter 14

References

1. Core R Team (Ed.), R Reference Index, R Foundation for Statistical Computing, Vienna, 2018.
2. Hellmann, Doug, The Python 3 Standard Library by example, Addison-Wesley, Boston MA, 2017.
3. Python Software Foundation (Ed.)., Python 3.7.1 documentation, Python Software Foundation, Wilmington Delaware, 2018
4. RStudio Consortium (Ed.), RStudio Documentation, Boston MA, 2018
5. Core R Team (Ed.), R Language Reference, R Foundation for Statistical Computing, Vienna, 2018.
6. Python Software Foundation(Ed.), Python Package Index, Wilmington Delaware, 2018
7. Dalgaard, Peter; Introductory Statistics with R, Springer-Valng, New York NY, 2002.
8. Lutz, Mark; Learning Python 4^{th} Ed., O'Reilly Media, Sebastopol CA, 2009.
9. Grus, Joel; Data Science from Scratch, First Principles with Python, O'Reilly Media, Sebastopol CA, 2015
10. Boschetti, Alberto; Luca Massaron; Python Data Science Essentials, Packt Publishing, Birmingham UK, 2015
11. Perktold, Josef; Skipper Seabold; Jonathan Taylor; StatsModels Washington DC Chicago IL, 2017
12. Skipper Seabold; Perktold, Josef; "Statsmodels: Econometric and Statistical Modeling with Python", PROC. OF THE 9th PYTHON IN SCIENCE CONF. (SCIPY 2010)
13. Veneble W.; B. Ripley; Modern Applied Statistics with S, 4^{th} ed, Springer, New York NY, 2002.
14. Härdle, Wolfgang K; Léopold Simar; Applied Multivariable Analysis,3^{rd} Ed., Springer, New York NY, 2012.
15. George, Janeve Understanding Programming Paradigms,Bangalore, India, 2013. (Accessed:12/07/2018 1:25PM.)
16. Wikipedia contributors, "Programming paradigm," in Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Programming_paradigm&oldid=871880549 (Accessed:12/07/2018 1:25PM.)
17. Aylett, Andrew; "What is the difference between a module and a script in Python?" https://stackoverflow.com/questions/2996110/what-is-the-difference-between-a-module-and-a-script-in-python, accessed 12/11/2018 2:58PM.
18. Wickham, Hadley Advanced R Chapman & Hall/CRC The R Series, Abingdon UK, 2014.
19. Black, Kelly; R Object-Oriented Programming, Packt Publishing, Birmingham UK, 2014.

20. Chambers, John M; Extending R, Chapman & Hall/CRC The R Series, Abington UK, 2016.

21. Chang, Winston R6 with R Infrastructure Group, accessed 12/12/2018 11:27AM.

22. Leisch, Friedrich; Creating R Packages: A Tutoria, Compstat 2008-Proceedings in Computational Statistics. Physica Verlag, Heidelberg, Germany, 2008. accessed 12/12/2018 3:58PM

Chapter 15

Bibliography

Page name: Programming paradigm
Author: Wikipedia contributors
Publisher: Wikipedia, The Free Encyclopedia.
Date of last revision: 4 December 2018 00:45 UTC
Date retrieved: 7 December 2018 18:30 UTC
Permanent link: https://en.wikipedia.org/w/index.php?title=Programming_paradigm&oldid=871880549
Primary contributors: Revision history statistics
Page Version ID: 871880549

Part IV

Appendices

Chapter 16

Appendix 1

16.1 Comparative Syntax for Programming Constructs of R and Python

As introduced in the text, the syntax of both R and python for the basic programming constructs are reviewed there for reader convenience.

	R	Python
Basic Data types and structures	scalar == vector[0]	true scalars
	vectors, (numerical, character, logical, factor)	(string, integer, float, complex, boolean)
	arrays, == vectors (m x n) matrices,	arrays (m x n) of scalars
	data frames,	dictionary,
	and lists	and lists
Operators - arithmetic	+ - * /	+ - * /
- exponent	^	**
- modulus remainder	%%	%
- modular division	%/%	//
- logical / boolean	&, |, !	and, or, not
- logical / comparison	==, <, <=, >, >=, !=	==, <, <=, >, >=, !=
Flow Control		
- conditionals, decisions	if-else, ifelse(), switch	if, else, elif
- Loops	while(), for()	while, for

16.2 Extended Structures

R	Python

Chapter 17

Appendix 2

17.1 Mixed R and Python Examples with rMarkdown and Jupyter-SOS Notebooks

17.1.1 Using rmarkdown as a Mixed Language Environment

17.1.2 Using Jupyter with SOS as a Mixed Language Environment

Chapter 18

Appendix 3, R and Python Packages

18.1 R Core Package List

The base packages are part of R and are grouped bt functionality and to some degree of chonology of incorporation into the base distribution. The Packages of base functions are,

Package	Description
base	Base R functions
compiler	Functions to provide an interface to a byte code compiler for R
datasets	Base R datasets
grDevices	R Graphics Devices and Support for Colours and Fonts. Support for base and grid graphics
graphics	R functions for base graphics
grid	The Grid Graphics Package. A rewrite of the graphics layout capabilities.
methods	Formal Methods and Classes. Formally defined methods and classes for R objects and other programming tools.
parallel	Support for Parallel Computation, including random-number generation.
splines	Regression Spline Functions and Classes.
stats	R statistical functions
stats4	Statistical Functions using S4 classes.
tcltk	Interface and language bindings to Tcl/Tk GUI elements.
tools	Tools for package development, administration and documentation.
utils	R utility functions

A list if contained functions in any package can be displayed in R by (eg. for parallel package),

```
library(help = "parallel")  # of substitute any package name for "parallel"
```

The other list installed with first R install are the recommended packages. These are kind of and appocrapha of functions providing useful capabilities from some books and papers about specialized uses of the R system. These are,

Package	Description
kernsmooth	Functions for **kernel smoothing** (and density estimation) corresponding to the book: Wand, M.P. and Jones, M.C. (1995) "Kernel Smoothing".
MASS	*Functions and datasets* to support Venables and Ripley, "Modern Applied Statistics with S" (4th edition, 2002).

Package	Description
Matrix	A rich hierarchy of **matrix classes**, including *triangular, symmetric, and diagonal matrices*, both *dense* and *sparse*, using 'LAPACK' and 'SuiteSparse' libraries.
boot	Functions and datasets for **bootstrapping** from the book "Bootstrap Methods and Their Application" by A. C. Davison and D. V. Hinkley (1997, CUP), originally written by Angelo Canty for S.
class	Various functions for **classification**, including *k-nearest neighbour*, *Learning Vector Quantization* and *Self-Organizing Maps*.
cluster	Methods for **Cluster analysis**. Much extended the original from Peter Rousseeuw, Anja Struyf and Mia Hubert, based on Kaufman and Rousseeuw (1990) "Finding Groups in Data".
codetools	Code analysis tools for R.
foreign	Reading and writing data stored by some versions of 'Epi Info', 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', and for reading and writing some 'dBase' files.
lattice	A powerful and elegant **high-level data visualization system** inspired by Trellis graphics, with an *emphasis on multivariate data*.
mgcv	Multiscale Graph Correlation (MGC) is a framework developed by Shen et al. (2017) <arXiv:1609.05148> that extends global correlation procedures to be multiscale;
nlme	*Linear and Nonlinear* **Mixed Effects Models**.Fit and compare Gaussian linear and nonlinear mixed-effects models.
nnet	Feed-Forward Neural Networks and Multinomial Log-Linear Models
rpart	*Recursive partitioning* for classification, regression and survival trees. An implementation of most of the functionality of the 1984 book by Breiman, Friedman, Olshen and Stone.
spatial	Functions for kriging and point pattern analysis.
survival	Contains the core **survival analysis** routines.

There are literally thousands of CRAN repository packages accumulated over the years plus many more managed outside of the CRAN repository. Inclusion in CRAN requires a minimum standard process of package structure and documentation.

18.2 Python Standard Library[2] List

see also: https://docs.python.org/3.7/library/index.html

18.2.1 Built-in Functions[2]

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()

chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	**import**()
complex()	hasattr()	max()	round()	

18.2.2 Built-in Constants

includes Constants added by the **site** module

False	True	None	NotImplemented	Ellipsis or "..."
`__debug__`	quit(code=None)	exit(code=None)	copyright	
credits	license			

18.2.3 Built-in Types

- Truth Value Testing
- Boolean Operations — `and`, `or`, `not`
- Comparisons `== <= >= < > !=`
- Numeric Types — `int`, `float`, `complex`
- Iterator Types
- Sequence Types — 'list', 'tuple', 'range'
- Text Sequence Type — 'str'
- Binary Sequence Types — 'bytes', `bytearray`, `memoryview`
- Set Types — `set`, `frozenset`
- Mapping Types — `dict`
- Context Manager `Types`
- Other Built-in Types
- Special Attributes

18.2.4 Built-in Exceptions

- Base classes
- Concrete exceptions
- Warnings
- Exception hierarchy

18.2.5 Text Processing Services

- `string` — Common string operations
- `re` — Regular expression operations
- `difflib` — Helpers for computing deltas
- `textwrap` — Text wrapping and filling
- `unicodedata` — Unicode Database
- `stringprep` — Internet String Preparation
- `rlcompleter` — Completion function for GNU readline

18.2.6 Binary Data Services

- `struct` — Interpret bytes as packed binary data

  - Functions and Exceptions
  - Format Strings
  - Classes
- codecs — Codec registry and base classes
  - Codec Base Classes
  - Encodings and Unicode
  - Standard Encodings
  - Python Specific Encodings
    - Text Encodings
    - Binary Transforms
    - Text Transforms
  - `encodings.idna` — Internationalized Domain Names in Applications
  - `encodings.mbcs` — Windows ANSI codepage
  - `encodings.utf_8_sig` — UTF-8 codec with BOM signature

18.2.7 Data Types

- `datetime` — Basic date and time types
  - Available Types
    - `timedelta` Objects
    - `date` Objects
    - `datetime` Objects
    - `time` Objects
    - `tzinfo` Objects
    - `timezone` Objects
    - `strftime()` and `strptime()` Behavior
- `calendar` — General calendar-related functions
- `collections` — Container datatypes
  - `ChainMap` objects
  - `Counter` objects
  - `deque` objects
  - `defaultdict` objects
  - `namedtuple()` Factory Function for Tuples with Named Fields
  - `OrderedDict` objects
  - `UserDict` objects
  - `UserList` objects
  - `UserString` objects
- `collections.abc` — Abstract Base Classes for Containers
- `heapq` — Heap queue algorithm
- `bisect` — Array bisection algorithm
- `array` — Efficient arrays of numeric values
- `weakref` — Weak references
- `types` — Dynamic type creation and names for built-in types
- `copy` — Shallow and deep copy operations
- `pprint` — Data pretty printer
- `reprlib` — Alternate repr() implementation
- `enum` — Support for enumerations

18.2.8 Numerical and Mathematical Modules

- `numbers` — Numeric abstract base classes
  - The numeric tower

- `math` — Mathematical functions
  - Number-theoretic and representation functions
  - Power and logarithmic functions
  - Trigonometric functions
  - Angular conversion
  - Hyperbolic functions
  - Special functions
  - Constants
- `cmath` — Mathematical functions for complex numbers
  - Conversions to and from polar coordinates
  - Power and logarithmic functions
  - Trigonometric functions
  - Hyperbolic functions
  - Classification functions
  - Constants
- `decimal` — Decimal fixed point and floating point arithmetic
- `fractions` — Rational numbers
- `random` — Generate pseudo-random numbers
  - Bookkeeping functions
  - Functions for integers
  - Functions for sequences
  - Real-valued distributions
  - Alternative Generator
- `statistics` — Mathematical statistics functions
  - Averages and measures of central location
  - Measures of spread

18.2.9 Functional Programming Modules

- `itertools` — Functions creating iterators for efficient looping
- `functools` — Higher-order functions and operations, functions that act on or return other functions.
- `operator` — Standard operators as functions, efficient functions corresponding to the intrinsic operators of Python.

18.2.10 File and Directory Access

- `pathlib` — Object-oriented filesystem paths
- `os.path` — Common pathname manipulations
- `fileinput` — Iterate over lines from multiple input streams
- `stat` — Interpreting stat() results
- `filecmp` — File and Directory Comparisons
- `tempfile` — Generate temporary files and directories
- `glob` — Unix style pathname pattern expansion
- `fnmatch` — Unix filename pattern matching
- `linecache` — Random access to text lines
- `shutil` — High-level file operations
- `macpath` — Mac OS 9 path manipulation functions

18.2.11 Data Persistence

- `pickle` — Python object serialization

- `copyreg` — Register pickle support functions
- `shelve` — Python object persistence
- `marshal` — Internal Python object serialization
- `dbm` — Interfaces to Unix "databases"
- `sqlite3` — DB-API 2.0 interface for SQLite databases

18.2.12 Data Compression and Archiving

- `zlib` — Compression compatible with gzip
- `gzip` — Support for gzip files
- `bz2` — Support for bzip2 compression
- `lzma` — Compression using the LZMA algorithm
- `zipfile` — Work with ZIP archives
- `tarfile` — Read and write tar archive files

18.2.13 File Formats

- `csv` — CSV File Reading and Writing
- `configparse` — Configuration file parser
- `netrc` — netrc file processing
- `xdrlib` — Encode and decode XDR data
- `plistlib` — Generate and parse Mac OS X .plist files

18.2.14 Cryptographic Services

- `hashlib` — Secure hashes and message digests
- `hmac` — Keyed-Hashing for Message Authentication
- `secrets` — Generate secure random numbers for managing secrets

18.2.15 Generic Operating System Services

- `os` — Miscellaneous operating system interfaces
- `io` — Core tools for working with streams
- `time` — Time access and conversions
- `argparse` — Parser for command-line options, arguments and sub-commands
- `getopt` — C-style parser for command line options
- `logging` — Logging facility for Python
- `logging.config` — Logging configuration
- `logging.handlers` — Logging handlers
- `getpass` — Portable password input
- `curses` — Terminal handling for character-cell displays
- `curses.textpad` — Text input widget for curses programs
- `curses.ascii` — Utilities for ASCII characters
- `curses.panel` — A panel stack extension for curses
- `platform` — Access to underlying platform's identifying data
- `errno` — Standard errno system symbols
- `ctypes` — A foreign function library for Python

18.2.16 Concurrent Execution

- `threading` — Thread-based parallelism
- `multiprocessing` — Process-based parallelism
- `concurrent`
  - `concurrent.futures` — Launching parallel tasks
- `subprocess` — Subprocess management
- `sched` — Event scheduler
- `queue` — A synchronized queue class
- `_thread` — Low-level threading API
- `_dummy_thread` — Drop-in replacement for the _thread module
- `dummy_threading` — Drop-in replacement for the threading module

18.2.17 Context Variables

- `contextvars` - module provides APIs to manage, store, and access context-local state (context variables)
- `asyncio` - support for asynchronous io

18.2.18 Networking and Interprocess Communication

- `asyncio` — Asynchronous I/O (see associated context services)
- `socket` — Low-level networking interface
- `ssl` — TLS/SSL wrapper for socket objects
- `select` — Waiting for I/O completion
- `selectors` — High-level I/O multiplexing
- `asyncore` — Asynchronous socket handler
- ‘asynchat — Asynchronous socket command/response handler
- `signal` — Set handlers for asynchronous events
- `mmap` — Memory-mapped file support

18.2.19 Internet Data Handling

- `email` — An email and MIME handling package
- `json` — JSON encoder and decoder
- `mailcap` — Mailcap file handling
- `mailbox` — Manipulate mailboxes in various formats
- `mimetypes` — Map filenames to MIME types
- `base64` — Base16, Base32, Base64, Base85 Data Encodings
- `binhex` — Encode and decode binhex4 files
- `binascii` — Convert between binary and ASCII
- `quopri` — Encode and decode MIME quoted-printable data
- `uu` — Encode and decode uuencode files

18.2.20 Structured Markup Processing Tools

- `html` — HyperText Markup Language support
  - `html.parser` — Simple HTML and XHTML parser
  - `html.entities` — Definitions of HTML general entities
- `xml` XML Processing Modules

  - xml.etree.ElementTree — The ElementTree XML API
  - xml.dom — The Document Object Model API
    * xml.dom.minidom — Minimal DOM implementation
  - xml.dom.pulldom — Support for building partial DOM trees
  - xml.sax — Support for SAX2 parsers
    * xml.sax.handler — Base classes for SAX handlers
    * xml.sax.saxutils — SAX Utilities
    * xml.sax.xmlreader — Interface for XML parsers
    * xml.parsers.expat — Fast XML parsing using Expat

18.2.21 Internet Protocols and Support

- 'webbrowser — Convenient Web-browser controller
- cgi — Common Gateway Interface support
- cgitb — Traceback manager for CGI scripts
- wsgiref — WSGI Utilities and Reference Implementation
- urllib — URL handling modules
  - urllib.request — Extensible library for opening URLs
  - urllib.response — Response classes used by urllib
  - urllib.parse — Parse URLs into components
  - urllib.error — Exception classes raised by urllib.request
  - urllib.robotparser — Parser for robots.txt
- http — HTTP modules
  - http.client — HTTP protocol client
- ftplib — FTP protocol client
- poplib — POP3 protocol client
- imaplib — IMAP4 protocol client
- nntplib — NNTP protocol client
- smtplib — SMTP protocol client
- smtpd — SMTP Server
- telnetlib — Telnet client
- uuid — UUID objects according to RFC 4122
- socketserver — A framework for network servers
- http.server — HTTP servers
- http.cookie — HTTP state management
- http.cookiejar — Cookie handling for HTTP clients
- xmlrpc — XMLRPC server and client modules
  - xmlrpc.client — XML-RPC client access
  - xmlrpc.server — Basic XML-RPC servers
- ipaddress — IPv4/IPv6 manipulation library

18.2.22 Multimedia Services

- audioop — Manipulate raw audio data
- aifc — Read and write AIFF and AIFC files
- sunau — Read and write Sun AU files
- wave — Read and write WAV files
- chunk — Read IFF chunked data
- colorsys — Conversions between color systems
- imghdr — Determine the type of an image
- sndhdr — Determine type of sound file
- ossaudiodev — Access to OSS-compatible audio devices

18.2.23 Internationalization

- `gettext` — Multilingual internationalization services
- `locale` — Internationalization services

18.2.24 Program Frameworks

- `turtle` — Turtle graphics
- `cmd` — Support for line-oriented command interpreters
- `shlex` — Simple lexical analysis

18.2.25 Graphical User Interfaces with Tk

- `tkinter` — Python interface to Tcl/Tk
- `tkinter.ttk` — Tk themed widgets
- `tkinter.tix` — Extension widgets for Tk
- `tkinter.scrolledtext` — Scrolled Text Widget
- `IDLE` - Python's Integrated Development and Learning Environment
- Other Graphical User Interface Packages
  - PyGObject - Provides introspection bindings for C libraries, incl. GTK+ 3 widget set.
  - PyGTK - PyGTK provides bindings for an older version of the library, GTK+ 2.
  - PyQt - A sip-wrapped binding to the Qt toolkit.
  - PySide - A newer binding to the Qt toolkit, provided by Nokia..
  - `wxPython` - A cross-platform GUI toolkit for Python that is built around wxWidgets C++ toolkit

18.2.26 Development Tools

- `typing` — Support for type hints
- `pydoc` — Documentation generator and online help system
- `doctest` — Test interactive Python examples
- `unittest` — Unit testing framework
  - `unittest.mock` — mock object library
- `2to3` - Automated Python 2 to 3 code translation
- `test` — Regression tests package for Python
  - `test.support` — Utilities for the Python test suite
    - `test.support.script_helper` — Utilities for the Python execution tests

18.2.27 Debugging and Profiling

- `bdb` — Debugger framework
- `faulthandler` — Dump the Python traceback
- `pdb` — The Python Debugger
- The Python Profilers
- `timeit` — Measure execution time of small code snippets
- `trace` — Trace or track Python statement execution
- `tracemalloc` — Trace memory allocations

18.2.28 Software Packaging and Distribution

- `distutils` — Building and installing Python modules

- `ensurepip` — Bootstrapping the pip installer
- `venv` — Creation of virtual environments
- `zipapp` — Manage executable Python zip archives

18.2.29 Python Runtime Services

- `sys` — System-specific parameters and functions
- `sysconfig` — Provide access to Python's configuration information
- `builtins` — Built-in objects
- `__main__` — Top-level script environment
- `warnings` — Warning control
- `dataclasses` — Data Classes
- `contextlib` — Utilities for with-statement contexts
- `abc` — Abstract Base Classes
- `atexit` — Exit handlers
- `traceback` — Print or retrieve a stack traceback
- `__future__` — Future statement definitions
- `gc` — Garbage Collector interface
- `inspect` — Inspect live objects
- `site` — Site-specific configuration hook

18.2.30 Custom Python Interpreters

- `code` — Interpreter base classes
- `codeop` — Compile Python code

18.2.31 Importing Modules

- `zipimport` — Import modules from Zip archives
- `pkgutil` — Package extension utility
- `modulefinder` — Find modules used by a script
- `runpy` — Locating and executing Python modules
- `importlib` — The implementation of import

18.2.32 Python Language Services

- `parser` — Access Python parse trees
- `ast` — Abstract Syntax Trees
- `symtable` — Access to the compiler's symbol tables
- `symbol` — Constants used with Python parse trees
- `token` — Constants used with Python parse trees
- `keyword` — Testing for Python keywords
- `tokenize` — Tokenizer for Python source
- `tabnanny` — Detection of ambiguous indentation
- `pyclbr` — Python class browser support
- `py_compile` — Compile Python source files
- `compileall` — Byte-compile Python libraries
- `dis` — Disassembler for Python bytecode
- `pickletools` — Tools for pickle developers

18.2.33 Miscellaneous Services

- `formatter` — Generic output formatting

18.2.34 MS Windows Specific Services

See also the pywin32 extensions. These are available to install via PIP, and provide access to many of the Windows APIs from Python.

- `msilib` — Read and write Microsoft Installer files
- `msvcrt` — Useful routines from the MS VC++ runtime
- `winreg` — Windows registry access
- `winsound` — Sound-playing interface for Windows

See also "Undocumented services"" below.

18.2.35 Unix Specific Services

Module provides interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it.

- `posix` — The most common POSIX system calls
- `pwd` — The password database
- `spwd` — The shadow password database
- `grp` — The group database
- `crypt` — Function to check Unix passwords
- `termios` — POSIX style tty control
- `tty` — Terminal control functions
- `pty` — Pseudo-terminal utilities
- `cntl` — The fcntl and ioctl system calls
- `pipes` — Interface to shell pipelines
- `resource` — Resource usage information
- `nis` — Interface to Sun's NIS (Yellow Pages)
- `syslog` — Unix syslog library routines

See also "Undocumented services"" below.

18.2.36 Undocumented Modules

- `ntpath` — Implementation of os.path on Win32 and Win64 platforms.
- `posixpath` — Implementation of os.path on POSIX.

18.3 Other Package Sources (Third Party)

18.3.1 CRAN R Package Repository

The packages of the CRAN Package repository meet a minimum requirement for package documentation and usability in order to be included the Available CRAN Packages list. There is also a compilation of the packages by topic in the CRAN Task Views.

An external repository developed and managed by bioinformatics genomic oriented users in Bioconnector.

Many high quality packages are available from github repositories, particularly thos maintained by RStudio

18.3.2 PyPI - Python Package Index

Chapter 19

Appendix 4 Selected Tutorials and Learning Resources

This book is available for download in pdf format.

1 The **R Project for Statistical Computing**, https://www.r-project.org/ Probably should start here, or of course, here,

2. **Python Software Foundation**, https://www.python.org/, Official Documentation, Tutorials and more.
3. **Stack overflow**, https://stackoverflow.com/questions, is an ever accessible help desk wiki. It addresses quests in a multitude of domains including **R**, **Python**, many areas of mathematics including **Statistics**, and much much more. If you can't find you answer there (which is very unlikely) you can put your question out there to be answered. !! A Highly Recommended Go To !!
4. Just **Google** your topic, (or use your favorite search engine). This will yield, blogs, articles and tutorials and online books etc.
5. **W3schools** Developers Site, https://www.w3schools.com/python/default.asp these are quick and clear and concise. They will get you up and running quickly, particularly with many Web oriented tools. *Tutorials, References and Examples*. Unfortunately R is not yet represented here.
6. **SciPy** https://scipy.org/ developed and supports much of what can be considered the center court of python data science. **Scipy Library** itself, **Numpy**, **Matplotlib**, **Pandas**, **IPython** and more. Tutorials for each accessible from here.
7. **Wikipedia** https://en.wikipedia.org/wiki/Main_Page Wikipaedia has it's proponents and opponents but with your eyes wide open you can find anything you need to learn often in depth and in brief. It is not peer reviewed by domain professionals and this necessitates using you common sense and being an informed user. BUT, most often those domain professionals are there orbiting especially in computer and mathematics fields. I use it alot of alot of things. But I also check the sources provided.
8. Advanced R by Hadley Wickham, https://adv-r.hadley.nz/index.html The best book for taking R beyond scripting.