

# IQSS Workshops

September 2019



# Contents

<b>Introduction</b>	<b>7</b>
Table of Contents . . . . .	7
Authors and Sources . . . . .	7
 <b>I R</b>	 <b>9</b>
<b>1 Welcome</b>	<b>11</b>
1.1 Materials and setup . . . . .	11
1.2 Workshop goals and approach . . . . .	11
 <b>2 Graphical User Interfaces (GUIs)</b>	 <b>13</b>
2.1 Launch RStudio (skip if not using Rstudio) . . . . .	13
2.2 Exercise 0 . . . . .	13
2.3 Exercise 0 solution . . . . .	14
 <b>3 R basics</b>	 <b>15</b>
3.1 Function calls . . . . .	15
3.2 Assignment . . . . .	15
3.3 Asking R for help . . . . .	15
 <b>4 Getting data into R</b>	 <b>17</b>
4.1 Installing and using R packages . . . . .	17
4.2 Readers for common file types . . . . .	17
4.3 Baby names data . . . . .	18
4.4 Exercise 1: Reading the baby names data . . . . .	18
4.5 Exercise 1 solution . . . . .	18

<b>5</b>	<b>Popularity of your name</b>	<b>19</b>
5.1	Filtering and arranging data . . . . .	19
5.2	Other logical operators . . . . .	20
5.3	Exercise 2: Peak popularity of your name . . . . .	20
5.4	Exercise 2 solution . . . . .	21
<b>6</b>	<b>Plotting baby name trends over time</b>	<b>23</b>
6.1	Exercise 3: Plotting peak popularity of your name . . . . .	23
6.2	Exercise 3 solution . . . . .	24
<b>7</b>	<b>Finding the most popular names</b>	<b>25</b>
7.1	Computing better measures of popularity . . . . .	25
7.2	Operating by group . . . . .	25
7.3	Exercise 4: Most popular names . . . . .	25
7.4	Exercise 4 solution . . . . .	26
<b>8</b>	<b>Percent choosing one of the top 10 names</b>	<b>29</b>
8.1	Exercise 4: Popularity of the most popular names . . . . .	29
8.2	Exercise 4 solution . . . . .	30
<b>9</b>	<b>Saving our Work</b>	<b>31</b>
9.1	Saving individual datasets . . . . .	31
9.2	Saving and loading R workspaces . . . . .	31
<b>10</b>	<b>Wrap-up</b>	<b>33</b>
10.1	Help us make this workshop better! . . . . .	33
10.2	Additional resources . . . . .	33
<b>II</b>	<b>Python</b>	<b>35</b>
<b>11</b>	<b>Introduction to Python</b>	<b>37</b>
11.1	Setup instructions . . . . .	37
11.2	Workshop goals and approach . . . . .	37
11.3	What is Python? . . . . .	38
11.4	How can I interact with Python? . . . . .	38

11.5 Reading the text of Alice in Wonderland from a file . . . . .	40
11.6 Counting chapters, lines, and words . . . . .	41
11.7 Exercise: Reading text from a file and splitting . . . . .	42
11.8 Exercise: count the number of main characters . . . . .	45
11.9 Working with nested structures: words within paragraphs within chapters . .	45
11.10Exercise: Iterating and counting things . . . . .	47
11.11Importing numpy and calculating simple statistics . . . . .	48
11.12Where to go from here . . . . .	48



# Introduction

## Table of Contents

Materials for the Data Science Services statistical software workshops from the Institute for Quantitative Social Science at Harvard.

1. Introduction to R
2. Regression models in R
3. Graphics in R using ggplot2
4. R data wrangling
5. Introduction to Python
6. Python web-scraping
7. Introduction to Stata

These workshops are a work-in-progress, please provide feedback! Email: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

## Authors and Sources

These content of these workshops are almost entirely the work of Ista Zahn, now at Harvard's School of Public Health. The current workshop materials have been modified by Steve Worthington, Jinjie Liu, and Yihan Wang, at Harvard's Institute for Quantitative Social Science.





# Part I

## R



# Chapter 1

## Welcome

**Topics** Assignment; Function arguments; Finding help; Reading data; Filtering and arranging data; Conditional operations; Saving data.

### 1.1 Materials and setup

**NOTE: skip this section if you are not running R locally** (e.g., if you are running R in your browser using a remote Jupyter server)

You should have R installed –if not:

- Download and install R from <http://cran.r-project.org>
- Download and install RStudio from <https://www.rstudio.com/products/rstudio/download/#download>

Notes and examples for this workshop are available at

Start RStudio create a new project: - On Windows click the start button and search for rstudio. On Mac RStudio will be in your applications folder. - In Rstudio go to **File -> New Project**. - Choose **New Directory** and **New Project**. - Choose a name and location for your new project directory.

### 1.2 Workshop goals and approach

In this workshop you will

- learn R basics,
- learn about the R package ecosystem,
- practice reading files and manipulating data in R

A more general goal is to get you comfortable with R so that it seems less scary and mystifying than it perhaps does now. Note that this is by no means a complete or thorough introduction to R! It's just enough to get you started.

This workshop is relatively informal, example-oriented, and hands-on. We won't spend much time examining language features in detail. Instead we will work through an example, and learn some things about the R along the way.

As an example project we will analyze the popularity of baby names in the US from 1960 through 2017. Among the questions we will use R to answer are:

- In which year did your name achieve peak popularity?
- How many children were born each year?
- What are the most popular names overall? For girls? For Boys?

## Chapter 2

# Graphical User Interfaces (GUIs)

There are many different ways you can interact with R. See the Data Science Tools workshop notes for details.

For this workshop I encourage you to use RStudio; it is a good R-specific IDE that mostly just works.

### 2.1 Launch RStudio (skip if not using Rstudio)

**Note:** skip this section if you are not using Rstudio (e.g., if you are running these examples in a Jupyter notebook).

- Start the RStudio program
- In RStudio, go to **File => New File => R Script**

The window in the upper-left is your R script. This is where you will write instructions for R to carry out.

The window in the lower-left is the R console. This is where results will be displayed.

### 2.2 Exercise 0

The purpose of this exercise is to give you an opportunity to explore the interface provided by RStudio (or whichever GUI you've decided to use). You may not know how to do these things; that's fine! This is an opportunity to figure it out.

Also keep in mind that we are living in a golden age of tab completion. If you don't know the name of an R function, try guessing the first two or three letters and pressing TAB. If you guessed correctly the function you are looking for should appear in a pop up!

1. Try to get R to add 2 plus 2.

```
##
```

2. Try to calculate the square root of 10.

```
##
```

3. R includes extensive documentation, including a manual named “An introduction to R”. Use the RStudio help pane. to locate this manual.

## 2.3 Exercise 0 solution

```
## 1. 2 plus 2  
2 + 2  
## or  
sum(2, 2)
```

```
## 2. square root of 10:  
sqrt(10)  
## or  
10^(1/2)
```

```
## 3. Find "An Introduction to R".
```

```
## Go to the main help page by running 'help.start()' or using the GUI  
## menu, find and click on the link to "An Introduction to R".
```

## Chapter 3

# R basics

### 3.1 Function calls

The general form for calling R functions is

```
## FunctionName(arg.1 = value.1, arg.2 = value.2, ..., arg.n = value.n)
```

Arguments can be matched by name; unnamed arguments will be matched by position.

### 3.2 Assignment

Values can be assigned names and used in subsequent operations

- The “gets” `<-` operator (less than followed by a dash) is used to save values
- The name on the left gets the value on the right.

```
sqrt(10) ## calculate square root of 10; result is not stored anywhere  
x <- sqrt(10) # assign result to a variable named x
```

Names should start with a letter, and contain only letters, numbers, underscores, and periods.

### 3.3 Asking R for help

You can ask R for help using the `help` function, or the `?` shortcut.

```
help(help)
```

The `help` function can be used to look up the documentation for a function, or to look up the documentation to a package. We can learn how to use the `stats` package by reading its documentation like this:

```
help(package = "stats")
```



## Chapter 4

# Getting data into R

R has data reading functionality built-in – see e.g., `help(read.table)`. However, faster and more robust tools are available, and so to make things easier on ourselves we will use a *contributed package* called `readr` instead. This requires that we learn a little bit about packages in R.

### 4.1 Installing and using R packages

A large number of contributed packages are available. If you are looking for a package for a specific task, <https://cran.r-project.org/web/views/> and <https://r-pkg.org> are good places to start.

You can install a package in R using the `install.packages()` function. Once a package is installed you may use the `library` function to attach it so that it can be used.

```
## install.packages("readr")
library(readr)
```

### 4.2 Readers for common file types

In order to read data from a file, you have to know what kind of file it is. The table below lists functions that can import data from common plain-text formats.

Data Type	Function
comma separated	<code>read_csv()</code>
tab separated	<code>read_delim()</code>
other delimited formats	<code>read_table()</code>
fixed width	<code>read_fwf()</code>

**Note** You may be confused by the existence of similar functions, e.g., `read.csv` and `read.delim`. These are legacy functions that tend to be slower and less robust than the `readr` functions. One way to tell them apart is that the faster more robust versions use underscores in their names (e.g., `read_csv`) while the older functions use dots (e.g., `read.csv`). My advice is to use the more robust newer versions, i.e., the ones with underscores.

### 4.3 Baby names data

The examples in this workshop use US baby names data retrieved from <https://catalog.data.gov/dataset/baby-names-from-social-security-card-applications-national-level-data>. A cleaned and merged version of these data is available at <http://tutorials.iq.harvard.edu/data/babyNames.csv>.

### 4.4 Exercise 1: Reading the baby names data

Make sure you have installed the `readr` package and attached it with `library(readr)`.

Baby names data are available at "<http://tutorials.iq.harvard.edu/data/babyNames.csv>".

1. Open the `read_csv` help page to determine how to use it to read in data.
2. Read the baby names data using the `read_csv` function and assign the result with the name `baby.names`.
3. BONUS (optional): Save the `baby.names` data as a Stata data set `babynames.dta` and as an R data set `babynames.rds`.

### 4.5 Exercise 1 solution

```
## read ?read_csv
```

```
baby.names <- read_csv("http://tutorials.iq.harvard.edu/data/babyNames.csv")
```

## Chapter 5

# Popularity of your name

In this section we will pull out specific names and examine changes in their popularity over time.

The `baby.names` object we created in the last exercise is a `data.frame`. There are many other data structures in R, but for now we'll focus on working with `data.frames`.

R has decent data manipulation tools built-in – see e.g., `help(Extract)`. However, these tools are powerful and complex and often overwhelm beginners. To make things easier on ourselves we will use a *contributed package* called `dplyr` instead.

```
## install.packages("dplyr")  
library(dplyr)
```

### 5.1 Filtering and arranging data

One way to find the year in which your name was the most popular is to filter out just the rows corresponding to your name, and then arrange (sort) by Count.

To demonstrate these techniques we'll try to determine whether “Alex” or “Jim” was more popular in 1992. We start by filtering the data so that we keep only rows where Year is equal to 1992 and Name is either “Alex” or “Mark”.

```
am <- filter(baby.names,  
             Year == 1992 & (Name == "Alex" | Name == "Mark"))  
am
```

Notice that we can combine conditions using `&` (AND) and `|` (OR).

In this case it's pretty easy to see that “Mark” is more popular, but to make it even easier we can arrange the data so that the most popular name is listed first.

```
arrange(am, Count)
```

```
arrange(am, desc(Count))
```

## 5.2 Other logical operators

In the previous example we used `==` to filter rows. Other relational and logical operators are listed below.

Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&gt;</code>	greater than
<code>&gt;=</code>	greater than or equal to
<code>&lt;</code>	less than
<code>&lt;=</code>	less than or equal to
<code>%in%</code>	contained in

These operators may be combined with `&` (and) or `|` (or).

## 5.3 Exercise 2: Peak popularity of your name

In this exercise you will discover the year your name reached its maximum popularity.

Read in the “babyNames.csv” file if you have not already done so, assigning the result to `baby.names`. The file is located at “<http://tutorials.iq.harvard.edu/data/babyNames.csv>”

Make sure you have installed the `dplyr` package and attached it with `library(dplyr)`.

1. Use `filter` to extract data for your name (or another name of your choice).

```
##
```

2. Arrange the data you produced in step 1 above by `Count`. In which year was the name most popular?

```
##
```

3. BONUS (optional): Filter the data to extract *only* the row containing the most popular boys name in 1999.

```
##
```

## 5.4 Exercise 2 solution

*# 1. Use `filter` to extract data for your name (or another name of your choice).*

```
george <- filter(baby.names, Name == "George")
```

*# 2. Arrange the data you produced in step 1 above by `Count`.  
# In which year was the name most popular?*

```
arrange(george, desc(Count))
```

*# 3. BONUS (optional): Filter the data to extract only the  
# row containing the most popular boys name in 1999.*

```
boys.1999 <- filter(baby.names,  
                    Year == 1999 & Sex == "Boys")
```

```
filter(boys.1999, Count == max(Count))
```



## Chapter 6

# Plotting baby name trends over time

It can be difficult to spot trends when looking at summary tables. Plotting the data makes it easier to identify interesting patterns.

R has decent plotting tools built-in – see e.g., `help(plot)`. However, To make things easier on ourselves we will use a *contributed package* called `ggplot2` instead.

```
## install.packages("ggplot2")  
library(ggplot2)
```

For quick and simple plots we can use the `qplot` function. For example, we can plot the number of babies given the name “Diana” over time like this:

```
diana <- filter(baby.names, Name == "Diana")
```

```
qplot(x = Year, y = Count,  
      data = diana)
```

Interestingly there are usually some gender-atypical names, even for very strongly gendered names like “Diana”. Splitting these trends out by Sex is very easy:

```
qplot(x = Year, y = Count, color = Sex,  
      data = diana)
```

### 6.1 Exercise 3: Plotting peak popularity of your name

Make sure the `ggplot2` package is installed, and that you have attached it using `library(ggplot2)`.

1. Use `filter` to extract data for your name (same as previous exercise)

```
##
```

2. Plot the data you produced in step 1 above, with `Year` on the x-axis and `Count` on the y-axis.

```
##
```

3. Adjust the plot so that it shows boys and girls in different colors.

```
##
```

4. BONUS (Optional): Adjust the plot to use lines instead of points.

## 6.2 Exercise 3 solution

```
# 1. Use `filter` to extract data for your name (same as previous exercise)
```

```
george <- filter(baby.names, Name == "George")
```

```
# 2. Plot the data you produced in step 1 above, with `Year` on the x-axis  
# and `Count` on the y-axis.
```

```
qplot(x = Year, y = Count, data = george)
```

```
# 3. Adjust the plot so that it shows boys and girls in different colors.
```

```
qplot(x = Year, y = Count, color = Sex, data = george)
```

```
# 4. BONUS (Optional): Adjust the plot to use lines instead of points.
```

```
qplot(x = Year, y = Count, color = Sex, data = george, geom = "line")
```



## Chapter 7

# Finding the most popular names

Our next goal is to find out which names have been the most popular.

### 7.1 Computing better measures of popularity

So far we've used `Count` as a measure of popularity. A better approach is to use proportion or rank to avoid confounding popularity with the number of babies born in a given year.

The `mutate` function makes it easy to add or modify the columns of a `data.frame`. For example, we can use it compute the log of the number of boys and girls given each name in each year:

```
baby.names <- mutate(baby.names, logCount = Count/1000)
baby.names
```

### 7.2 Operating by group

Because of the nested nature of our data, we want to compute rank or proportion within each `Sex X Year` group. The `dplyr` package makes this relatively easy.

```
baby.names <- mutate(group_by(baby.names, Year, Sex),
  Rank = rank(Count))
```

Note that the data remains grouped until you change the groups by running `group_by` again or remove grouping information with `ungroup`.

### 7.3 Exercise 4: Most popular names

In this exercise your goal is to identify the most popular names for each year.

1. Use `mutate` and `group_by` to create a column named “Proportion” where `Proportion = Count/sum(Count)` for each `Year X Sex` group.

```
##
```

2. Use `mutate` and `group_by` to create a column named “Rank” where `Rank = rank(-Count)` for each `Year X Sex` group.

```
##
```

3. Filter the baby names data to display only the most popular name for each `Year X Sex` group.

```
##
```

4. Plot the data produced in step 4, putting `Year` on the x-axis and `Proportion` on the y-axis. How has the proportion of babies given the most popular name changed over time?

```
##
```

5. BONUS (optional): Which names are the most popular for both boys and girls?

## 7.4 Exercise 4 solution

```
## 1. Use `mutate` and `group_by` to create a column named "Proportion"
##      where `Proportion = Count/sum(Count)` for each `Year X Sex` group.
```

```
baby.names <- mutate(group_by(baby.names, Year, Sex),
                     Proportion = Count/sum(Count))
```

```
## 2. Use `mutate` and `group_by` to create a column named "Rank" where
##      `Rank = rank(-Count)` for each `Year X Sex` group.
```

```
baby.names <- mutate(group_by(baby.names, Year, Sex),
                     Rank = rank(-Count))
```

```
## 3. Filter the baby names data to display only the most popular name
##      for each `Year X Sex` group.
```

```
top1 <- filter(baby.names, Rank == 1)
```

```
## 4. Plot the data produced in step 3, putting `Year` on the x-axis  
##      and `Proportion` on the y-axis. How has the proportion of babies  
##      given the most popular name changed over time?
```

```
qplot(x = Year, y = Proportion, color = Sex,  
      data = top1,  
      geom = "line")
```

```
## 5. BONUS (optional): Which names are the most popular for both boys  
##      and girls?
```

```
girls.and.boys <- inner_join(filter(baby.names, Sex == "Boys"),  
                             filter(baby.names, Sex == "Girls"),  
                             by = c("Year", "Name"))
```

```
girls.and.boys <- mutate(girls.and.boys,  
                          Product = Count.x * Count.y,  
                          Rank = rank(-Product))
```

```
filter(girls.and.boys, Rank == 1)
```



## Chapter 8

# Percent choosing one of the top 10 names

You may have noticed that the percentage of babies given the most popular name of the year appears to have decreases over time. We can compute a more robust measure of the popularity of the most popular names by calculating the number of babies given one of the top 10 girl or boy names of the year.

In order to compute this measure we need to operate within groups, as we did using `mutate` above, but this time we need to collapse each group into a single summary statistic. We can achieve this using the `summarize` function. For example, we can calculate the number of babies born each year:

```
bn.by.year <- summarize(group_by(baby.names, Year),  
                          Total = sum(Count))  
bn.by.year
```

### 8.1 Exercise 4: Popularity of the most popular names

In this exercise we will plot trends in the proportion of boys and girls given one of the 10 most popular names each year.

1. Filter the `baby.names` data, retaining only the 10 most popular girl and boy names for each year.

```
##
```

2. Summarize the data produced in step one to calculate the total Proportion of boys and girls given one of the top 10 names each year.

```
##
```

3. Plot the data produced in step 2, with year on the x-axis and total proportion on the y axis. Color by sex.

```
##
```

## 8.2 Exercise 4 solution

```
## 1. Filter the baby.names data, retaining only the 10 most  
##     popular girl and boy names for each year.
```

```
most.popular <- filter(group_by(baby.names, Year, Sex),  
                        Rank <= 10)
```

```
## 2. Summarize the data produced in step one to calculate the total  
##     Proportion of boys and girls given one of the top 10 names  
##     each year.
```

```
top10 <- summarize(group_by(most.popular, Year, Sex),  
                    TotalProportion = sum(Proportion))
```

```
## 3. Plot the data produced in step 2, with year on the x-axis  
##     and total proportion on the y axis. Color by sex.
```

```
qplot(x = Year, y = TotalProportion, color = Sex,  
       data = top10,  
       geom = "line")
```

## Chapter 9

# Saving our Work

Now that we have made some changes to our data set, we might want to save those changes to a file.

### 9.1 Saving individual datasets

```
# write data to a .csv file  
write_csv(baby.names, "babyNames.csv")
```

```
# write data to an R file  
write_rds(baby.names, "babyNames.rds")
```

### 9.2 Saving and loading R workspaces

In addition to importing individual datasets, R can save and load entire workspaces

```
ls() # list objects in our workspace  
save.image(file="myWorkspace.RData") # save workspace  
rm(list=ls()) # remove all objects from our workspace  
ls() # list stored objects to make sure they are deleted
```

```
## Load the "myWorkspace.RData" file and check that it is restored  
load("myWorkspace.RData") # load myWorkspace.RData  
ls() # list objects
```





# Chapter 10

## Wrap-up

### 10.1 Help us make this workshop better!

Please take a moment to fill out a very short feedback form. These workshops exist for you – tell us what you need! <http://tinyurl.com/R-intro-feedback>

### 10.2 Additional resources

- IQSS workshops: [http://projects.iq.harvard.edu/rtc/filter\\_by/workshops](http://projects.iq.harvard.edu/rtc/filter_by/workshops)
- IQSS statistical consulting: <http://dss.iq.harvard.edu>
- Software (all free!):
  - R and R package download: <http://cran.r-project.org>
  - Rstudio download: <http://rstudio.org>
  - ESS (emacs R package): <http://ess.r-project.org/>
- Online tutorials
  - <http://www.codeschool.com/courses/try-r>
  - <http://www.datacamp.org>
  - <http://swirlstats.com/>
  - <http://r4ds.had.co.nz/>
- Getting help:
  - Documentation and tutorials: <http://cran.r-project.org/other-docs.html>
  - Recommended R packages by topic: <http://cran.r-project.org/web/views/>
  - Mailing list: <https://stat.ethz.ch/mailman/listinfo/r-help>
  - StackOverflow: <http://stackoverflow.com/questions/tagged/r>
- Coming from... Stata : <http://www.princeton.edu/~otorres/RStata.pdf>  
SAS/SPSS : <http://www.et.bs.ehu.es/~etptupaf/pub/R/RforSAS&SPSSusers.pdf>  
matlab : <http://www.math.umaine.edu/~hiebler/comp/matlabR.pdf> Python  
: <http://mathesaurus.sourceforge.net/matlab-python-xref.pdf>



# Part II

# Python



# Chapter 11

## Introduction to Python

**Topics** Assignment; Function arguments; Finding help; Reading data; Filtering and arranging data; Conditional operations; Saving data.

### 11.1 Setup instructions

#### 11.1.1 Install the Anaconda Python distribution

If using your own computer please install the Anaconda Python distribution from <https://www.anaconda.com/download/>. (Note that Python version  $\leq 3.0$  differs considerably from more recent releases. For this workshop you will need version  $\geq 3.4$ .)

Accepting the defaults proposed by the Anaconda installer is generally recommended.

#### 11.1.2 Download workshop materials

Download the materials from <http://tutorials.iq.harvard.edu/Python/PythonIntro.zip> and extract the zipped directory (Right-click => Extract All on Windows, double-click on Mac).

#### 11.1.3 Launch Jupyter Notebook

Start the **Anaconda Navigator** program in the usual way. Click the or **Launch** button under **Jupyter Notebook**.

### 11.2 Workshop goals and approach

In this workshop you will - learn about the python package and application ecosystem, - learn python language basics and common idioms, and, - practice reading files and manipulating

data in python.

A more general goal is to get you comfortable with Python so that it seems less scary and mystifying than it perhaps does now. Note that this is by no means a complete or thorough introduction to Python! It's just enough to get by.

This workshop is relatively *informal*, *example-oriented*, and *hands-on*. We won't spend much time examining language features in detail. Instead we will work through an example, and learn some things about the language along the way.

As an example project we will analyze the text of Lewis Carroll's *Alice's Adventures in Wonderland*. Among the questions we will use Python to answer are: - How many total and unique words are there? - How many chapters and paragraphs? - How many words are in each chapter, and what is the average words per chapter? - How many times is each main character mentioned?

## 11.3 What is Python?

Python is a relatively easy to learn general purpose programming language. People use Python to manipulate, analyze, and visualize data, make web sites, write games, and much more. Youtube, DropBox, and BitTorrent are among the things people used python to make.

Like most popular open source programming languages, Python can be thought of as a *platform* that runs a huge number and variety of packages. The language itself is mostly valuable because it makes it easy to create and use a large number of useful packages.

## 11.4 How can I interact with Python?

A number of interfaces designed to make it easy to interact with Python are available. The Anaconda distribution that we installed earlier includes both a web-based *Jupyter Notebook* and a more conventional Integrated Development Environment called *Spyder*. For this workshop I encourage you to use *Jupyter Notebook*. In real life you should experiment and choose the interface that you find most comfortable.

To get started, start the *Jupyter Notebook* application, and navigate to the *PythonIntro* directory you downloaded and extracted earlier. Start a new notebook by clicking **New => Python 3** as shown below.

A Jupyter Notebook contains one or more *cells* containing notes or code. To insert a new cell click the + button in the upper left. To execute a cell, select it and press **Control+Enter** or click the Run button at the top.

```
knitr::opts_chunk$set(eval = FALSE, results = FALSE, message = FALSE, warning = FALSE, error = F
```

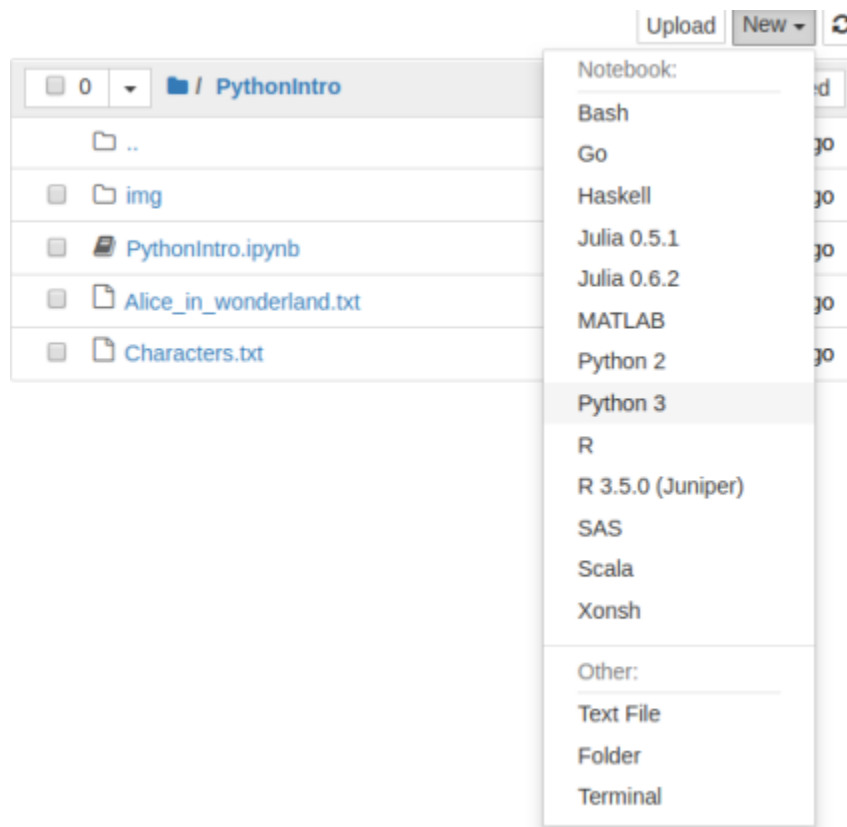


Figure 11.1: notebook\_new.png

## 11.5 Reading the text of Alice in Wonderland from a file

Reading information from a file is the first step in many projects, so we'll start there. The workshop materials you downloaded earlier include a file named `Alice_in_wonderland.txt` which contains the text of Lewis Carroll's *Alice's Adventures in Wonderland*.

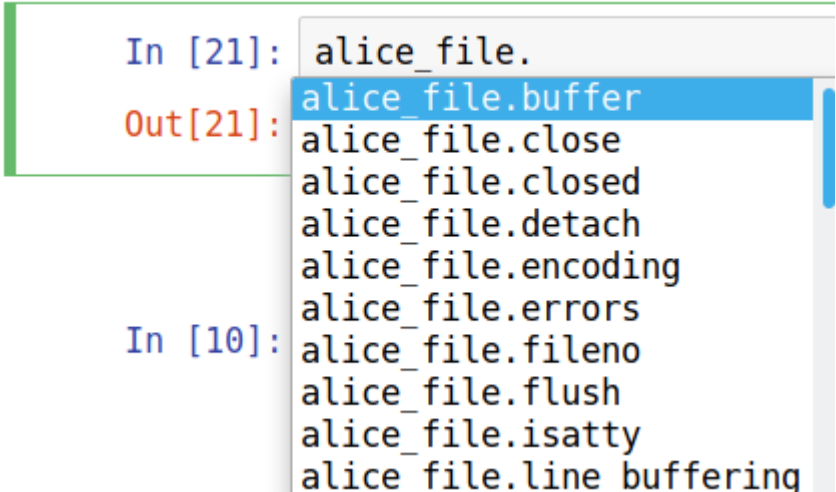
We can open a connection to a file using the `open` function, and store the result using the `=` operator.

```
alice_file = open("Alice_in_wonderland.txt")
```

The name on the left of the equals sign (`alice_file`) is one that we chose. When choosing names, *start with a letter*, and use only *letters, numbers and underscores*.

The `alice_file` object we just created does *not* contain the contents of `Alice_in_wonderland.txt`. It is a representation in Python of the *file itself* rather than the *contents* of the file.

The `alice_file` object provides *methods* that we can use to do things with it. Methods are invoked using syntax that looks like `ObjectName.method()`. You can see the methods available for acting on an object by typing the object's name followed by a `.` and pressing the `tab` key. For example, typing `alice_file.` and pressing `tab` will display a list of methods



The screenshot shows a Jupyter Notebook interface. On the left, the input prompt 'In [21]:' is followed by 'alice\_file.'. On the right, the output prompt 'Out[21]:' is followed by a list of methods: 'alice\_file.buffer', 'alice\_file.close', 'alice\_file.closed', 'alice\_file.detach', 'alice\_file.encoding', 'alice\_file.errors', 'alice\_file.fileno', 'alice\_file.flush', 'alice\_file.isatty', and 'alice\_file.line buffering'. The 'alice\_file.buffer' method is highlighted with a blue background. Below this, the input prompt 'In [10]:' is followed by 'alice\_file.', and the same list of methods is shown, indicating a second tab completion attempt.

as shown below.

Among the methods we have for doing things with our `alice_file` object is one named `read`. We can use the `help` function to learn more about it.

```
help(alice_file.read)
```

Since `alice_file.read` looks promising, we will invoke this method and see what it does.



```
alice_txt = alice_file.read()
print(alice_txt[:500]) # the [:500] gets the first 500 character -- more on this later.
```

That's all there is to it! We've read the contents of `Alice_in_wonderland.txt` and stored this text in a Python object we named `alice_txt`. Now let's start to explore this object, and learn some more things about Python along the way.

## 11.6 Counting chapters, lines, and words

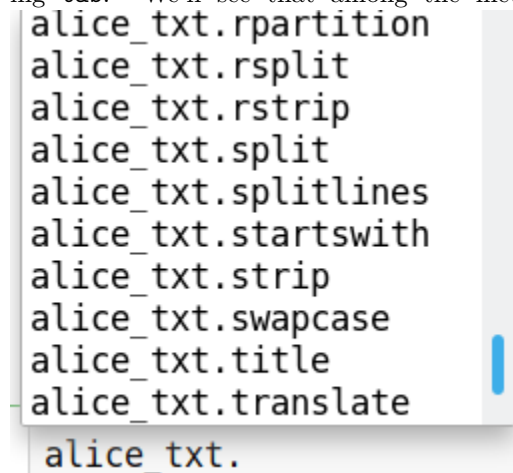
Now that we have the text we can start answering some questions about it. To begin with, how many words does it contain? To answer this question we can split the text up so there is one element per word, and then count the number of words.

### 11.6.1 Splitting a string into a list of words

How do we figure out how to split strings in Python? By asking Python what our `alice_txt` object is and what methods it provides. We can ask Python what things are using the `type` function, like this:

```
type(alice_txt)
```

Python tells us that `alice_txt` is of type `str` (i.e., it is a string). We can find out what methods are available for working strings by typing `alice_txt.` and pressing `tab`. We'll see that among the methods is one named `split`, as shown below.



```
alice_txt.rpartition
alice_txt.rsplitt
alice_txt.rstrip
alice_txt.split
alice_txt.splitlines
alice_txt.startswith
alice_txt.strip
alice_txt.swapcase
alice_txt.title
alice_txt.translate
alice_txt.
```

To learn how to use this method we can check the documentation.

```
help(alice_txt.split)
```

Since the default is to split on whitespace (spaces, newlines, tabs) we can get a reasonable word count simply by calling the `split` method and counting the number of elements in the result.

```
alice_words = alice_txt.split()
len(alice_words)
```

## 11.6.2 Using sets to calculate the number of unique words

According to our computation above, there are about 26 thousand total words in *Alice's Adventures in Wonderland*. But how many *unique* words are there? Python has a special data structure called a *set* that makes it easy to find out. A *set* drops all duplicates, giving a collection of the unique elements.

```
len(set(alice_words))
```

There are 5295 unique words in the text.

## 11.7 Exercise: Reading text from a file and splitting

*Alice's Adventures in Wonderland* is full of memorable characters. The main characters from the story are listed, one-per-line, in the file named `Characters.txt`.

NOTE: we will not always explicitly demonstrate everything you need to know in order to complete an exercise. Instead we focus on teaching you how to discover available methods and how use the help function to learn how to use them. It is expected that you will spend some time during the exercises looking for appropriate methods and perhaps reading documentation.

1. Open the `Characters.txt` file and read its contents.
2. Split text on newlines to produce a list with one element per line. Store the result as "alice\_characters".

““

### 11.7.1 Working with lists

The `split` methods we used to break up the text of *Alice in Wonderland* into words produced a *list*. A lot of the techniques we'll use later to analyze this text also produce lists, so its worth taking a minute to learn more about them.

It is always a good idea to know what type of things you're working with in Python. As you gain experience, you won't have to look this things up as often, but even experienced Python programmers use the `type` function to learn about the objects they are working with.

```
type(alice_words)
```

A *list* in Python is used to store a collection of items. As with other types in Python, you can get a list of methods by typing the name of the object followed by a `.` and pressing **tab**.

#### 11.7.1.1 Extracting subsets from lists

Among the things you can do with a list is extract subsets using bracket indexing notation. This is useful in many situations, including the current one where we want to inspect a long list without printing out the whole thing.

The examples below show how indexing works in Python.

```
alice_words[0] # first word (yes, we count from zero!)
```

```
alice_words[1] # second word
```

```
alice_words[:10] # first 10 words
```

```
alice_words[10:20] # words 11 through 20
```

```
alice_words[-1] # the last word
```

```
alice_words[-10:] # the last 10 words
```

Note that the displayed representation of lists and other data structures in python often closely matches the syntax used to create them. For example, we can create a list using square brackets, just as we see when we print a list:

```
['her',  
 'own',  
 'child-life',  
 'and',  
 'the',  
 'happy',  
 'summer',  
 'days.',  
 'THE',  
 'END']
```

#### 11.7.1.2 Sorting and other in-place methods

There are many other things we can do with lists besides extracting subsets using bracket indexing. For example, there are methods to append and remove elements from a list.

When using a list method that you are unfamiliar with, it is always a good idea to read the documentation.

Note that many methods modify the object *in place*. For example, if we wanted to sort the last 10 words in `alice_words` we would do it like this:

```
last_10 = alice_words[-10:]
print(last_10)
last_10.sort()
print(last_10)
```

### 11.7.2 Counting chapters and paragraphs

Now that we know how to split a string and how to work with the resulting list, we can split on chapter markers to count the number of chapters. All we need to do is specify the string to split on. Since each chapter is marked with the string 'CHAPTER ' followed by the chapter number, we can split the text up into chapters using this as the separator.

```
alice_chapters = alice_txt.split("CHAPTER ")
len(alice_chapters)
```

Since the first element contains the material *before* the first chapter, this tells us there are twelve chapters in the book.

We can count paragraphs in a similar way. Paragraphs are indicated by a blank line, i.e., two newlines in a row. When working with strings we can represent newlines with `\n`, so our basic paragraph separator is `\n\n`.

```
alice_paragraphs = alice_txt.split("\n\n")
```

Before counting the number of paragraphs, I want to inspect the result to see if it looks correct:

```
print(alice_paragraphs[0], "\n=====")
print(alice_paragraphs[1], "\n=====")
print(alice_paragraphs[2], "\n=====")
print(alice_paragraphs[3], "\n=====")
print(alice_paragraphs[4], "\n=====")
print(alice_paragraphs[5], "\n=====")
```

We're counting the title, author, and chapter lines as paragraphs, but this will do for a rough count.

```
len(alice_paragraphs)
```

## 11.8 Exercise: count the number of main characters

So far we've learned that there are 12 chapters, around 830 paragraphs, and about 26 thousand words in *Alice's Adventures in Wonderland*. Along the way we've also learned how to open a file and read its contents, split strings, calculate the length of objects, discover methods for string and list objects, and index/subset lists in Python. Now it is time for you to put these skills to use to learn something about the main characters in the story.

1. Count the number of main characters in the story (i.e., get the length of the list you created in previous exercise).
2. Extract and print just the first character from the list you created in the previous exercise.
3. (BONUS, optional): Sort the list you created in step 2 alphabetically, and then extract the last element.

## 11.9 Working with nested structures: words within paragraphs within chapters

This far our analysis has treated the text as a “flat” data structure. For example, when we counted words we just counted words in the whole document, rather than counting the number of words in each chapter. If we want to treat our document as a nested structure, with words forming sentences, sentences forming paragraphs, paragraphs forming chapters, and chapters forming the book, we need to learn some additional tools. Specifically, we need to learn how to iterate over lists (or other collections) and do things with each element in a collection.

There are several ways to iterate in Python, of which we will focus on *for loops* and *list comprehensions*.

### 11.9.1 Iterating over paragraphs using for-loops

A *for loop* is a way of cycling through the elements of a collection and doing something with each one. As a simple example, we can cycle through the first 6 paragraphs and print each one. Cycling through with a loop makes it easy to insert a separator between the paragraphs, making it much easier to read the output.

```
for paragraph in alice_paragraphs[:6]:
    print(paragraph)
    print('=====')
print('DONE.')
```

Notice that the syntax of a for-loop is

```
for <thing> in <collection>:  
    do stuff with <thing>
```

Notice also that the body of the for-loop is indented. This is important, because it is this indentation that defines the body of the loop. Notice that “DONE.” is only printed once, since `print('DONE.')` is not indented and is therefore outside of the body of the loop.

Loops in Python are great because the syntax is relatively simple, and because they are very powerful. Inside of the body of a loop you can use all the tools you use elsewhere in python.

Here is one more example of a loop, this time iterating over all the chapters and calculating the number of paragraphs in each chapter.

```
for chapter in alice_chapters[1:]:  
    paragraphs = chapter.split("\n\n")  
    print(len(paragraphs))
```

### 11.9.2 Iterating and collecting paragraphs per chapter using list comprehension

We could use for-loops to fill in lists of values, but there is a special syntax in Python that is often better for this use case. This special syntax is called a *list comprehension* and it looks like this:

```
paragraphs_per_chapter = [len(chapter.split("\n\n"))  
                           for chapter in alice_chapters[1:]]  
print(paragraphs_per_chapter)
```

Notice that *list comprehension* is very similar to a *for loop*, though the order is different. In a *for-loop* the `for` part comes first and the expressions that make up the body come second and are indented. In a *list comprehension* the expression comes first and the `for` part comes afterward. Notice also the square brackets surrounding the whole thing – these brackets are what tells Python that you want a list.

Here is another list comprehension that counts the number of times the name “Alice” appears in each chapter.

```
alices_per_chapter = [chapter.count("Alice") for chapter in alice_chapters]  
print(alices_per_chapter)
```

### 11.9.3 Organizing results in dictionaries

Our code for calculating the number of of times “Alice” was mentioned per chapter worked, but with a little effort we can make it much easier to interpret by associating each count with the chapter it corresponds to. In Python we can use a `dict` (i.e., “dictionary”) to store key-value pairs.

First, we can iterate over each chapter and grab just the first line (that is, the chapter titles). These will become our keys.

```
chapter_names = [chapter.splitlines()[0] for chapter in alice_chapters[1:]]
print(chapter_names)
```

Finally we can combine the chapter titles and counts and convert them to a dictionary.

```
dict(zip(chapter_names,
        [chapter.count("Alice")
         for chapter in alice_chapters]))
```

## 11.10 Exercise: Iterating and counting things

Now that we know how to iterate using for-loops and list comprehensions the possibilities really start to open up. For example, we can use these techniques to count the number of times each character appears in the story.

1. Make sure you have both the text and the list of characters.

Open and read both “Alice\_in\_wonderland.txt” and “Characters.txt” if you have not already done so.

2. Which chapter has the most words?

Split the text into chapters (i.e., split on “CHAPTER”) and use a for-loop or list comprehension to iterate over the chapters. For each chapter, split it into words and calculate the length.

3. How many times is each character mentioned in the text?

Iterate over the list of characters using a for-loop or list comprehension. For each character, call the count method with that character as the argument.

4. (BONUS, optional): Put the character counts computed above in a dictionary with character names as the keys and counts as the values.
5. (BONUS, optional): Use a nested list comprehension to calculate the number of times each character is mentioned in each chapter.

## 11.11 Importing numpy and calculating simple statistics

Now that we know how to iterate over lists and calculate numbers for each element, we may wish to do some simple math using these numbers. For example, we may want to calculate the mean and standard deviation of the distribution of the number of paragraphs in each chapter. Python has a handful of math functions built-in (e.g., `min` and `max`) but built-in math support is pretty limited.

When you find that something isn't available in Python itself, it's time to look for a package that does it. Although it is somewhat overkill for simply calculating a mean we're going to use a popular package called *numpy* for this. The *numpy* package is included in the Anaconda Python distribution we are using, so we don't need to install it separately.

In order to use *numpy* or other packages, you must first import them. We can import numpy as follows:

```
import numpy
```

The *numpy* package is very popular and includes a lot of useful functions. For example, we can use it to calculate means and standard deviations:

```
print(numpy.mean(paragraphs_per_chapter))  
print(numpy.std(paragraphs_per_chapter))
```

and compute correlations:

```
words_per_chapter = [len(chapter.split()) for chapter in alice_chapters]  
alices_per_chapter = [chapter.count("Alice") for chapter in alice_chapters]  
  
print(numpy.corrcoef(words_per_chapter, alices_per_chapter))
```

## 11.12 Where to go from here

By this time you've learned a lot about python, including how to read files, call functions, lookup and use methods, process text, manipulate lists and dictionaries, and iterate using loops and comprehensions. There is more to learn, but you probably know enough already to be dangerous. Your next steps are to a) keep learning Python basics and b) find and learn how to use packages that help you accomplish your substantive goals.

Here are some packages you might be interested in learning:

- Graphics
  - matplotlib
  - seaborn



- plotly
- Quantitative Data Analysis
  - numpy
  - scipy
  - pandas
  - scikit-learn
  - statsmodels
- Text analysis
  - textblob
  - nltk
  - Gensim
- Webscraping
  - scrapy
  - requests
  - lxml
  - BeautifulSoup
- Social Network Analysis
  - networkx
  - graph-tool