

Block 02 Data

Structured, Semi-Structured, and Unstructured Data

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:

<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-2>



There are broadly three categories of data: *structured*, *semi-structured* and *unstructured*. This categorization is useful as it can direct as to what solutions may be used for storing and analyzing the various kinds of data, or what steps need to be taken in order to analyze the data. However, this categorization is not as clear-cut as one would hope, and there are data sets that are a combination of structured, unstructured and semi-structured data.

Structured Data

Structured data is data that is organized according to a predetermined set of rules. A characteristic of structured data is that they are relatively straightforward to read, filter and analyze, and that there is a wealth of statistical and machine learning procedures for directly drawing insights from them. Structured data are typically represented in tabular format, in one or more different tables, with predetermined relationships between the different rows and columns. For example, the data that a smartwatch or a smartphone collects when someone is jogging includes records of latitude, longitude, altitude, distance run, heart rate, and current speed at regular times. So, we can organize the data in a tabular format as follows

	timestamp	latitude	longitude	altitude	distance	heart_rate
speed						
	2013-06-01 18:40:29	50.81381	-1.712606	80.20001	1805.94	133
4.060059						
	2013-06-01 18:40:30	50.81383	-1.712649	80.00000	1810.00	133
4.550049						
	2013-06-01 18:40:31	50.81385	-1.712700	79.79999	1814.55	133
2.979981						
	2013-06-01 18:40:32	50.81387	-1.712734	79.79999	1817.53	133
2.969971						

2013-06-01 18:40:33	50.81388	-1.712777	79.59998	1820.50	133
3.650024					
2013-06-01 18:40:34	50.81389	-1.712826	79.59998	1824.15	133
3.229980					
2013-06-01 18:40:35	50.81391	-1.712862	79.40002	1827.38	133
4.650024					
2013-06-01 18:40:36	50.81393	-1.712911	79.40002	1832.03	133
4.149902					
2013-06-01 18:40:37	50.81395	-1.712963	79.20001	1836.18	133
2.000000					
2013-06-01 18:40:38	50.81395	-1.712994	79.20001	1838.18	133
4.210083					
2013-06-01 18:40:39	50.81396	-1.713053	79.00000	1842.39	133
5.189941					

The organization of the data in this way prescribes that the set of values recorded by the smartwatch or smartphone at each timestamp will be organized in rows, and each row will consist of the values that each variable takes, organized in a way so that each column of the resulting table contains the values of one and only one variable. We also know exactly how to organize a new record of the same information in the table. The existence of a predetermined set of rules for structuring the data makes it easier to organize, search, and analyze the information within it. To appreciate this, try to quickly spot all records after 18:40:36 on 2013-06-01, or determine whether at those timestamps the average speed is greater than 3 metres per second.

There is a wealth of structured data sets generated both by machines and humans. Examples of structured data sets that you have encountered include stock prices, addresses, identity information (e.g. social security numbers, names, surname, etc), sensory data produced by smartphones (such as the above example), flight arrival and departure details, etc.

[Relational database](#) management systems (RDBMS), like SQL that you will encounter later, are very powerful tools for the efficient storing and handling of structured data (e.g. searching, filtering, joining, sorting etc).

Unstructured Data

Unstructured data sets are data sets for which it is difficult to have a predetermined set of rules for organizing them. Examples include images, videos, sounds (music tracks for example), text, websites, books. As a result the vast majority of the data that we interact with nowadays is unstructured!

You can appreciate the challenges involved with unstructured data if you try a simple exercise: organize the text in this paragraph in a tabular format. You will see that there is more than one way of doing so. Would you organize the text by words, by sentences, by letters? Would you keep the order the words appear in? What happens with punctuation?

Clearly, the lack of structure makes unstructured data more difficult to search, organize and analyze. This is the reason why many machine learning and statistical procedures for unstructured data rely on the appropriate transformation of unstructured data into a structured form. For example, [topic models](#) can be used for the discovery of topics or themes in massive and, otherwise unstructured, collections of documents (see, for example, Blei, 2012 for a review of basic topic models). While topic models provide insights for unstructured data,

they typically require the transformation of unstructured text into a [document-term matrix](#) which is then structured data! As an example in R, we use the [tm](#) R package to convert text from four documents into a document-term matrix.

```
library("tm")
doc1 <- "I love programming in R and hate programming in Python"
doc2 <- "I love programming in Python and hate programming in R"
doc3 <- "I love programming in Python and R"
doc4 <- "I hate programming"
## Build a corpus and a document-term matrix
corpus <- Corpus(VectorSource(c(doc1, doc2, doc3, doc4)))
dt_mat <- DocumentTermMatrix(corpus)
as.matrix(dt_mat)
```

	Terms					
Docs	and	hate	love	programming	python	
1	1	1	1	2	1	
2	1	1	1	2	1	
3	1	0	1	1	1	
4	0	1	0	1	0	

Of course, the document-term matrix is nowhere close to capturing all the information that is in the unstructured text!

The specifics of topic modelling are beyond the scope of this course, but if you want to run the above commands on your own, you first need to type `install.packages("tm")` in R in order to install the **tm** R package. Also, if you are curious, you can browse through the help pages of `Corpus()`, `VectorSource()`, and `DocumentTermMatrix()` (e.g. by typing `?Corpus`, `?VectorSource`, and `?DocumentTermMatrix`, respectively) to see what each of these functions does.

Semi-Structured Data

Semi-structured data does not conform to relational databases such SQL, but still contains some level of organization through semantic elements like tags and metadata. An example of semi-structured data are Markdown documents! For example, the contents of the `hello-Rmd.Rmd` R Markdown file that we encountered in previous weeks are

```
# My first R Markdown file
```

```
After a hard training day with Yoda, I decided to author my first [R
Markdown](https://rmarkdown.rstudio.com) file. This is a text chunk
written in *Markdown syntax*. I can write bold and italic, and
even record quotes I want to remember like
```

```
> *Do. Or do not. There is no try*
>
> Yoda, The Empire Strikes Back
```

I can also ask R to run code and return the results. For example, I can ask R to print the quote

```
`{r quote}
print("Do. Or do not. There is no try")
`
```

I can also do complex arithmetic. For example, if your R installation could do infinite arithmetic you could see that `1/81` has all single digits numbers from 0 to 9 repeating in its decimal, except 8!

```
```{r arithmetic}
print(1/81, 15)
```
```

Clearly, there is a quite a bit of unstructured data in the above, like free text and code. But, there is a certain level of organization of R Markdown files, by using `#` for titles, `>` for quote markers and ````` for code blocks, `[]()` for links and so on. So, extracting the code from an R Markdown document, or counting the number of links, paragraphs, sections or quotes is a relatively straightforward process.

See [Wikipedia's page on Semi-structured data](#) for examples.

Useful Links and Resources

- [What's the difference between structured, semi-structured and unstructured data?](#) by Bernard Marr: A business perspective on structured, unstructured, and semi-structured data.
- [topicmodels](#) R package: a package with a nice [vignette](#) about topic models.

References

Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, 55(4), 77–84.

File Formats for Data Exchange

Plain Text

The simplest way to store information in a computer is in plain text. Plain text files are perhaps the most basic (and perhaps the most reliable) form of data storage. They are not the most sophisticated or efficient to read and write, but they have the advantage of being readable and writable across platforms and with virtually any text editor. The typical extension for plain text file is `.txt` but they can be also encountered with other extensions or no extension at all.

Just as an example, if you open `nile.txt` file in your favourite text editor — even your source-code editor —, you will see

```
1120 1160 963 1210 1160 1160 813 1230 1370 1140 995 935 1110 994 1020 960
1180 799 958 1140 1100 1210 1150 1250 1260 1220 1030 1100 774 840 874 694
940 833 701 916 692 1020 1050 969 831 726 456 824 702 1120 1100 832 764 821
768 845 864 862 698 845 744 796 1040 759 781 865 845 944 984 897 822 1010
771 676 649 846 812 742 801 1040 860 874 848 890 744 749 838 1050 918 986
797 923 975 815 1020 906 901 1170 912 746 919 718 714 740
```

The data are from Cobb (1978) and represent successive measurements of the annual flow of the river Nile (in 10^8 m^3) at [Aswan](#), between 1871 and 1970. `nile.txt` contains the flow measurements in a single separated by spaces.

If you open the `sunset-salvo` file with a text editor you will see

```
The combination of some data and an aching desire for an answer does not
ensure that a reasonable answer can be extracted from a given body of data
```

This is a quote extracted from a worth-reading academic paper titled “Sunset Salvo” that has been written by one of the most prominent statisticians, John W. Tukey (Tukey, 1986). The data in this file can be either the letters and spaces or the words themselves, if, for example, one wants to model the frequency or order of letters or words.

Delimiter-Separated Values

As we discussed, a widespread way to represent structured data is in two-dimensional arrays. For example, the jogging data we encountered in the previous section is a two-dimensional array:

```
timestamp latitude longitude altitude distance heart_rate
speed
2013-06-01 18:40:29 50.81381 -1.712606 80.20001 1805.94 133
4.060059
2013-06-01 18:40:30 50.81383 -1.712649 80.00000 1810.00 133
4.550049
```

```

2013-06-01 18:40:31 50.81385 -1.712700 79.79999 1814.55      133
2.979981
2013-06-01 18:40:32 50.81387 -1.712734 79.79999 1817.53      133
2.969971
2013-06-01 18:40:33 50.81388 -1.712777 79.59998 1820.50      133
3.650024
2013-06-01 18:40:34 50.81389 -1.712826 79.59998 1824.15      133
3.229980
2013-06-01 18:40:35 50.81391 -1.712862 79.40002 1827.38      133
4.650024
2013-06-01 18:40:36 50.81393 -1.712911 79.40002 1832.03      133
4.149902
2013-06-01 18:40:37 50.81395 -1.712963 79.20001 1836.18      133
2.000000
2013-06-01 18:40:38 50.81395 -1.712994 79.20001 1838.18      133
4.210083
2013-06-01 18:40:39 50.81396 -1.713053 79.00000 1842.39      133
5.189941

```

Such data structures are ubiquitous; each line is a data record (in our case a record coming from the smartwatch or smartphone) and each of these records consists of the values that one of more variables take.

A common way that such structured data are stored and a shared is in text files containing [delimiter-separated values](#). Such files are organized as follows: each line corresponds to a separate record, and the values of the record are separated by a special character (the *delimiter*), with the most common being the comma, tab, and colon.

For example, if you open the *tab-delimited file* (`running_dat.tsv`) with the above data in a text editor you should see

```

"timestamp" "latitude" "longitude" "altitude" "distance" "heart_rate"
"speed"
2013-06-01 18:40:29 50.813805 -1.7126063 80.2000122 1805.9399512 133
4.060058600000005
2013-06-01 18:40:30 50.8138298 -1.7126487 80 1810.0000098 133
4.550048800000001
2013-06-01 18:40:31 50.8138543 -1.7127005 79.7999878 1814.5500586 133
2.979980500000001
2013-06-01 18:40:32 50.8138709 -1.7127338 79.7999878 1817.5300391 133
2.969970699999998
2013-06-01 18:40:33 50.8138757 -1.7127769 79.5999756 1820.5000098 133
3.650024399999989
2013-06-01 18:40:34 50.8138862 -1.712826 79.5999756 1824.1500342 133
3.229980400000016
2013-06-01 18:40:35 50.8139051 -1.7128616 79.4000244 1827.3800146 133
4.650024499999997
2013-06-01 18:40:36 50.8139326 -1.7129115 79.4000244 1832.0300391 133
4.149902299999989
2013-06-01 18:40:37 50.8139517 -1.7129626 79.2000122 1836.1799414 133
2
2013-06-01 18:40:38 50.8139537 -1.7129945 79.2000122 1838.1799414 133
4.210082999999994
2013-06-01 18:40:39 50.8139622 -1.7130533 79 1842.3900244 133
5.189941400000018

```

If you open the *comma delimited* file (see `running_dat.csv`) with the above data in a text editor you should see

```
"timestamp","latitude","longitude","altitude","distance","heart_rate","speed"
2013-06-01 18:40:29,50.813805,-
1.7126063,80.2000122,1805.9399512,133,4.060058600000005
2013-06-01 18:40:30,50.8138298,-
1.7126487,80,1810.0000098,133,4.550048800000001
2013-06-01 18:40:31,50.8138543,-
1.7127005,79.7999878,1814.5500586,133,2.979980500000001
2013-06-01 18:40:32,50.8138709,-
1.7127338,79.7999878,1817.5300391,133,2.969970699999998
2013-06-01 18:40:33,50.8138757,-
1.7127769,79.5999756,1820.5000098,133,3.650024399999989
2013-06-01 18:40:34,50.8138862,-
1.712826,79.5999756,1824.1500342,133,3.229980400000016
2013-06-01 18:40:35,50.8139051,-
1.7128616,79.4000244,1827.3800146,133,4.650024499999997
2013-06-01 18:40:36,50.8139326,-
1.7129115,79.4000244,1832.0300391,133,4.149902299999989
2013-06-01 18:40:37,50.8139517,-1.7129626,79.2000122,1836.1799414,133,2
2013-06-01 18:40:38,50.8139537,-
1.7129945,79.2000122,1838.1799414,133,4.210082999999994
2013-06-01 18:40:39,50.8139622,-
1.7130533,79,1842.3900244,133,5.189941400000018
```

`running_dat.tsv` is known as a *tab-separated values* file (TSV file, in short) and `running_dat.csv` is a *comma-separated values* file (CSV file, in short).

TSV, and particularly CSV, files are widely used for storing and sharing data, and can be opened by many applications, including most spreadsheet programs (like [Microsoft Excel](#) and [Libre Office](#) Calc), and have excellent support in many programming languages (including R and Python). Despite the C in “CSV file” standing for comma, you may encounter files with “.csv” extension using different delimiters (e.g. colon, semi-colon, or tab).

You may be wondering what happens if one of the data values is or includes the delimiter; this is a well-studied topic known as *delimiter collision* and most modern spreadsheet applications and programming languages try hard to deal with it; see, for example, [Wikipedia’s delimiter page](#) for more information.

XML

[XML \(Extensible Markup Language\)](#) is a markup language (like Markdown is) that defines a set of rules for encoding information in objects called *XML documents*. XML is a format for organizing structured and semi-structured data in a way that is both human-readable and machine-readable. XML documents are typically stored in plain text files with extension `.xml`.

Since its proposal by [W3C](#) in 1998, the XML format has found a diverse range of uses, ranging from a way to store and share structured and semi-structured data, up to defining other popular file formats. In fact, you are using XML files on a daily basis; for example, Word document files with `.docx` extension from recent versions of Microsoft Office are bundles of many XML documents (see this [blog post](#) for more information).

An example of an XML document is in `statisticians.xml`. This XML document provides information and URLs to the Wikipedia pages for some famous statisticians. If you open `statisticians.xml` in a text editor you should see:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of famous statisticians -->
<records>
  <statistician>
    <name>Ronald</name>
    <middle>Aylmer</middle>
    <surname>Fisher</surname>
    <dob>17/02/1890</dob>
    <wiki>https://en.wikipedia.org/wiki/Ronald_Fisher</wiki>
  </statistician>
  <statistician>
    <name>William</name>
    <middle>Sealy</middle>
    <surname>Gosset</surname>
    <dob>13/08/1876</dob>
    <wiki>https://en.wikipedia.org/wiki/William_Sealy_Gosset</wiki>
  </statistician>
  <statistician>
    <name>David</name>
    <middle>Roxbee</middle>
    <surname>Cox</surname>
    <dob>15/07/1924</dob>
    <wiki>https://en.wikipedia.org/wiki/David_Cox_(statistician)</wiki>
  </statistician>
  <statistician>
    <name>Thomas</name>
    <middle></middle>
    <surname>Bayes</surname>
    <dob>07/04/1761</dob>
    <wiki>https://en.wikipedia.org/wiki/Thomas_Bayes</wiki>
  </statistician>
  <statistician>
    <name>Karl</name>
    <middle></middle>
    <surname>Pearson</surname>
    <dob>27/03/1857</dob>
    <wiki>https://en.wikipedia.org/wiki/Karl_Pearson</wiki>
  </statistician>

```



```

<statistician>
  <name>John</name>
  <middle>Wilder</middle>
  <surname>Tukey</surname>
  <dob>16/06/1915</dob>
  <wiki>https://en.wikipedia.org/wiki/John_Tukey</wiki>
</statistician>
</records>

```

Let's introduce some terminology. Clearly, an XML document is a string of characters. XML documents typically begin with an *XML declaration* like `<?xml version="1.0" encoding="UTF-8"?>` above. Such declarations describe information about the XML document. The characters making up the XML document are divided into *markup* and *content*. Whatever is between `<` and `>` is markup, and anything that is not markup is content. For example, in the line `<surname>Tukey</surname>`, `<surname>` and `</surname>` are markup, while `Tukey` is content. `<surname>` and `</surname>` are instances of start and end *tags*, respectively, which are the main tag types XML supports. Also,

```

<statistician>
  <name>Karl</name>
  <middle></middle>
  <surname>Pearson</surname>
  <dob>27/03/1857</dob>
  <wiki>https://en.wikipedia.org/wiki/Karl_Pearson</wiki>
</statistician>

```

is an *element* and

```
<surname>Pearson</surname>
```

is one of its *child elements*. Finally, XML supports comments by enclosing text between `<!--` and `>`. For example, `<!-- A list of famous statisticians -->`.

XML is a great format for storing hierarchical data. For example, as is apparent by the statisticians example above, the contents of an XML document can be graphically represented as a tree. The top node is `records`, which has six *children* nodes `statistician`, each of which has five *children* nodes, `name`, `middle`, `surname`, `dob`, and `wiki`.

An XML parser is a special bit of software that processes XML documents and passes the structured information within them to another application or on a file. Mainstream programming languages, like R and Python, have packages and modules that allow parsing XML files.

More technical details on XML (e.g. metadata and attributes on tags, XML schemas, etc) and extensions (e.g. XML namespaces) can be found at [Wikipedia's XML page](#).

JSON

[JavaScript Object Notation](#) (JSON) is a data interchange file format, which, like XML, aims at being both human-readable and machine-readable. JSON was originally used in JavaScript, but many modern programming languages (like R and Python) provide great support for generation and parsing of JSON-format data.

JSON is seeing a diverse range of technological applications, being used for storing and sharing data sets, for communication between different devices and programs (e.g. server-client communication), for interfacing with [APIs](#), etc. In fact, your web browser and the apps on your mobile phone or tablet are creating, processing, sending and receiving hundreds of JSON files every day. One of the reason for its popularity is that it is a lightweight format to store and transmit information, and easy/fast to parse (using a JSON parser).

For example, `statisticians.json` has the same information as `statisticians.xml`. If you open `statisticians.json` in a text editor you should see

```
[
  {
    "name": "Ronald",
    "middle": "Aylmer",
    "surname": "Fisher",
    "dob": "17/02/1890",
    "wiki": "https://en.wikipedia.org/wiki/Ronald_Fisher"
  },
  {
    "name": "William",
    "middle": "Sealy",
    "surname": "Gosset",
    "dob": "13/08/1876",
    "wiki": "https://en.wikipedia.org/wiki/William_Sealy_Gosset"
  },
  {
    "name": "David",
    "middle": "Roxbee",
    "surname": "Cox",
    "dob": "15/07/1924",
    "wiki": "https://en.wikipedia.org/wiki/David_Cox_(statistician)"
  },
  {
    "name": "Thomas",
    "middle": null,
    "surname": "Bayes",
    "dob": "07/04/1761",
    "wiki": "https://en.wikipedia.org/wiki/Thomas_Bayes"
  },
  {
    "name": "Karl",
    "middle": null,
    "surname": "Pearson",
    "dob": "27/03/1857",
    "wiki": "https://en.wikipedia.org/wiki/Karl_Pearson"
  },
  {
    "name": "John",
    "middle": "Wilder",

```

```

    "surname": "Tukey",
    "dob": "16/06/1915",
    "wiki": "https://en.wikipedia.org/wiki/John_Tukey"
  }
]

```

Clearly, this file contains the same information as `statisticians.xml` but in a less verbose manner (less characters; e.g. there are no end tags), sacrificing a bit in terms of detail that can be transmitted along with the file (e.g. no support for comments or metadata, no namespaces, etc.)

JSON supports some basic variable types, including strings, numbers, Booleans, null, arrays, and objects. For example, the below

```

{
  "language": "Python",
  "release": 1991,
  "os": ["Linux", "macOS", "Windows"],
  "oo": true,
  "pastnames": null
}

```

is a JSON object (what is between `{` and `}`), with five *key-value* combinations; e.g. “language” is a key and “Python” is the value. The value of “release” is a number, the value of “os” is an array, the value of “oo” (object-oriented) is a Boolean, and “pastnames” is null.

More details on JSON and extensions can be found at [Wikipedia’s JSON page](https://en.wikipedia.org/wiki/JSON) and at json.org.

Spreadsheets

A **spreadsheet** is a computer application for the organization, analysis and storage of tabular data. Examples of spreadsheet are Microsoft Excel, LibreOffice Calc or Apple Numbers. They all support a variety of formats for storing tabular data. For example, opening up `running.csv` in Microsoft Excel one sees the following (you get a similar picture if you open `running.csv` in LibreOffice)

| | A | B | C | D | E | F | G | H |
|----|------------------|------------|------------|------------|-------------|------------|-----------|---|
| 1 | timestamp | latitude | longitude | altitude | distance | heart_rate | speed | |
| 2 | 01/06/2013 18:40 | 50.813805 | -1.7126063 | 80.2000122 | 1805.939951 | 133 | 4.0600586 | |
| 3 | 01/06/2013 18:40 | 50.8138298 | -1.7126487 | 80 | 1810.00001 | 133 | 4.5500488 | |
| 4 | 01/06/2013 18:40 | 50.8138543 | -1.7127005 | 79.7999878 | 1814.550059 | 133 | 2.9799805 | |
| 5 | 01/06/2013 18:40 | 50.8138709 | -1.7127338 | 79.7999878 | 1817.530039 | 133 | 2.9699707 | |
| 6 | 01/06/2013 18:40 | 50.8138757 | -1.7127769 | 79.5999756 | 1820.50001 | 133 | 3.6500244 | |
| 7 | 01/06/2013 18:40 | 50.8138862 | -1.712826 | 79.5999756 | 1824.150034 | 133 | 3.2299804 | |
| 8 | 01/06/2013 18:40 | 50.8139051 | -1.7128616 | 79.4000244 | 1827.380015 | 133 | 4.6500245 | |
| 9 | 01/06/2013 18:40 | 50.8139326 | -1.7129115 | 79.4000244 | 1832.030039 | 133 | 4.1499023 | |
| 10 | 01/06/2013 18:40 | 50.8139517 | -1.7129626 | 79.2000122 | 1836.179941 | 133 | 2 | |
| 11 | 01/06/2013 18:40 | 50.8139537 | -1.7129945 | 79.2000122 | 1838.179941 | 133 | 4.210083 | |
| 12 | 01/06/2013 18:40 | 50.8139622 | -1.7130533 | 79 | 1842.390024 | 133 | 5.1899414 | |
| 13 | | | | | | | | |
| 14 | | | | | | | | |
| 15 | | | | | | | | |

Screenshot of Microsoft Excel showing the data in running.csv

The data are opened in a *sheet*, which is part of a *workbook*. A sheet is customarily a two-dimensional array with rows named with numbers and columns named by letters. Every combination of number and letter corresponds to a *cell*, and each value in the data occupies a single cell.

Modern spreadsheet applications allow for the computation of summaries using in-cell formulas, provide data manipulation utilities and tools to carry out a range of statistical analyses and produce visualizations. A description of spreadsheet features is out of the scope of this course. YouTube provides a range of Excel quick tours for a range of levels (e.g. [this one](#) is for absolute beginners and [this one](#) is for more advanced users). Also, LibreOffice provides a range of guides for its applications, including Calc; see [here](#).

Spreadsheets also provide alternative formats for saving tabular data, along with any formulas or visualizations that have been produced. For example, the default file extension for Microsoft Excel is `.xlsx`, for LibreOffice Calc is `.ods` (which are both bundles of XML files), and for Apple Numbers is `.numbers`.

Other Data Formats

There is a wide variety of other file formats (and database systems!) to store structured, semi-structured, and unstructured data. Many file formats are associated with software for the statistical analysis of data, like SAS, SPSS and MINITAB, or have been invented for particular purposes, such as the [nifti](#) data format that is used in neuroimaging applications.

There are also *binary* data formats, that is data formats where the data are not stored or communicated as text, as is done, for example, for CSV, TSV, XML, and JSON. Examples of such data formats are NASA's HDF5 ([hierarchical data format](#)) and UCAR's netCDF data files ([network common data form](#)), which allow users to store scientific data in array-oriented ways, including descriptions, labels, formats, units, etc. Other binary data formats are formats used for images, such PNG ([portable network graphics](#)) and JPEG ([Joint Photographic Experts Group](#) format) and videos, such as MP4 ([MPEG-4](#)).

Useful Links and Resources

- [LibreOffice](#): An advanced office suite that is open-source and can be used as an alternative to Microsoft Office; see [here](#) for documentation.
- [Wikipedia's delimiter page](#)
- [Wikipedia's JSON page](#)
- [Wikipedia's XML page](#)
- [Introduction to XML](#): w3schools.com introduction to XML
- [Introduction to JSON](#): w3schools.com introduction to JSON
- [An informal introduction to DOCX](#) by Stepan Yakovenko: An accessible introduction to the DOCX and the underlying usage of XML files.
- [Introducing JSON](#)
- [Learn JSON in 10 minutes](#): A video providing a very good overview of JSON.

References

- Cobb, G. W. (1978). The problem of the Nile: Conditional solution to a changepoint problem. *Biometrika*, 65(2), 243–251.
- Tukey, J. W. (1986). Sunset salvo. *The American Statistician*, 40(1), 72–76.

Import/Export of Data-Exchange Files in R

Extracting Data From Plain Text

Import

The R function `scan()` can be used to read data from a file. For example, in order to read the measurements in `nile.txt` (see previous section for more details), we do

```
nile <- scan("nile.txt")
nile
 [1] 1120 1160  963 1210 1160 1160  813 1230 1370 1140  995  935 1110  994
1020
[16]  960 1180  799  958 1140 1100 1210 1150 1250 1260 1220 1030 1100  774
840
[31]  874  694  940  833  701  916  692 1020 1050  969  831  726  456  824
702
[46] 1120 1100  832  764  821  768  845  864  862  698  845  744  796 1040
759
[61]  781  865  845  944  984  897  822 1010  771  676  649  846  812  742
801
[76] 1040  860  874  848  890  744  749  838 1050  918  986  797  923  975
815
[91] 1020  906  901 1170  912  746  919  718  714  740
```

Note here that we assign the result of `scan("nile.txt")` to a variable called `nile`, and we assume that `nile.txt` is in our working directory (type `getwd()` to see your working directory). If `nile.txt` is not in your working directory, section “Installing and Interacting with R” describes how you can change it. Alternatively, you can replace `scan("/path/to/nile.txt")`, where `/path/to/` is the path to the directory that `nile.txt` leaves, or, if you are using the terminal, navigating into the directory where `nile.txt` is and starting R from there. In what follows, we assume that all the data files we import data from are in the working directory.

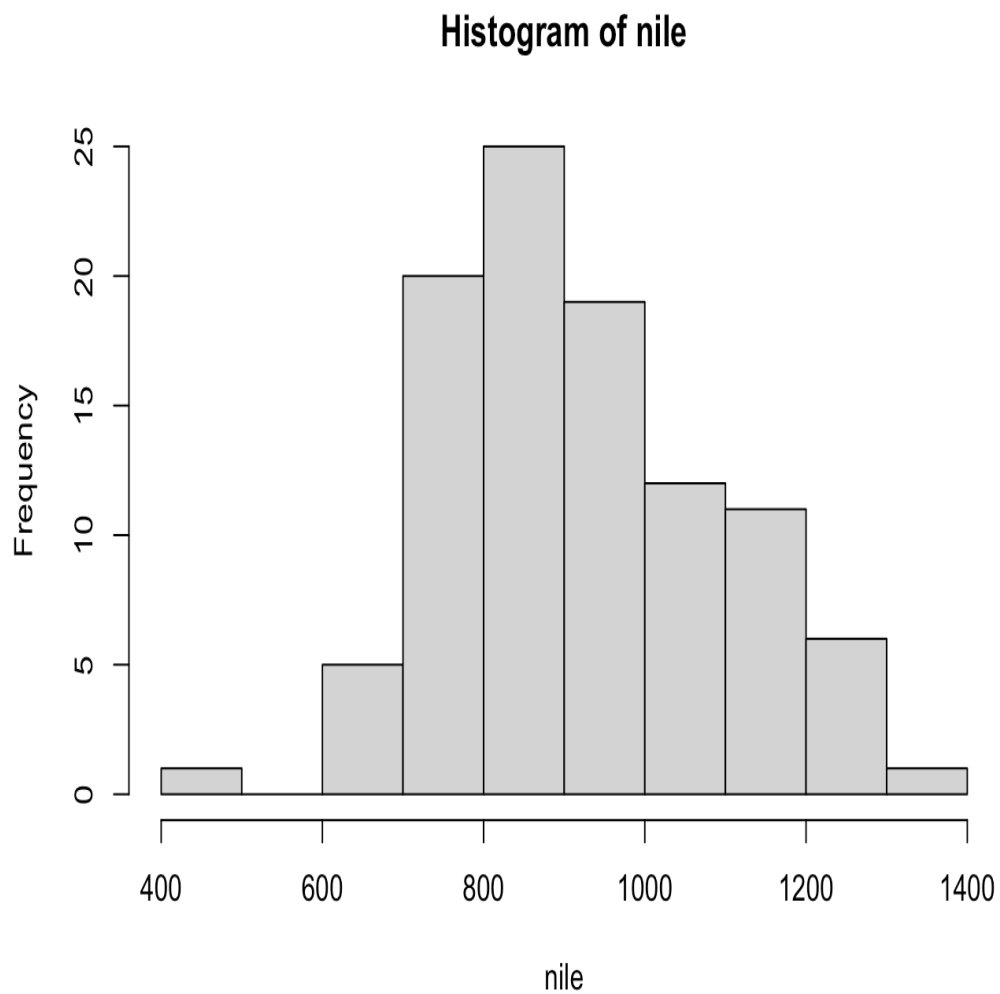
Each call to `scan()` can read a particular type of data. For example, the above line of code read the contents of `nile.txt` as double ([double precision numbers](#)).

```
typeof(nile)
[1] "double"
```

The `typeof()` function can be used to determine the R internal type or storage mode of any object (here of `nile`).

So, a histogram of the annual flow of the river Nile between 1871 and 1970 is

```
hist(nile)
```



If we want to read the contents of `nile.txt` as characters we do

```
nile_char <- scan("nile.txt", what = character())
nile_char
 [1] "1120" "1160" "963"  "1210" "1160" "1160" "813"  "1230" "1370" "1140"
[11] "995"  "935"  "1110" "994"  "1020" "960"  "1180" "799"  "958"  "1140"
[21] "1100" "1210" "1150" "1250" "1260" "1220" "1030" "1100" "774"  "840"
[31] "874"  "694"  "940"  "833"  "701"  "916"  "692"  "1020" "1050" "969"
[41] "831"  "726"  "456"  "824"  "702"  "1120" "1100" "832"  "764"  "821"
[51] "768"  "845"  "864"  "862"  "698"  "845"  "744"  "796"  "1040" "759"
[61] "781"  "865"  "845"  "944"  "984"  "897"  "822"  "1010" "771"  "676"
[71] "649"  "846"  "812"  "742"  "801"  "1040" "860"  "874"  "848"  "890"
[81] "744"  "749"  "838"  "1050" "918"  "986"  "797"  "923"  "975"  "815"
[91] "1020" "906"  "901"  "1170" "912"  "746"  "919"  "718"  "714"  "740"
typeof(nile_char)
[1] "character"
```

By default, `scan()` expects to read “white-space” delimited files. For reading files with different delimiters, the `sep` argument can be used. For example, the contents of `sunset-salvo-sem` are as follows

```
The;combination;of;some;data;and;an;aching;desire;for;an;answer;does;not;en
sure;that;a;reasonable;answer;can;be;extracted;from;a;given;body;of;data
```

Hence, the words are delimited by semi-colon (;), and we need to tell that to `scan()` in order to extract them:

```
scan("sunset-salvo-sem", what = character(), sep = ";")
 [1] "The"          "combination" "of"          "some"        "data"
 [6] "and"          "an"          "aching"      "desire"      "for"
[11] "an"          "answer"      "does"        "not"         "ensure"
[16] "that"        "a"          "reasonable" "answer"      "can"
[21] "be"          "extracted"   "from"        "a"           "given"
[26] "body"        "of"          "data"
```

`scan()` is one of the most basic utilities for reading data in R. It may seem simple at first but it offers a wide range of options, including skipping lines, reading only the first few lines of a file and reading different data types in tabular data; for example, we can read the data in `running_dat.tsv` by skipping the first line with the names, specifying that the file is tab-delimited (`sep = "\t"`), and that the first column is a character and the others are numbers (note that the type of data is specified here by simply supplying a list of a character and a number to `what`).

```
scan("running_dat.tsv", what = list("", 1.0, 1.0, 1.0, 1.0, 1.0, 1.0), sep
= "\t", skip = 1)
```

```
[[1]]
 [1] "2013-06-01 18:40:29" "2013-06-01 18:40:30" "2013-06-01 18:40:31"
 [4] "2013-06-01 18:40:32" "2013-06-01 18:40:33" "2013-06-01 18:40:34"
 [7] "2013-06-01 18:40:35" "2013-06-01 18:40:36" "2013-06-01 18:40:37"
[10] "2013-06-01 18:40:38" "2013-06-01 18:40:39"

[[2]]
 [1] 50.81381 50.81383 50.81385 50.81387 50.81388 50.81389 50.81391
50.81393
 [9] 50.81395 50.81395 50.81396

[[3]]
 [1] -1.712606 -1.712649 -1.712700 -1.712734 -1.712777 -1.712826 -1.712862
 [8] -1.712911 -1.712963 -1.712994 -1.713053

[[4]]
 [1] 80.20001 80.00000 79.79999 79.79999 79.59998 79.59998 79.40002
79.40002
 [9] 79.20001 79.20001 79.00000

[[5]]
 [1] 1805.94 1810.00 1814.55 1817.53 1820.50 1824.15 1827.38 1832.03
1836.18
[10] 1838.18 1842.39

[[6]]
 [1] 133 133 133 133 133 133 133 133 133 133 133 133

[[7]]
```



```
[1] 4.060059 4.550049 2.979981 2.969971 3.650024 3.229980 4.650024
4.149902
[9] 2.000000 4.210083 5.189941
```

Export

A convenient way to write lines to a file is the function `cat()`. `cat()` is used to output objects either in the standard output connection (typically directly in the R prompt), or to a file. For example,

```
cat(nile, file = "nile1.txt", sep = ",")
```

will create a file called `nile1.txt` in the working directory, where the values stored in `nile` are separated by `,`. See `?cat` for more information. Other useful, low-level functions to export lines whose help pages you may want to check out are `write` and `writeLines`.

Delimiter-Separated Values

Import

The R function `read.table()` is the principal means for importing tables from text files containing delimiter-separated values (like CSV files, for example). It uses `scan()` internally but it packs it in a convenient interface. For example, we can import the data in `running_dat.tsv` by doing

```
running <- read.table("running_dat.tsv", header = TRUE, sep = "\t")
running
```

| | | timestamp | latitude | longitude | altitude | distance | heart_rate |
|-------|------------|-----------|----------|-----------|----------|----------|------------|
| speed | | | | | | | |
| 1 | 2013-06-01 | 18:40:29 | 50.81381 | -1.712606 | 80.20001 | 1805.94 | 133 |
| | | | | | | | |
| 2 | 2013-06-01 | 18:40:30 | 50.81383 | -1.712649 | 80.00000 | 1810.00 | 133 |
| | | | | | | | |
| 3 | 2013-06-01 | 18:40:31 | 50.81385 | -1.712700 | 79.79999 | 1814.55 | 133 |
| | | | | | | | |
| 4 | 2013-06-01 | 18:40:32 | 50.81387 | -1.712734 | 79.79999 | 1817.53 | 133 |
| | | | | | | | |
| 5 | 2013-06-01 | 18:40:33 | 50.81388 | -1.712777 | 79.59998 | 1820.50 | 133 |
| | | | | | | | |
| 6 | 2013-06-01 | 18:40:34 | 50.81389 | -1.712826 | 79.59998 | 1824.15 | 133 |
| | | | | | | | |
| 7 | 2013-06-01 | 18:40:35 | 50.81391 | -1.712862 | 79.40002 | 1827.38 | 133 |
| | | | | | | | |
| 8 | 2013-06-01 | 18:40:36 | 50.81393 | -1.712911 | 79.40002 | 1832.03 | 133 |
| | | | | | | | |
| 9 | 2013-06-01 | 18:40:37 | 50.81395 | -1.712963 | 79.20001 | 1836.18 | 133 |
| | | | | | | | |
| 10 | 2013-06-01 | 18:40:38 | 50.81395 | -1.712994 | 79.20001 | 1838.18 | 133 |
| | | | | | | | |
| 11 | 2013-06-01 | 18:40:39 | 50.81396 | -1.713053 | 79.00000 | 1842.39 | 133 |
| | | | | | | | |

First, note that we did not need to specify the type of the data as we did with `scan`; indeed, `read.table()` managed to figure out the type correctly, which is clear if we use the `str()` function to get a concise description of the internal structure of the object `running`

```
str(running)
```

The ability to automatically figure out the variable types comes in extremely handy when it comes to large structured data sets or data sets that we are not otherwise familiar with.

Secondly, we tell `read.table()` that the first row of `running_dat.tsv` has the variable names, by setting the argument `header` to `TRUE`, and that `running_dat.tsv` is tab-separated.

We can do the same with comma-separated files. For example,

```
read.table("running_dat.csv", header = TRUE, sep = ",")
      timestamp latitude longitude altitude distance heart_rate
speed
1  2013-06-01 18:40:29 50.81381 -1.712606 80.20001  1805.94      133
4.060059
2  2013-06-01 18:40:30 50.81383 -1.712649 80.00000  1810.00      133
4.550049
3  2013-06-01 18:40:31 50.81385 -1.712700 79.79999  1814.55      133
2.979981
4  2013-06-01 18:40:32 50.81387 -1.712734 79.79999  1817.53      133
2.969971
5  2013-06-01 18:40:33 50.81388 -1.712777 79.59998  1820.50      133
3.650024
6  2013-06-01 18:40:34 50.81389 -1.712826 79.59998  1824.15      133
3.229980
7  2013-06-01 18:40:35 50.81391 -1.712862 79.40002  1827.38      133
4.650024
8  2013-06-01 18:40:36 50.81393 -1.712911 79.40002  1832.03      133
4.149902
9  2013-06-01 18:40:37 50.81395 -1.712963 79.20001  1836.18      133
2.000000
10 2013-06-01 18:40:38 50.81395 -1.712994 79.20001  1838.18      133
4.210083
11 2013-06-01 18:40:39 50.81396 -1.713053 79.00000  1842.39      133
5.189941
```

or simply

```
read.csv("running_dat.csv")
      timestamp latitude longitude altitude distance heart_rate
speed
1  2013-06-01 18:40:29 50.81381 -1.712606 80.20001  1805.94      133
4.060059
2  2013-06-01 18:40:30 50.81383 -1.712649 80.00000  1810.00      133
4.550049
3  2013-06-01 18:40:31 50.81385 -1.712700 79.79999  1814.55      133
2.979981
4  2013-06-01 18:40:32 50.81387 -1.712734 79.79999  1817.53      133
2.969971
5  2013-06-01 18:40:33 50.81388 -1.712777 79.59998  1820.50      133
3.650024
6  2013-06-01 18:40:34 50.81389 -1.712826 79.59998  1824.15      133
3.229980
```

```

7  2013-06-01 18:40:35 50.81391 -1.712862 79.40002 1827.38      133
4.650024
8  2013-06-01 18:40:36 50.81393 -1.712911 79.40002 1832.03      133
4.149902
9  2013-06-01 18:40:37 50.81395 -1.712963 79.20001 1836.18      133
2.000000
10 2013-06-01 18:40:38 50.81395 -1.712994 79.20001 1838.18      133
4.210083
11 2013-06-01 18:40:39 50.81396 -1.713053 79.00000 1842.39      133
5.189941

```

In fact, `read.csv()` (for comma-separated values using `.` for decimal points) and `read.csv2()` (for semi-colon-separated values using `,` for decimal points) are convenience functions that are the same as `read.table()` apart from having different default arguments (e.g. `write.csv()` has `sep = ","`, `dec = "."` and `header = TRUE` by default; see `?read.table` for details and type `read.csv()` in R command to see that it simply calls `read.table()`). The same is true for `read.delim()` (for tab-delimited files using `.` for decimal points) and `read.delim2()` (for tab-delimited files using `,` for decimal points). For example, treating `sunset-salvo-sem` as semi-colon-delimited tabular data gives

```

read.csv2("sunset-salvo-sem", header = FALSE)
      V1      V2 V3   V4   V5  V6 V7      V8      V9 V10 V11      V12  V13 V14
1 The combination of some data and an aching desire for an answer does not
      V15  V16 V17      V18   V19 V20 V21      V22  V23 V24      V25  V26
V27
1 ensure that a reasonable answer can be extracted from a given body
of
      V28
1 data

```

Note here that we had to set `header = FALSE` in order to let `read.csv2()` (and implicitly `read.table()`) know that `sunset-salvo-sem` does not have names and the imported data get the default column names `V1` to `V28`. Otherwise,

```

read.csv2("sunset-salvo-sem")
[1] The combination of some data and
[7] an aching desire for an answer
[13] does not ensure that a reasonable
[19] answer.1 can be extracted from a.1
[25] given body of.1 data.1
<0 rows> (or 0-length row.names)

```

which imports the words in `sunset-salvo-sem` as the column names and imports no (0 rows of) observations! Now, try to import the data in `running_dat.csv` with `header = FALSE` to see what happens.

[readr](#) is a noteworthy R package when it comes to importing tabular data from files, offering fast reading of tabular data, the option to provide column specifications, and many other features. See the [readr's vignettes](#) for an introduction.

Export

We also can write data in files with delimiter-separated values. For example, if for some reason we wanted to write the data in `running` in a file with `&&`-delimited values with decimal characters represented by `*` (both quite atypical choices!), we do

```
write.table(running, file = "running.dat", sep = "&&", dec = "*")
```

We can quickly inspect the lines in the file that was generated by doing

```
readLines("running.dat")
[1]
 "\"timestamp\"&&\"latitude\"&&\"longitude\"&&\"altitude\"&&\"distance\"&&\"heart_rate\"&&\"speed\""
[2] "\"1\"&&\"2013-06-01 18:40:29\"&&50*813805&&-1*7126063&&80*2000122&&1805*9399512&&133&&4*060058600000005"
[3] "\"2\"&&\"2013-06-01 18:40:30\"&&50*8138298&&-1*7126487&&80&&1810*0000098&&133&&4*550048800000001"
[4] "\"3\"&&\"2013-06-01 18:40:31\"&&50*8138543&&-1*7127005&&79*7999878&&1814*5500586&&133&&2*979980500000001"
[5] "\"4\"&&\"2013-06-01 18:40:32\"&&50*8138709&&-1*7127338&&79*7999878&&1817*5300391&&133&&2*969970699999998"
[6] "\"5\"&&\"2013-06-01 18:40:33\"&&50*8138757&&-1*7127769&&79*5999756&&1820*5000098&&133&&3*650024399999989"
[7] "\"6\"&&\"2013-06-01 18:40:34\"&&50*8138862&&-1*712826&&79*5999756&&1824*1500342&&133&&3*229980400000016"
[8] "\"7\"&&\"2013-06-01 18:40:35\"&&50*8139051&&-1*7128616&&79*4000244&&1827*3800146&&133&&4*650024499999997"
[9] "\"8\"&&\"2013-06-01 18:40:36\"&&50*8139326&&-1*7129115&&79*4000244&&1832*0300391&&133&&4*149902299999989"
[10] "\"9\"&&\"2013-06-01 18:40:37\"&&50*8139517&&-1*7129626&&79*2000122&&1836*1799414&&133&&2"
[11] "\"10\"&&\"2013-06-01 18:40:38\"&&50*8139537&&-1*7129945&&79*2000122&&1838*1799414&&133&&4*210082999999994"
[12] "\"11\"&&\"2013-06-01 18:40:39\"&&50*8139622&&-1*7130533&&79&&1842*3900244&&133&&5*189941400000018"
```

See `?write.table` for more options. There are also the convenience functions `write.csv()` and `write.csv2()` that simply have different default arguments to `write.table()`.

XML

Import

There are several ways to import XML data into R, with the most commonly used being the [XML](#) and [xml2](#) R packages.

From the two, **XML** is perhaps the most feature-rich providing options for the very fine control of the process of importing and exporting XML files (see the [pages of the Omega project](#) for tutorials and details), but **xml2** is particularly straightforward to use, because it takes care of several details relating to XML files automatically (see the **xml2** page for details).

After installing **xml2** (type `install.packages("xml2")` to do this) we do

```
library("xml2")
stats_people <- read_xml("statisticians.xml")
stats_people
{xml_document}
<records>
[1] <statistician>\n  <name>Ronald</name>\n  <middle>Aylmer</middle>\n
<surn ...
[2] <statistician>\n  <name>William</name>\n  <middle>Sealy</middle>\n
<surn ...
[3] <statistician>\n  <name>David</name>\n  <middle>Roxbee</middle>\n
<surna ...
[4] <statistician>\n  <name>Thomas</name>\n  <middle/>\n
<surname>Bayes</sur ...
[5] <statistician>\n  <name>Karl</name>\n  <middle/>\n
<surname>Pearson</sur ...
[6] <statistician>\n  <name>John</name>\n  <middle>Wilder</middle>\n
<surnam ...
```

Printing the object `stats_people` we see the first tag `<records>` and a printout of its 6 `<statistician>` children. **xml2** provides functions to *interrogate* `stats_people`.

For example, `xml_name()` and `xml_children()` can be used to extract the tag name of `stats_people`, and its children.

```
xml_name(stats_people)
[1] "records"
xml_children(stats_people)
{xml_nodeset (6)}
[1] <statistician>\n  <name>Ronald</name>\n  <middle>Aylmer</middle>\n
<surn ...
[2] <statistician>\n  <name>William</name>\n  <middle>Sealy</middle>\n
<surn ...
[3] <statistician>\n  <name>David</name>\n  <middle>Roxbee</middle>\n
<surna ...
[4] <statistician>\n  <name>Thomas</name>\n  <middle/>\n
<surname>Bayes</sur ...
[5] <statistician>\n  <name>Karl</name>\n  <middle/>\n
<surname>Pearson</sur ...
[6] <statistician>\n  <name>John</name>\n  <middle>Wilder</middle>\n
<surnam ...
```

If we want to extract the surnames of the famous statistics we first have to find all children with tag `<surname>`. This can be done using XPath (XML Path language) syntax, which allows to select nodes from an XML document (see the [w3schools.com tutorial](http://w3schools.com/tutorial) for learning XPath). We do

```
surname_nodes <- xml_find_all(stats_people, ".//surname")
surname_nodes
{xml_nodeset (6)}
[1] <surname>Fisher</surname>
[2] <surname>Gosset</surname>
[3] <surname>Cox</surname>
[4] <surname>Bayes</surname>
[5] <surname>Pearson</surname>
[6] <surname>Tukey</surname>
```

where “`./surname`” is the XPath expression to find the “surname” tag. Then we can extract the surnames as

```
xml_text(surname_nodes)
[1] "Fisher" "Gosset" "Cox"      "Bayes"   "Pearson" "Tukey"
```

The data in `statisticians.xml` can be represented in tabular form, where we have a statistician in each row and the various attributes for each statistician on the columns. The **XML R** package provides a handy function called `xmlToDataFrame()` that allows us to extract data from XML files in tabular form. For example,

```
library("XML")
stats_people_df <- xmlToDataFrame("statisticians.xml")
stats_people_df
  name middle surname      dob
1 Ronald Aylmer  Fisher 17/02/1890
2 William Sealy  Gosset 13/08/1876
3 David Roxbee   Cox   15/07/1924
4 Thomas                Bayes 07/04/1761
5 Karl              Pearson 27/03/1857
6 John Wilder     Tukey  16/06/1915

                                wiki
1      https://en.wikipedia.org/wiki/Ronald_Fisher
2      https://en.wikipedia.org/wiki/William_Sealy_Gosset
3 https://en.wikipedia.org/wiki/David_Cox_(statistician)
4      https://en.wikipedia.org/wiki/Thomas_Bayes
5      https://en.wikipedia.org/wiki/Karl_Pearson
6      https://en.wikipedia.org/wiki/John_Tukey
```

We can now export `stats_people_df` using `write.table()`, `write.csv()` and `write.csv2()`.

Bear in mind though, that in most cases, it is not possible to represent data in XML files in tabular form. See `?xmlToDataFrame` for which data types are supported and about what `xmlToDataFrame()` can do.

Export

Objects like `stats_people`, i.e. objects that represent XML documents or nodes (in R terminology, objects of class `xml_document` or `xml_node`) can be written on disk using the `write_xml()` functions from **xml2**. This is useful, for example, if we have an R script whose results need to be exported as XML to be later read by another utility in a data-analytic pipeline.

For example, if we want to remove Thomas Bayes’ record from `stats_people` and write the result on disk we do

```
bayes_record <- xml_children(stats_people)[[4]]
xml_remove(bayes_record)
stats_people
{xml_document}
<records>
[1] <statistician>\n  <name>Ronald</name>\n  <middle>Aylmer</middle>\n
<surn ...
```

```

[2] <statistician>\n  <name>William</name>\n  <middle>Sealy</middle>\n
<surname>...
[3] <statistician>\n  <name>David</name>\n  <middle>Roxbee</middle>\n
<surname>...
[4] <statistician>\n  <name>Karl</name>\n  <middle/>\n
<surname>Pearson</surname>...
[5] <statistician>\n  <name>John</name>\n  <middle>Wilder</middle>\n
<surname>...
write_xml(stats_people, "statisticians_no_bayes.xml")
NULL
readLines("statisticians_no_bayes.xml")
[1] "<?xml version='1.0' encoding='UTF-8'?"
[2] "<!-- A list of famous statisticians -->"
[3] "<records>"
[4] "  <statistician>"
[5] "    <name>Ronald</name>"
[6] "    <middle>Aylmer</middle>"
[7] "    <surname>Fisher</surname>"
[8] "    <dob>17/02/1890</dob>"
[9] "    <wiki>https://en.wikipedia.org/wiki/Ronald_Fisher</wiki>"
[10] "  </statistician>"
[11] "  <statistician>"
[12] "    <name>William</name>"
[13] "    <middle>Sealy</middle>"
[14] "    <surname>Gosset</surname>"
[15] "    <dob>13/08/1876</dob>"
[16] "    <wiki>https://en.wikipedia.org/wiki/William_Sealy_Gosset</wiki>"
[17] "  </statistician>"
[18] "  <statistician>"
[19] "    <name>David</name>"
[20] "    <middle>Roxbee</middle>"
[21] "    <surname>Cox</surname>"
[22] "    <dob>15/07/1924</dob>"
[23] "    <wiki>https://en.wikipedia.org/wiki/David_Cox_(statistician)</wiki>"
[24] "  </statistician>"
[25] "  <statistician>"
[26] "    <name>Karl</name>"
[27] "    <middle/>"
[28] "    <surname>Pearson</surname>"
[29] "    <dob>27/03/1857</dob>"
[30] "    <wiki>https://en.wikipedia.org/wiki/Karl_Pearson</wiki>"
[31] "  </statistician>"
[32] "  <statistician>"
[33] "    <name>John</name>"
[34] "    <middle>Wilder</middle>"
[35] "    <surname>Tukey</surname>"
[36] "    <dob>16/06/1915</dob>"
[37] "    <wiki>https://en.wikipedia.org/wiki/John_Tukey</wiki>"
[38] "  </statistician>"
[39] "</records>"

```

We knew that Thomas Bayes' record was at the fourth node (this is what `[[4]]` above simply extracts the fourth record) by looking at the order the surnames appear when we extracted those earlier.

See, the [xml2 vignette on node modification](#).

JSON

Import

The `jsonlite` R package provides a JSON parser and a range of tools for working with JSON documents in R. [jsonlite's CRAN page](#) provides a range of vignettes that demonstrate its functionality, in particular, reading JSON documents from files, getting JSON documents from APIs, combining JSON documents, and more.

If we want to read `statisticians.json`, then we do

```
library("jsonlite")
stats_people <- fromJSON("statisticians.json")
stats_people
  name middle surname      dob
1 Ronald  Aylmer   Fisher 17/02/1890
2 William Sealy   Gosset 13/08/1876
3 David  Roxbee    Cox  15/07/1924
4 Thomas  <NA>     Bayes 07/04/1761
5 Karl    <NA>     Pearson 27/03/1857
6 John   Wilder    Tukey 16/06/1915

                                wiki
1 https://en.wikipedia.org/wiki/Ronald_Fisher
2 https://en.wikipedia.org/wiki/William_Sealy_Gosset
3 https://en.wikipedia.org/wiki/David_Cox_(statistician)
4 https://en.wikipedia.org/wiki/Thomas_Bayes
5 https://en.wikipedia.org/wiki/Karl_Pearson
6 https://en.wikipedia.org/wiki/John_Tukey
```

There is also a function `read_json()`, which will read the contents of the JSON document and bring into R as a nested *list* (we will see what this is in later sections), that is somewhat closer to the hierarchy present in a JSON document.

We can also take existing R objects and convert them to JSON

```
running_json <- toJSON(running)
running_json
[{"timestamp":"2013-06-01 18:40:29","latitude":50.8138,"longitude":-
1.7126,"altitude":80.2,"distance":1805.94,"heart_rate":133,"speed":4.0601},
{"timestamp":"2013-06-01 18:40:30","latitude":50.8138,"longitude":-
1.7126,"altitude":80,"distance":1810,"heart_rate":133,"speed":4.55},{"times
tamp":"2013-06-01 18:40:31","latitude":50.8139,"longitude":-
1.7127,"altitude":79.8,"distance":1814.5501,"heart_rate":133,"speed":2.98},
{"timestamp":"2013-06-01 18:40:32","latitude":50.8139,"longitude":-
1.7127,"altitude":79.8,"distance":1817.53,"heart_rate":133,"speed":2.97},{"
timestamp":"2013-06-01 18:40:33","latitude":50.8139,"longitude":-
1.7128,"altitude":79.6,"distance":1820.5,"heart_rate":133,"speed":3.65},{"t
imestamp":"2013-06-01 18:40:34","latitude":50.8139,"longitude":-
1.7128,"altitude":79.6,"distance":1824.15,"heart_rate":133,"speed":3.23},{"
timestamp":"2013-06-01 18:40:35","latitude":50.8139,"longitude":-
1.7129,"altitude":79.4,"distance":1827.38,"heart_rate":133,"speed":4.65},{"
timestamp":"2013-06-01 18:40:36","latitude":50.8139,"longitude":-
1.7129,"altitude":79.4,"distance":1832.03,"heart_rate":133,"speed":4.1499},
{"timestamp":"2013-06-01 18:40:37","latitude":50.814,"longitude":-
1.713,"altitude":79.2,"distance":1836.1799,"heart_rate":133,"speed":2},{"ti
```



```
mestamp":"2013-06-01 18:40:38","latitude":50.814,"longitude":-
1.713,"altitude":79.2,"distance":1838.1799,"heart_rate":133,"speed":4.2101}
,{"timestamp":"2013-06-01 18:40:39","latitude":50.814,"longitude":-
1.7131,"altitude":79,"distance":1842.39,"heart_rate":133,"speed":5.1899}]
```

This long string is the *minified* version of the JSON representation of the data in running. We can *prettify* it (or indent the document) to more clearly see what `toJSON()` does by doing

```
prettify(running_json)
[
  {
    "timestamp": "2013-06-01 18:40:29",
    "latitude": 50.8138,
    "longitude": -1.7126,
    "altitude": 80.2,
    "distance": 1805.94,
    "heart_rate": 133,
    "speed": 4.0601
  },
  {
    "timestamp": "2013-06-01 18:40:30",
    "latitude": 50.8138,
    "longitude": -1.7126,
    "altitude": 80,
    "distance": 1810,
    "heart_rate": 133,
    "speed": 4.55
  },
  {
    "timestamp": "2013-06-01 18:40:31",
    "latitude": 50.8139,
    "longitude": -1.7127,
    "altitude": 79.8,
    "distance": 1814.5501,
    "heart_rate": 133,
    "speed": 2.98
  },
  {
    "timestamp": "2013-06-01 18:40:32",
    "latitude": 50.8139,
    "longitude": -1.7127,
    "altitude": 79.8,
    "distance": 1817.53,
    "heart_rate": 133,
    "speed": 2.97
  },
  {
    "timestamp": "2013-06-01 18:40:33",
    "latitude": 50.8139,
    "longitude": -1.7128,
    "altitude": 79.6,
    "distance": 1820.5,
    "heart_rate": 133,
    "speed": 3.65
  },
  {
    "timestamp": "2013-06-01 18:40:34",
    "latitude": 50.8139,
    "longitude": -1.7128,
    "altitude": 79.6,
```

```

        "distance": 1824.15,
        "heart_rate": 133,
        "speed": 3.23
    },
    {
        "timestamp": "2013-06-01 18:40:35",
        "latitude": 50.8139,
        "longitude": -1.7129,
        "altitude": 79.4,
        "distance": 1827.38,
        "heart_rate": 133,
        "speed": 4.65
    },
    {
        "timestamp": "2013-06-01 18:40:36",
        "latitude": 50.8139,
        "longitude": -1.7129,
        "altitude": 79.4,
        "distance": 1832.03,
        "heart_rate": 133,
        "speed": 4.1499
    },
    {
        "timestamp": "2013-06-01 18:40:37",
        "latitude": 50.814,
        "longitude": -1.713,
        "altitude": 79.2,
        "distance": 1836.1799,
        "heart_rate": 133,
        "speed": 2
    },
    {
        "timestamp": "2013-06-01 18:40:38",
        "latitude": 50.814,
        "longitude": -1.713,
        "altitude": 79.2,
        "distance": 1838.1799,
        "heart_rate": 133,
        "speed": 4.2101
    },
    {
        "timestamp": "2013-06-01 18:40:39",
        "latitude": 50.814,
        "longitude": -1.7131,
        "altitude": 79,
        "distance": 1842.39,
        "heart_rate": 133,
        "speed": 5.1899
    }
]

```

Export

We can write tabular objects like `running` as JSON to disk, using `jsonlite`'s `write_json()` function. For example,

```
write_json(running, "running.json", pretty = TRUE)
```

will write the tabular data in `running` in a JSON document in `running.json`. `pretty = TRUE` tells `write_json()` to indent the objects in the JSON document before writing.

Spreadsheets

There are various packages to read spreadsheet formats. For example, the [readxl](#) R package provides excellent out-of-the-box support for reading Microsoft Excel files (with extensions `.xls` for older versions, and `.xlsx` for newer versions of Microsoft Excel). See, [readxl's page](#) for demonstrations. Also, the [writexl](#) allows to write R objects with rectangular data into `.xlsx` files. See, [writexl's page](#) for demonstrations.

writexl is part of a collection of packages by the [rOpenSci](#) initiative, which is worth taking a look at.

The [readODS](#) R package provides utilities to read and write Open Document Spreadsheets (with extension `.ods`) files from R. See the package's vignettes for more information.

Other File Formats

There are additional R and Python packages that provide great support for reading data from a wide variety of file formats for data exchange. See, for example, the file types that the [foreign](#) R package supports, the R's [Data Import/Export documentation](#), and the files that can be read by the [pandas](#) Python package.

Useful Links and Resources

- [R data import/export documentation](#)
- [pandas IO tools](#)
- [XPath cheat sheet](#) by devhints.io and [w3schools.com XPath tutorial](#).
- [readr](#) R package for importing tabular data from files.
- [XML](#) provides options for the very fine control of the process of importing and exporting XML files; see, also the [pages of the Omega project](#) for tutorials and details.
- [xml2](#) pages
- [jsonlite](#) CRAN page and the vignettes there.
- [readxl](#) and [writexl](#) R packages for reading and writing spreadsheets from Microsoft Excel files.
- [readODS](#) R package provides utilities to read and write Open Document Spreadsheets.

Data Types in R

Data Types in R

The most common data types in R are

- Logical
- Character
- Integer
- Numeric
- Factor.

Objects with these data types can be organized into data structures, such as

- Atomic vector
- Matrix
- Array
- List
- Data frame.

Logical

One of the most useful data types in R (and programming in general) is logical. An object of type *logical* in R has two possible values, `TRUE` and `FALSE`. For example, if we do

```
a <- TRUE
b <- FALSE
```

we have created two objects of type logical with names `a` and `b` and values `TRUE` and `FALSE`, respectively.

The class of object `a` is

```
class(a)
[1] "logical"
```

R provides the key logical operators to carry out [Boolean algebra](#). In particular,

| Operator | Description |
|--------------------|-------------------|
| <code>&</code> | AND (conjunction) |
| <code> </code> | OR (disjunction) |
| <code>!</code> | NOT (negation) |

So, we can easily reproduce the results of the basic Boolean algebra operations in R

```
a & a
[1] TRUE
```

```

b & b
[1] FALSE
a & b
[1] FALSE
a | a
[1] TRUE
b | b
[1] FALSE
a | b
[1] TRUE
!a
[1] FALSE
!b
[1] TRUE

```

Character

A character object represents text (or string) values in R. We use double or single quotation marks to represent text. For example,

```

str1 <- "Data"
str2 <- 'Structures'
class(str1)
[1] "character"

```

We can also combine single and double quotes if we want to make quotes part of the string. For example,

```

str3 <- "in 'R'"
str3
[1] "in 'R'"

```

The `paste()` function can be used to concatenate the strings

```

str <- paste(str1, str2, str3)
print(str)
[1] "Data Structures in 'R'"

```

We can also use the `cat()` function directly. For example

```

cat(str1, str2, str3, "\n")
Data Structures in 'R'

```

Adding `"\n"` tells R to break a line after concatenating the strings. See `?Quotes` for a descriptions of the various “character constants” (also known as special characters) in R.

Of course, forgetting quotation marks can result in errors. For example,

```

str5 <- in 'R'
Error: <text>:1:9: unexpected 'in'
1: str5 <- in
              ^

```

and

```
str6 <- Structures
Error in eval(expr, envir, enclos): object 'Structures' not found
```

In the last code chunk, R tries, unsuccessfully, to find an object called `Structures` and assign its value to `str6`. Note that this could be potentially dangerous, and definitely not what we had in mind (`str6 <- "Structures"`), if an object named `Structures` actually existed!

Numeric and Integer

In R, real numbers are represented as `numeric`. For example,

```
x <- 2.5
class(x)
[1] "numeric"
z <- 2
class(z)
[1] "numeric"
```

Even though the number 2 is an integer, R assigns `z` above as `numeric` by default. If we want an integer value we have to explicitly write

```
z <- as.integer(5)
z
[1] 5
class(z)
[1] "integer"
```

Another popular way to declare an integer in R is appending an `L` to the number

```
z <- 5L
z
[1] 5
class(z)
[1] "integer"
```

As you may expect, R can be used to do basic arithmetic. Below are the basic arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^` or `**`
- Modulo: `%%`

Below are some examples

```
# Addition
3 + 2
[1] 5
# Subtraction
5 - 2
[1] 3
```

```

# Multiplication
5 * 3
[1] 15
5 * -2
[1] -10
5 * 2.5
[1] 12.5
# Division
5 / 3
[1] 1.666667
5 / -2
[1] -2.5
5 / 2.5
[1] 2
# Exponentiation
2**2
[1] 4
5^2.5
[1] 55.9017
5^-2
[1] 0.04
# Modulus
10 %% 3
[1] 1
9 %% 3
[1] 0

```

We can also compare numbers with the result of the comparison being a logical. For example

```

2 > 4
[1] FALSE
exp(pi) < pi^exp(1)
[1] FALSE
log(4) < 4
[1] TRUE
a <- 2.321
a == 2.321
[1] TRUE

```

Coerce and Test

We can coerce objects from one type to another using the `as.*()` functions. Try to understand what the following functions do, and why the result is as is below.

```

as.character(FALSE)
[1] "FALSE"
as.character(1.123)
[1] "1.123"
as.integer(1.123)
[1] 1
as.logical(4)
[1] TRUE
as.logical(0)
[1] FALSE
as.numeric(1L)
[1] 1
as.numeric("abc")
Warning: NAs introduced by coercion

```

```

[1] NA
as.list(mtcars)
$mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8
19.7
[31] 15.0 21.4

$cyl
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4

$displ
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6
275.8
[13] 275.8 275.8 472.0 460.0 440.0 78.7 75.7 71.1 120.1 318.0 304.0
350.0
[25] 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

$hp
 [1] 110 110 93 110 175 105 245 62 95 123 123 180 180 180 205 215 230
66 52
[20] 65 97 150 150 245 175 66 91 113 264 175 335 109

$drat
 [1] 3.90 3.90 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 3.07
2.93
[16] 3.00 3.23 4.08 4.93 4.22 3.70 2.76 3.15 3.73 3.08 4.08 4.43 3.77 4.22
3.62
[31] 3.54 4.11

$wt
 [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
4.070
[13] 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520 3.435
3.840
[25] 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780

$qsec
 [1] 16.46 17.02 18.61 19.44 17.02 20.22 15.84 20.00 22.90 18.30 18.90
17.40
[13] 17.60 18.00 17.98 17.82 17.42 19.47 18.52 19.90 20.01 16.87 17.30
15.41
[25] 17.05 18.90 16.70 16.90 14.50 15.50 14.60 18.60

$vs
 [1] 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1

$am
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1

$gear
 [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4

$carb
 [1] 4 4 1 1 2 1 4 2 2 4 4 3 3 3 4 4 4 1 2 1 1 2 2 4 2 1 2 2 4 6 8 2

```

We can also test the type of an object using `is.*()` functions. For example,

```
is.character(3)
```



```

[1] FALSE
is.character("hello")
[1] TRUE
is.logical(4)
[1] FALSE
is.logical(TRUE)
[1] TRUE
is.numeric(4)
[1] TRUE
is.numeric(FALSE)
[1] FALSE
is.complex(FALSE)
[1] FALSE
is.complex(43.3)
[1] FALSE
is.complex(43.3i + 2)
[1] TRUE

```

NA and NaN

NA (not available) is a logical constant in R of length 1 which contains a missing value indicator. For example, if you compute the mean of a character, or try to convert a character to numeric, you get NA

```

mean("a")
Warning in mean.default("a"): argument is not numeric or logical: returning
NA
[1] NA
as.numeric("b")
Warning: NAs introduced by coercion
[1] NA

```

NAs occur often when subsetting vectors, as we see in the next section.

Another special object is [NaN](#) (not available number), which is often confused to the NA. For example, the logarithm and the sqrt of $\sqrt{-1}$ are NaN, simply because they do not exist in the reals.

```

log(-1)
Warning in log(-1): NaNs produced
[1] NaN
sqrt(-1)
Warning in sqrt(-1): NaNs produced
[1] NaN

```

Note though that NaN is always NA, but NA is not always NaN, as the following tests show.

```

is.na(NA)
[1] TRUE
is.nan(NA)
[1] FALSE
is.na(NaN)
[1] TRUE
is.nan(NaN)
[1] TRUE

```

Also, note that almost every operation performed with `NA` or `NaN` will return `NA` or `NaN`.

```
NA + 1
[1] NA
NA + NaN
[1] NA
NaN * 3
[1] NaN
3^NaN
[1] NaN
```

but

```
length(NA)
[1] 1
length(NaN)
[1] 1
```

Useful Links and Resources

- [Chapter 14 in *R for data science*](#)
- [Chapter 5 in *Hands-on programming with R*](#)

Data Structures in R



Vectors

Vectors are one-dimensional arrays that can hold data. They are simple objects to store numeric data, character data and logical data. Bear in mind that a vector can contain only one single data type. You cannot have different data types inside the same vector.

In R, we can create a vector with the combine function `c()`. The vector elements are separated by a comma inside the parentheses. For example,

```
marks <- c(80, 75, 90, 99, 100)
names <- c("John", "Jane", "Richard", "Emma")
logicals <- c(FALSE, TRUE, TRUE, FALSE, FALSE, TRUE)
marks
[1] 80 75 90 99 100
names
[1] "John" "Jane" "Richard" "Emma"
logicals
[1] FALSE TRUE TRUE FALSE FALSE TRUE
class(marks)
[1] "numeric"
class(names)
[1] "character"
class(logicals)
[1] "logical"
```

We can even give names to the elements of our vectors. This is very useful to keep track of what each element represents.

To name elements of a vector, you can use the `names()` function. For example,

```
# Creating a simple character vector
Full_Name <- c("John", "Doe")
Full_Name
[1] "John" "Doe"
# Naming the vector
names(Full_Name) <- c("First Name", "Last Name")
Full_Name
First Name Last Name
```

```

      "John"      "Doe"
# Creating a simple numeric vector
marks <- c(80, 75, 90, 99, 100)
marks
[1] 80 75 90 99 100
# Naming the vector
names(marks) <- c("John", "Jane", "Richard", "Emma", "Tim")
marks
      John      Jane Richard      Emma      Tim
      80      75      90      99      100

```

Now that we know how to create vectors, we can perform calculations with them. These are performed element-wise. Here is an example:

```

# Creating two simple numeric vectors
x <- c(2, 3, 4)
y <- c(5, 6, 7)
# Addition
x + y
[1] 7 9 11
# Subtraction
x - y
[1] -3 -3 -3
# Multiplication
x * y
[1] 10 18 28
# Division
y / x
[1] 2.50 2.00 1.75
# Exponentiation
y ^ x
[1] 25 216 2401
# Modulo
y %% x
[1] 1 0 3

```

In addition, here are some useful functions that can be used with vectors:

- `length()`: Returns the length of the vector.
- `sum()`: Returns the sum of the elements of the vector.
- `min()`: Returns the lowest value found inside the vector.
- `max()`: Returns the highest value found inside the vector.
- `mean()`: Returns the mean of the elements of the vector.
- `median()`: Returns the median of the elements of the vector.
- `sd()`: Returns the standard deviation of the elements of the vector.
- `var()`: Returns the variance of the elements of the vector.
- `cov()`: Returns the covariance of two vectors.
- `cor()`: Returns the correlation of two vectors.
- `sort()`: Sorts a vector into ascending or descending order.

Here is a demonstration using the marks vector we previously created.

```

marks
      John      Jane Richard      Emma      Tim
      80      75      90      99      100
length(marks)

```

```

[1] 5
sum(marks)
[1] 444
min(marks)
[1] 75
max(marks)
[1] 100
mean(marks)
[1] 88.8
median(marks)
[1] 90
sd(marks)
[1] 11.16692
var(marks)
[1] 124.7
marks2 <- c(85, 50, 64, 95, 45)
marks2
[1] 85 50 64 95 45
cov(marks, marks2)
[1] 27.95
cor(marks, marks2)
[1] 0.1152433
sort(marks, decreasing = FALSE)
  Jane   John Richard   Emma   Tim
   75    80    90    99   100
sort(marks, decreasing = TRUE)
  Tim   Emma Richard   John   Jane
  100    99    90    80    75

```

We can even make element-wise comparisons between vectors using relational operators. Let us compare average daily weekday temperatures for 2 weeks, namely `temperature_week_1` and `temperature_week_2` respectively:

```

temperature_week_1 <- c(10,11,12,15,13)
temperature_week_2 <- c(10,9,13,15,16)
temperature_week_1
[1] 10 11 12 15 13
temperature_week_2
[1] 10 9 13 15 16
temperature_week_1 < temperature_week_2
[1] FALSE FALSE TRUE FALSE TRUE
temperature_week_1 <= temperature_week_2
[1] TRUE FALSE TRUE TRUE TRUE
temperature_week_1 > temperature_week_2
[1] FALSE TRUE FALSE FALSE FALSE
temperature_week_1 >= temperature_week_2
[1] TRUE TRUE FALSE TRUE FALSE
temperature_week_1 == temperature_week_2
[1] TRUE FALSE FALSE TRUE FALSE
temperature_week_1 != temperature_week_2
[1] FALSE TRUE TRUE FALSE TRUE

```

Now, suppose we want to select specific elements of a vector. R lets us do this using square brackets `[]`:

```

# Access first element
marks[1]
John
80

```

```

# Access second element
marks[2]
Jane
  75
# Access last element
marks[length(marks)]
Tim
100
# Access first two elements
marks[1:2]
John Jane
  80   75
# Access second to fourth element
marks[2:4]
  Jane Richard   Emma
    75     90     99
# Access second to fourth element (alternative method)
marks[c(2,3,4)]
  Jane Richard   Emma
    75     90     99
# Access first, third and fifth elements of marks vector
marks[c(1,3,5)]
  John Richard   Tim
    80     90    100
# Access elements of marks vector using names
marks[c("John", "Jane", "Tim")]
John Jane Tim
  80   75 100

```

Finally, you can also pass a logical vector inside square brackets. R will only return elements that correspond to TRUE. For week 1, suppose we only want the day(s) where the temperature was higher than week 2. Here is how to do it:

```

weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
# Naming temperature vectors
names(temperature_week_1) <- weekdays
names(temperature_week_2) <- weekdays
# Making comparisons
temperature_week_1 > temperature_week_2
  Monday  Tuesday Wednesday Thursday   Friday
  FALSE    TRUE    FALSE    FALSE    FALSE
# Selecting based on condition
temperature_week_1[temperature_week_1 > temperature_week_2]
Tuesday
  11

```

Factors

Data often fall into a limited number of categories. For instance, cars can be categorized according to their brand, such as Audi, Jaguar, Mercedes, BMW and so on. Categorical data in R are typically stored in “factors.” Factors are very important, especially when visualizing data and building models, so it is important to learn about them now.

We use the `factor()` function to create factors in R. For example,

```
colour_vector <- c("blue", "red", "green", "green", "blue", "green",
"yellow", "grey")
class(colour_vector)
[1] "character"
colour_vector_factor <- factor(colour_vector)
class(colour_vector_factor)
[1] "factor"
colour_vector_factor
[1] blue   red    green  green  blue   green  yellow grey
Levels: blue green grey red yellow
```

We first construct a character vector called `colour_vector` with five character values “red,” “blue,” “yellow,” “green” and “grey.” Using the `factor()` function, we convert it into a factor variable called `colour_vector_factor`. After printing `colour_vector_factor`, R conveniently displays the levels (or the categories) of our factor variable.

There are two types of categorical variables, namely “ordinal categorical variables” and “nominal categorical variables.” Nominal categorical variables have categories without an implied order, that is, it is not possible to say that “one category ranks higher or lower than the other.” An example is hair colour. On the other hand, ordinal categorical variables have categories with a natural ordering. For instance, suppose the categorical variable `customer_satisfaction` has the following categories; “Low,” “Medium” and “High.” Then, clearly, “Low” ranks below “Medium,” which in turn ranks below “High.”

Sometimes, the factor levels don’t have entirely meaningful names. For example, we might have “L,” “M” and “H” instead of “Low,” “Medium” and “High.” Fortunately, R lets us change the names of the levels (or categories) using the `levels()` function as follows:

```
customer_satisfaction <- factor(c("L", "M", "L", "L", "H"))
customer_satisfaction
[1] L M L L H
Levels: H L M
# Renaming factor levels
levels(customer_satisfaction) <- c("High", "Low", "Medium")
customer_satisfaction
[1] Low   Medium Low    Low    High
Levels: High Low Medium
```

Note that we did not specify the order of the levels. R automatically ranked the categories in alphabetical order. This is why “H” is before “L” and “L” is before “M.” When using the `levels()` function, we should be careful to respect this ordering, else R will not name the levels correctly.

To explicitly instruct R to order the factors, we should set `ordered = TRUE` when calling the `factor()` function. We can also specify the levels in the same command, Here are some examples:

```
speed <- c("fast", "slow", "slow", "fast", "medium")
speed_factor <- factor(speed, ordered = TRUE, levels = c("slow", "medium",
"fast"))
speed_factor
[1] fast    slow    slow    fast    medium
Levels: slow < medium < fast
speed_factor[1] > speed_factor[2]
[1] TRUE
speed_factor[4] > speed_factor[5]
[1] TRUE
speed_factor[3] > speed_factor[4]
[1] FALSE
summary(speed_factor)
  slow medium    fast
    2      1      2
```

Here, we create a factor variable called `speed_factor` with three levels, namely “slow,” “medium” and “fast” in that order. We then compare a few elements of the `speed_factor` variable to see whether they rank above or below each other. Finally, we use the `summary()` function to obtain a convenient summary of our factor.

Matrices

A matrix is similar to a vector in the sense that it holds elements of the same data type. However, a matrix is two-dimensional whereas a vector is only one-dimensional. This means that the elements of a matrix are arranged into a number of rows and columns. You cannot have different data types inside the same matrix.

We use the `matrix()` function to create a matrix in R.

Here are some examples:

```
# 3 x 4 matrix, byrow = TRUE
my_matrix <- matrix(1:12, byrow = TRUE, nrow = 3)
# 3 x 4 matrix, byrow = FALSE
my_matrix_1 <- matrix(1:12, byrow = FALSE, nrow = 3)
# 4 x 3 matrix, byrow = FALSE
my_matrix_2 <- matrix(1:12, byrow = FALSE, nrow = 4)
my_matrix
  [,1] [,2] [,3] [,4]
[1,]   1   2   3   4
[2,]   5   6   7   8
[3,]   9  10  11  12
my_matrix_1
  [,1] [,2] [,3] [,4]
[1,]   1   4   7  10
[2,]   2   5   8  11
[3,]   3   6   9  12
my_matrix_2
  [,1] [,2] [,3]
[1,]   1   5   9
```



```
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

You can also create matrices from a collection of vectors. Here, we will be using the temperature vectors from the previous subsection:

```
temperature_week_1
  Monday Tuesday Wednesday Thursday Friday
      10      11      12      15      13
temperature_week_2
  Monday Tuesday Wednesday Thursday Friday
      10      9      13      15      16
# Combine temperature_week_1 and temperature_week_2 into a single vector
temperature_combined <- c(temperature_week_1, temperature_week_2)
temperature_combined
  Monday Tuesday Wednesday Thursday Friday Monday Tuesday
Wednesday
      10      11      12      15      13      10      9
13
  Thursday Friday
      15      16
# Converting into a matrix
temperature_week_1_and_2 <- matrix(temperature_combined, byrow = FALSE,
nrow = 2)
temperature_week_1_and_2
      [,1] [,2] [,3] [,4] [,5]
[1,]   10   12   13    9   15
[2,]   11   15   10   13   16
# R does not throw any error, but does the above matrix make sense in our
context? What is wrong?
# Let us try setting byrow = TRUE
temperature_week_1_and_2 <- matrix(temperature_combined, byrow = TRUE, nrow
= 2)
temperature_week_1_and_2
      [,1] [,2] [,3] [,4] [,5]
[1,]   10   11   12   15   13
[2,]   10    9   13   15   16
# This matrix makes more sense as the data for each week is shown on a
separate row
```

We can also use the `rownames()` and `colnames()` functions to give names to the rows and the columns of a matrix. For example,

```
temperature_week_1_and_2
      [,1] [,2] [,3] [,4] [,5]
[1,]   10   11   12   15   13
[2,]   10    9   13   15   16
weeks <- c("Week 1", "Week 2")
weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
rownames(temperature_week_1_and_2) <- weeks
colnames(temperature_week_1_and_2) <- weekdays
temperature_week_1_and_2
      Monday Tuesday Wednesday Thursday Friday
Week 1     10      11      12      15      13
Week 2     10       9      13      15      16
```

Just like vectors, we can perform a lot of interesting calculations on matrices. In addition to the functions introduced in the previous subsection, here are four additional functions that are useful when working with matrices:

- `rowSums()`: Returns a vector where each element represents the total of the values of the corresponding row.
- `colSums()`: Returns a vector where each element represents the total of the values of the corresponding column.
- `rbind()`: Adds a row or multiple rows to a matrix. It can also be used to merge multiple matrices or vectors together by row.
- `cbind()`: Adds a column or multiple column to a matrix. It can also be used to merge multiple matrices or vectors together by column.

In the following demo, we are interested in the daily earnings of two merchants, John and Jane over a period of one week. First, we create an earnings vector for John (`earnings_John`) and Jane (`earnings_Jane`), respectively. Then, we combine them into a named matrix (`earnings_combined`) and we display the results:

```
# Create earnings vector for merchant John
earnings_John <- c(50, 60, 55, 74, 80)
# Create earnings vector for merchant Jane
earnings_Jane <- c(53, 57, 79, 88, 93)
# Create vector containing names of the days of the week
weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
# Create vector containing names of the merchants
merchant_names <- c("John", "Jane")
# Creating a matrix to store the merchants' earnings together
earnings_combined <- matrix(c(earnings_John, earnings_Jane), byrow = TRUE,
nrow = 2)
# Naming a matrix
rownames(earnings_combined) <- merchant_names
colnames(earnings_combined) <- weekdays
# Viewing matrix
earnings_combined
```

| | Monday | Tuesday | Wednesday | Thursday | Friday |
|------|--------|---------|-----------|----------|--------|
| John | 50 | 60 | 55 | 74 | 80 |
| Jane | 53 | 57 | 79 | 88 | 93 |

If we want to add the earnings for Saturday and Sunday, we first create a named matrix to store the weekend earnings (`earnings_combined_weekend`) and then merge it with `earnings_combined` using `cbind()` to create a new matrix called `earnings_whole_week`. As the name suggests, the latter contains the earnings for Jane and John for the entire week.

```
# Creating earnings vectors for weekend
earnings_John_weekend <- c(110, 120)
earnings_Jane_weekend <- c(100, 130)
weekends <- c("Saturday", "Sunday")
merchant_names
[1] "John" "Jane"
earnings_combined_weekend <- matrix(c(earnings_John_weekend,
earnings_Jane_weekend), byrow = TRUE, nrow = 2)
earnings_combined_weekend
```

| | [,1] | [,2] |
|------|------|------|
| [1,] | 110 | 120 |
| [2,] | 100 | 130 |

```
# Assigning row and column names
rownames(earnings_combined_weekend) <- merchant_names
colnames(earnings_combined_weekend) <- weekends
# Viewing weekend matrix
earnings_combined_weekend
  Saturday Sunday
John      110   120
Jane      100   130
# Viewing weekday matrix
earnings_combined
  Monday Tuesday Wednesday Thursday Friday
John    50     60         55       74     80
Jane    53     57         79       88     93
# We now merge them together
earnings_whole_week <- cbind(earnings_combined, earnings_combined_weekend)
earnings_whole_week
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
John    50     60         55       74     80      110     120
Jane    53     57         79       88     93      100     130
```

Now, suppose we want to add a third merchant called Tim. We can do so using the `rbind()` function:

```
# Create earnings vector for Tim
earnings_Tim <- c(40, 48, 75, 65, 29, 67, 84)
# Add earnings_Tim to earnings_whole_week
earnings_whole_week <- rbind(earnings_whole_week, earnings_Tim)
# Naming rows
rownames(earnings_whole_week) <- c("John", "Jane", "Tim")
# Viewing updated matrix
earnings_whole_week
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
John    50     60         55       74     80      110     120
Jane    53     57         79       88     93      100     130
Tim     40     48         75       65     29       67      84
```

Finally, suppose we want to find the:

- Total earnings for the three merchants per day.
- Total earnings for each merchant for the whole week.

We can find these using the `colSums()` and `rowSums()` functions respectively:

```
# Total earnings for the three merchants per day
total_earnings_per_day <- colSums(earnings_whole_week)
total_earnings_per_day
  Monday  Tuesday Wednesday  Thursday   Friday  Saturday   Sunday
    143     165      209      227     202     277     334
# Total earnings for each merchant for the whole week
total_earnings_for_week <- rowSums(earnings_whole_week)
total_earnings_for_week
John Jane  Tim
  549  600  408
```

Just like vectors, we can also select specific matrix elements. Here are some examples:

```
# Select only first row
John_Only <- earnings_whole_week[1,]
```

```

John_Only
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
    50      60      55      74      80      110      120
# Select only first and third rows
John_and_Tim_only <- earnings_whole_week[c(1,3),]
John_and_Tim_only
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
John    50      60      55      74      80      110      120
Tim     40      48      75      65      29      67      84
# Select only third column
Wednesday_only <- earnings_whole_week[,3]
Wednesday_only
John Jane Tim
  55   79   75
# Selecting only fourth to seventh columns
Thursday_to_Sunday_only <- earnings_whole_week[,4:7]
Thursday_to_Sunday_only
  Thursday Friday Saturday Sunday
John     74     80     110     120
Jane     88     93     100     130
Tim      65     29     67     84
# Selecting only fourth to sixth columns and first and third rows
selection <- earnings_whole_week[c(1,3), 4:7]
selection
  Thursday Friday Saturday Sunday
John     74     80     110     120
Tim      65     29     67     84
# Select by name
selection_1 <- earnings_whole_week[c("John", "Tim"), c("Thursday",
"Friday", "Saturday", "Sunday")]
selection_1
  Thursday Friday Saturday Sunday
John     74     80     110     120
Tim      65     29     67     84

```

Finally, just like vectors, we can perform element-wise arithmetic operations with matrices:

```

mat1 <- matrix(1:4, nrow = 2)
mat2 <- matrix(5:8, nrow = 2)
mat1
  [,1] [,2]
[1,]   1   3
[2,]   2   4
mat2
  [,1] [,2]
[1,]   5   7
[2,]   6   8
mat1 + mat2
  [,1] [,2]
[1,]   6  10
[2,]   8  12
mat2 - mat1
  [,1] [,2]
[1,]   4   4
[2,]   4   4
mat1 * mat2
  [,1] [,2]
[1,]   5  21
[2,]  12  32
mat1 / mat2

```

```

      [,1]      [,2]
[1,] 0.2000000 0.4285714
[2,] 0.3333333 0.5000000
mat1 ^ mat2
      [,1] [,2]
[1,]     1 2187
[2,]    64 65536

```

Note that R performed the matrix multiplication element-wise. This is not the standard way that matrices are multiplied. To perform matrix multiplication in the standard way, we can use the `%*%` operator like this:

```

A <- matrix(c(2,4,6,8), nrow = 2)
B <- matrix(c(1,3,4,7,9,10), nrow = 2)
A
      [,1] [,2]
[1,]     2     6
[2,]     4     8
B
      [,1] [,2] [,3]
[1,]     1     4     9
[2,]     3     7    10
# Standard matrix multiplication to calculate AB
AB <- A %*% B
AB
      [,1] [,2] [,3]
[1,]    20    50    78
[2,]    28    72   116

```

Arrays

An array is the multi-dimensional extension of matrices in R. In fact, a matrix can be viewed as a two-dimensional array. Just like vectors and matrices, an array can only store one data type at a time.

To create an array, we use the `array()` function. The two main arguments of the `array()` function are:

- **data:** This is a vector which contains the elements to be stored in the array.
- **dim:** This specifies the dimension of the array. For example, if we set `dim = c(4, 3, 2)`, we are telling R to create an array consisting of 2 matrices, each with 4 rows and 3 columns.
- **dimnames:** This optional argument specifies the names of the dimensions.

Again, you cannot have different data types inside the same matrix.

Here is a simple example of how to create an array:

```

# Create a vector of input elements
vector <- c(1,2,3,5,7,1,3,6,7,8,9,9)
# Use the above vector to create an array of two 3x2 matrices
my_array <- array(vector, dim = c(3,2,2))
my_array
, , 1

```

```

      [,1] [,2]
[1,]    1    5
[2,]    2    7
[3,]    3    1

```

```

, , 2

```

```

      [,1] [,2]
[1,]    3    8
[2,]    6    9
[3,]    7    9

```

Suppose there are only 3 students in a school. Suppose further that there are 3 terms in an academic year and that each student takes 5 tests during each term. The names of the students are “John,” “Jane” and “Tim.” Assume that we have 9 vectors of length 5, each containing the test results for one student for one term.

```

John_Term1 <- c(74, 72, 71, 79, 90)
John_Term2 <- c(78, 80, 80, 88, 82)
John_Term3 <- c(85, 90, 72, 77, 86)
Jane_Term1 <- c(63, 43, 62, 85, 65)
Jane_Term2 <- c(60, 27, 74, 63, 57)
Jane_Term3 <- c(55, 72, 64, 47, 75)
Tim_Term1 <- c(81, 83, 90, 84, 94)
Tim_Term2 <- c(99, 77, 99, 87, 91)
Tim_Term3 <- c(80, 94, 95, 87, 80)

```

The above code block is not complete. Complete it by constructing the 9 vectors using the above names, using some numeric values for the test results.

Currently, this data is stored in 9 different objects (9 different vectors). We want to store them in one single object, that is, we want to store them in one single array. There are many ways we can define the dimensions of our array but a natural choice is to store the values in an array consisting of three (5×3) matrices, where each matrix represents the marks for one student only. Inside a matrix, each row will represent a test and each column will correspond to a term.

```

marks_combined <- array(c(John_Term1, John_Term2, John_Term3,
                          Jane_Term1, Jane_Term2, Jane_Term3,
                          Tim_Term1, Tim_Term2, Tim_Term3),
                        dim = c(5, 3, 3))

```

```

marks_combined
, , 1

```

```

      [,1] [,2] [,3]
[1,]   74   78   85
[2,]   72   80   90
[3,]   71   80   72
[4,]   79   88   77
[5,]   90   82   86

```

```

, , 2

```

```

      [,1] [,2] [,3]
[1,]   63   60   55
[2,]   43   27   72

```

```
[3,] 62 74 64
[4,] 85 63 47
[5,] 65 57 75
```

```
, , 3
```

```
      [,1] [,2] [,3]
[1,] 81 99 80
[2,] 83 77 94
[3,] 90 99 95
[4,] 84 87 87
[5,] 94 91 80
```

The dimension of an array (and that of matrices!) can be found as

```
dim(marks_combined)
[1] 5 3 3
```

We can use the `dimnames` argument of the `array()` function to add names to the dimensions of the array.

```
matrix_names <- c("John", "Jane", "Tim")
row_names <- c("Test 1", "Test 2", "Test 3", "Test 4", "Test 5")
column_names <- c("Term 1", "Term 2", "Term 3")
# Let us create an upgraded version of our array
marks_combined_upgraded <- array(c(John_Term1, John_Term2, John_Term3,
                                   Jane_Term1, Jane_Term2, Jane_Term3,
                                   Tim_Term1, Tim_Term2, Tim_Term3),
                                dim = c(5, 3, 3),
                                dimnames = list(row_names, column_names,
matrix_names))
marks_combined_upgraded
, , John
```

```
      Term 1 Term 2 Term 3
Test 1    74    78    85
Test 2    72    80    90
Test 3    71    80    72
Test 4    79    88    77
Test 5    90    82    86
```

```
, , Jane
```

```
      Term 1 Term 2 Term 3
Test 1    63    60    55
Test 2    43    27    72
Test 3    62    74    64
Test 4    85    63    47
Test 5    65    57    75
```

```
, , Tim
```

```
      Term 1 Term 2 Term 3
Test 1    81    99    80
Test 2    83    77    94
Test 3    90    99    95
Test 4    84    87    87
Test 5    94    91    80
```

Just like with vectors and matrices, we can access array elements in the usual way by using square brackets []:

```
# Select first matrix only
marks_combined_upgraded[, , 1]
  Term 1 Term 2 Term 3
Test 1   74   78   85
Test 2   72   80   90
Test 3   71   80   72
Test 4   79   88   77
Test 5   90   82   86
# Select Tim's matrix only (using name instead of index)
marks_combined_upgraded[, , "Tim"]
  Term 1 Term 2 Term 3
Test 1   81   99   80
Test 2   83   77   94
Test 3   90   99   95
Test 4   84   87   87
Test 5   94   91   80
# Selecting John and Tim's matrices
marks_combined_upgraded[, , c("John", "Tim")]
, , John
  Term 1 Term 2 Term 3
Test 1   74   78   85
Test 2   72   80   90
Test 3   71   80   72
Test 4   79   88   77
Test 5   90   82   86
, , Tim
  Term 1 Term 2 Term 3
Test 1   81   99   80
Test 2   83   77   94
Test 3   90   99   95
Test 4   84   87   87
Test 5   94   91   80
# Selecting only Term 1 and Term 3 marks
marks_combined_upgraded[,c("Term 1", "Term 3"),]
, , John
  Term 1 Term 3
Test 1   74   85
Test 2   72   90
Test 3   71   72
Test 4   79   77
Test 5   90   86
, , Jane
  Term 1 Term 3
Test 1   63   55
Test 2   43   72
Test 3   62   64
Test 4   85   47
Test 5   65   75
, , Tim
```



```

      Term 1 Term 3
Test 1      81     80
Test 2      83     94
Test 3      90     95
Test 4      84     87
Test 5      94     80
# Selecting John and Tim's Test 3 results for Term 2
marks_combined_upgraded["Test 2", "Term 2", c("John", "Tim")]
John Tim
  80   77

```

Finally, we can perform calculations across the array elements using the `apply()` function. This function takes three arguments:

- **X:** An array (or a matrix) containing the data we are interested in.
- **MARGIN:** This specifies whether the function will be applied on rows only (set `MARGIN = c(1)`), columns only (set `MARGIN = c(2)`), or both rows and columns (set `MARGIN = c(1,2)`).
- **FUN:** This specifies the function to be applied. It can be any compatible function. Examples include but are not limited to: `sum()`, `min()`, `max()`, `mean()`, `median()`, `sd()`, `var()`.

In the following examples, we use the `sum()` function to calculate various interesting totals.

```

# Find total marks for all students, across all tests and terms
sum(marks_combined_upgraded)
[1] 3437
# Find total marks for all students per term
apply(marks_combined_upgraded, c(2), sum)
Term 1 Term 2 Term 3
  1136   1142   1159
# Find the total marks for all students per test per term
apply(marks_combined_upgraded, c(1,2), sum)
      Term 1 Term 2 Term 3
Test 1    218    237    220
Test 2    198    184    256
Test 3    223    253    231
Test 4    248    238    211
Test 5    249    230    241
# (Less useful in this context) Find the total marks for all students per
test
apply(marks_combined_upgraded, c(1), sum)
Test 1 Test 2 Test 3 Test 4 Test 5
   675   638   707   697   720

```

Lists

Lists can hold elements of different data types. They allow us to store multiple objects under one single object in an orderly fashion. Lists can store vectors, matrices, data frames and even other lists.

To create a list, we use the `list()` function. The arguments of the `list()` function represent the components of the list.

Below is an example where we create a list containing a vector, a matrix and a data frame. We create the data frame `my_data_frame` by selecting the first 8 rows of the build-in `mtcars` dataset.

```
# Creating a simple vector
my_vector <- 1:10
# Creating a simple matrix
my_matrix <- matrix(c(4, 6, 7, 1), nrow = 2)
# Creating a simple data frame
my_data_frame <- mtcars[1:8, ]
my_vector
[1] 1 2 3 4 5 6 7 8 9 10
my_matrix
      [,1] [,2]
[1,]    4    7
[2,]    6    1
my_data_frame
      mpg  cyl  disp  hp  drat    wt  qsec vs  am  gear  carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46 0   1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02 0   1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61 1   1    4    1
Hornet 4 Drive  21.4   6 258.0 110 3.08 3.215 19.44 1   0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02 0   0    3    2
Valiant        18.1   6 225.0 105 2.76 3.460 20.22 1   0    3    1
Duster 360     14.3   8 360.0 245 3.21 3.570 15.84 0   0    3    4
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00 1   0    4    2
# Creating a list from the above variables
my_list <- list(my_vector, my_matrix, my_data_frame)
my_list
[[1]]
[1] 1 2 3 4 5 6 7 8 9 10

[[2]]
      [,1] [,2]
[1,]    4    7
[2,]    6    1

[[3]]
      mpg  cyl  disp  hp  drat    wt  qsec vs  am  gear  carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46 0   1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02 0   1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61 1   1    4    1
Hornet 4 Drive  21.4   6 258.0 110 3.08 3.215 19.44 1   0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02 0   0    3    2
Valiant        18.1   6 225.0 105 2.76 3.460 20.22 1   0    3    1
Duster 360     14.3   8 360.0 245 3.21 3.570 15.84 0   0    3    4
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00 1   0    4    2
```

Again, if we want to name the components of our list, we use the `names()` function:

```
my_names <- c("I am a Vector",
              "I am a Matrix",
              "I am a Data Frame")
names(my_list) <- my_names
my_list
$I am a Vector`
 [1]  1  2  3  4  5  6  7  8  9 10

$I am a Matrix`
      [,1] [,2]
[1,]    4    7
[2,]    6    1

$I am a Data Frame`
      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Mazda RX4           21.0   6  160.0 110 3.90 2.620 16.46 0  1   4    4
Mazda RX4 Wag       21.0   6  160.0 110 3.90 2.875 17.02 0  1   4    4
Datsun 710           22.8   4  108.0  93 3.85 2.320 18.61 1  1   4    1
Hornet 4 Drive       21.4   6  258.0 110 3.08 3.215 19.44 1  0   3    1
Hornet Sportabout   18.7   8  360.0 175 3.15 3.440 17.02 0  0   3    2
Valiant             18.1   6  225.0 105 2.76 3.460 20.22 1  0   3    1
Duster 360          14.3   8  360.0 245 3.21 3.570 15.84 0  0   3    4
Merc 240D            24.4   4  146.7  62 3.69 3.190 20.00 1  0   4    2
```

Now, it's time to learn how to access the components of our list. We use:

The `$` sign to access a component by name. Double square brackets `[[]]` to access a component by index or by name. Any components in the components of the list can, then, be further accessed according to their type.

Here is a demonstration of this:

```
# Selecting first component by index
my_list[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10
# Selecting first component by name
my_list$I am a Vector`
 [1]  1  2  3  4  5  6  7  8  9 10
# Selecting second component by name using the $ notation
my_list$I am a Matrix`
      [,1] [,2]
[1,]    4    7
[2,]    6    1
# Selecting second component by name using square brackets
my_list[["I am a Matrix"]]
      [,1] [,2]
[1,]    4    7
[2,]    6    1
# Selecting fourth element of first component by index only
my_list[[1]][4]
 [1]  4
# Selecting fourth element of first component by name and index
my_list$I am a Vector`[4]
 [1]  4
# Selecting first row of second component by index only
my_list[[2]][1,]
 [1]  4  7
```

```
# Selecting first element of second component by name and index
my_list$I am a Matrix[1,1]
[1] 4
# Selecting first element of second component by index only
my_list[[2]][1,1]
[1] 4
```

Data Frames

A data frame in R is a matrix-like object where each column contains values of one variable, with one big advantage: it can store different data types. While each column can only contain values of a single data type, we can have columns of different data types. Another requirement is that each column should contain the same number of items or observations.

In the previous section, we saw the built-in `mtcars` data frame. The name is an abbreviated version of Motor Trend Car Road Tests. According to the dataset description (see `?mtcars`), “the data was extracted from the 1974 *Motor Trend US magazine*, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles.” Here is the dataset

```
mtcars
```

| | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
|---------------------|------|-----|-------|-----|------|-------|-------|----|----|------|------|
| Mazda RX4 | 21.0 | 6 | 160.0 | 110 | 3.90 | 2.620 | 16.46 | 0 | 1 | 4 | 4 |
| Mazda RX4 Wag | 21.0 | 6 | 160.0 | 110 | 3.90 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| Datsun 710 | 22.8 | 4 | 108.0 | 93 | 3.85 | 2.320 | 18.61 | 1 | 1 | 4 | 1 |
| Hornet 4 Drive | 21.4 | 6 | 258.0 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| Hornet Sportabout | 18.7 | 8 | 360.0 | 175 | 3.15 | 3.440 | 17.02 | 0 | 0 | 3 | 2 |
| Valiant | 18.1 | 6 | 225.0 | 105 | 2.76 | 3.460 | 20.22 | 1 | 0 | 3 | 1 |
| Duster 360 | 14.3 | 8 | 360.0 | 245 | 3.21 | 3.570 | 15.84 | 0 | 0 | 3 | 4 |
| Merc 240D | 24.4 | 4 | 146.7 | 62 | 3.69 | 3.190 | 20.00 | 1 | 0 | 4 | 2 |
| Merc 230 | 22.8 | 4 | 140.8 | 95 | 3.92 | 3.150 | 22.90 | 1 | 0 | 4 | 2 |
| Merc 280 | 19.2 | 6 | 167.6 | 123 | 3.92 | 3.440 | 18.30 | 1 | 0 | 4 | 4 |
| Merc 280C | 17.8 | 6 | 167.6 | 123 | 3.92 | 3.440 | 18.90 | 1 | 0 | 4 | 4 |
| Merc 450SE | 16.4 | 8 | 275.8 | 180 | 3.07 | 4.070 | 17.40 | 0 | 0 | 3 | 3 |
| Merc 450SL | 17.3 | 8 | 275.8 | 180 | 3.07 | 3.730 | 17.60 | 0 | 0 | 3 | 3 |
| Merc 450SLC | 15.2 | 8 | 275.8 | 180 | 3.07 | 3.780 | 18.00 | 0 | 0 | 3 | 3 |
| Cadillac Fleetwood | 10.4 | 8 | 472.0 | 205 | 2.93 | 5.250 | 17.98 | 0 | 0 | 3 | 4 |
| Lincoln Continental | 10.4 | 8 | 460.0 | 215 | 3.00 | 5.424 | 17.82 | 0 | 0 | 3 | 4 |
| Chrysler Imperial | 14.7 | 8 | 440.0 | 230 | 3.23 | 5.345 | 17.42 | 0 | 0 | 3 | 4 |
| Fiat 128 | 32.4 | 4 | 78.7 | 66 | 4.08 | 2.200 | 19.47 | 1 | 1 | 4 | 1 |
| Honda Civic | 30.4 | 4 | 75.7 | 52 | 4.93 | 1.615 | 18.52 | 1 | 1 | 4 | 2 |
| Toyota Corolla | 33.9 | 4 | 71.1 | 65 | 4.22 | 1.835 | 19.90 | 1 | 1 | 4 | 1 |
| Toyota Corona | 21.5 | 4 | 120.1 | 97 | 3.70 | 2.465 | 20.01 | 1 | 0 | 3 | 1 |
| Dodge Challenger | 15.5 | 8 | 318.0 | 150 | 2.76 | 3.520 | 16.87 | 0 | 0 | 3 | 2 |
| AMC Javelin | 15.2 | 8 | 304.0 | 150 | 3.15 | 3.435 | 17.30 | 0 | 0 | 3 | 2 |
| Camaro Z28 | 13.3 | 8 | 350.0 | 245 | 3.73 | 3.840 | 15.41 | 0 | 0 | 3 | 4 |
| Pontiac Firebird | 19.2 | 8 | 400.0 | 175 | 3.08 | 3.845 | 17.05 | 0 | 0 | 3 | 2 |
| Fiat X1-9 | 27.3 | 4 | 79.0 | 66 | 4.08 | 1.935 | 18.90 | 1 | 1 | 4 | 1 |
| Porsche 914-2 | 26.0 | 4 | 120.3 | 91 | 4.43 | 2.140 | 16.70 | 0 | 1 | 5 | 2 |
| Lotus Europa | 30.4 | 4 | 95.1 | 113 | 3.77 | 1.513 | 16.90 | 1 | 1 | 5 | 2 |
| Ford Pantera L | 15.8 | 8 | 351.0 | 264 | 4.22 | 3.170 | 14.50 | 0 | 1 | 5 | 4 |
| Ferrari Dino | 19.7 | 6 | 145.0 | 175 | 3.62 | 2.770 | 15.50 | 0 | 1 | 5 | 6 |
| Maserati Bora | 15.0 | 8 | 301.0 | 335 | 3.54 | 3.570 | 14.60 | 0 | 1 | 5 | 8 |
| Volvo 142E | 21.4 | 4 | 121.0 | 109 | 4.11 | 2.780 | 18.60 | 1 | 1 | 4 | 2 |

Each column represents one aspect of the cars and each row represents all the aspects of one car.

This dataset contains only 32 observations, so we can display it entirely on one single page. However, what if we have a very long dataset? Can we only display part of it to gain a feel of the data? The answer is yes. To do so, we can use the `head()` and `tail()` functions. Let's try this with another built-in dataset called `iris`:

```
# How many rows are there in the iris data set?
nrow(iris)
[1] 150
# Displaying only the first six rows
head(iris, n = 6)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4          0.2  setosa
2          4.9         3.0          1.4          0.2  setosa
3          4.7         3.2          1.3          0.2  setosa
4          4.6         3.1          1.5          0.2  setosa
5          5.0         3.6          1.4          0.2  setosa
6          5.4         3.9          1.7          0.4  setosa
# Displaying only the first ten rows
head(iris, n = 10)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4          0.2  setosa
2          4.9         3.0          1.4          0.2  setosa
3          4.7         3.2          1.3          0.2  setosa
4          4.6         3.1          1.5          0.2  setosa
5          5.0         3.6          1.4          0.2  setosa
6          5.4         3.9          1.7          0.4  setosa
7          4.6         3.4          1.4          0.3  setosa
8          5.0         3.4          1.5          0.2  setosa
9          4.4         2.9          1.4          0.2  setosa
10         4.9         3.1          1.5          0.1  setosa
# Displaying only the last six rows. The argument n is set to 6 by default.
tail(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
145          6.7         3.3          5.7          2.5 virginica
146          6.7         3.0          5.2          2.3 virginica
147          6.3         2.5          5.0          1.9 virginica
148          6.5         3.0          5.2          2.0 virginica
149          6.2         3.4          5.4          2.3 virginica
150          5.9         3.0          5.1          1.8 virginica
```

Another way to get rapid insight into a data frame is via the `str()` function. The following information is returned when calling the `str()` function on a data frame:

1. The total number of observations (or rows)
2. The total number of variables (or columns)
3. A complete list of the names of the variables
4. The data type of each variable
5. A glimpse of the first few observations

```
str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num   6  6  4  6  8  6  8  4  4  6 ...
 $ disp: num  160 160 108 258 360 ...
```

```

$ hp : num 110 110 93 110 175 105 245 62 95 123 ...
$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt : num 2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num 16.5 17 18.6 19.4 17 ...
$ vs : num 0 0 1 1 0 1 0 1 1 1 ...
$ am : num 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num 4 4 1 1 2 1 4 2 2 4 ...
str(iris)
'data.frame': 150 obs. of 5 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...

```

We can create data frames through the `data.frame()` function. As arguments, we simply pass the vectors that will become the various columns of our data frame. While the vectors can be of different data types, they need to be of the same length.

Suppose there are 5 students in a class and they sit an exam where the pass mark is 75 out of 100. Let us create a data frame called `Class_DF` that contains the following variables:

- Student: The name of the student
- Marks: The number of marks (out of 100) for each student
- Rank: The rank of each student
- Passed: Whether each student has passed or not

Here is how we can create the data frame:

```

# Creating variables
Student <- c("Jane", "John", "Tim", "Michael", "Emma")
Marks <- c(80,95,98.5,78.4,50.25)
Rank <- as.integer(c(3,2,1,4,5))
Passed <- c(TRUE,TRUE,TRUE,TRUE,FALSE)
# Creating Data Frame
Class_DF <- data.frame(Student, Marks, Rank, Passed)
str(Class_DF)
'data.frame': 5 obs. of 4 variables:
 $ Student: chr "Jane" "John" "Tim" "Michael" ...
 $ Marks : num 80 95 98.5 78.4 50.2
 $ Rank : int 3 2 1 4 5
 $ Passed : logi TRUE TRUE TRUE TRUE FALSE
Class_DF
  Student Marks Rank Passed
1 Jane 80.00 3 TRUE
2 John 95.00 2 TRUE
3 Tim 98.50 1 TRUE
4 Michael 78.40 4 TRUE
5 Emma 50.25 5 FALSE

```

Observe that each column has a different data type. We would not have been able to store this data in a matrix, so, in this sense, data frames are super convenient. In fact, data frames are very popular. You are very likely to encounter them regularly when working with R. In a

later week, we will use the `dplyr` package to do all sorts of interesting operations on data frames.

Just like matrices, we can select elements of data frames in the usual way by using square brackets `[]`.

```
# Select first column
Class_DF[, 1]
[1] "Jane"      "John"      "Tim"       "Michael"   "Emma"
# Select first to third columns
Class_DF[, 1:3]
  Student Marks Rank
1   Jane 80.00    3
2   John 95.00    2
3    Tim 98.50    1
4 Michael 78.40    4
5   Emma 50.25    5
# Select second and fourth columns
Class_DF[, c(2, 4)]
  Marks Passed
1 80.00  TRUE
2 95.00  TRUE
3 98.50  TRUE
4 78.40  TRUE
5 50.25 FALSE
# Select first row
Class_DF[1,]
  Student Marks Rank Passed
1   Jane    80    3  TRUE
# Select first to third rows
Class_DF[1:3,]
  Student Marks Rank Passed
1   Jane  80.0    3  TRUE
2   John  95.0    2  TRUE
3    Tim  98.5    1  TRUE
# Select second and fourth rows
Class_DF[c(2, 4), ]
  Student Marks Rank Passed
2   John  95.0    2  TRUE
4 Michael  78.4    4  TRUE
```

In the above demo, we have used selection by index. We could have done the same thing via selection by name. Data frames are formally a special kind of list. So, we can also use the `$` notation to select specific columns:

```
Class_DF$Student
[1] "Jane"      "John"      "Tim"       "Michael"   "Emma"
Class_DF$Marks
[1] 80.00 95.00 98.50 78.40 50.25
Class_DF$Rank
[1] 3 2 1 4 5
Class_DF$Passed
[1] TRUE TRUE TRUE FALSE
# Selecting name of second student
Class_DF$Student[2]
[1] "John"
```

Now, suppose we want to select rows based on some condition. Consider the following different scenarios. Suppose we only want to display observations where:

- The student has passed
- The marks are higher than 90
- The students are ranked among the top 3
- The student's name contains exactly 4 letters.

To subset a data frame based on some condition, we use the `subset()` function as follows:

```
# Scenario 1: Display only students who have passed
subset(Class_DF, Passed == TRUE)
  Student Marks Rank Passed
1   Jane  80.0    3   TRUE
2   John  95.0    2   TRUE
3    Tim  98.5    1   TRUE
4 Michael  78.4    4   TRUE
# Scenario 2: Display only students who scored higher than 90 marks
subset(Class_DF, Marks > 90)
  Student Marks Rank Passed
2   John  95.0    2   TRUE
3    Tim  98.5    1   TRUE
# Scenario 3: Display only students ranked among the top 3
subset(Class_DF, Rank <= 3)
  Student Marks Rank Passed
1   Jane  80.0    3   TRUE
2   John  95.0    2   TRUE
3    Tim  98.5    1   TRUE
# Scenario 4: Display only students whose name contain exactly 4 letters
subset(Class_DF, nchar(as.character(Student)) == 4)
  Student Marks Rank Passed
1   Jane 80.00    3   TRUE
2   John 95.00    2   TRUE
5   Emma 50.25    5  FALSE
```

Coerce and Test

As with types we can use `as.*()` and `is.*()` functions to coerce and test data structures in R. Try to understand the result of the commands below.

```
is.data.frame(mtcars)
[1] TRUE
is.matrix(mtcars)
[1] FALSE
is.list(mtcars)
[1] TRUE
as.matrix(iris[1:5, ])
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 "5.1"         "3.5"         "1.4"         "0.2"         "setosa"
2 "4.9"         "3.0"         "1.4"         "0.2"         "setosa"
3 "4.7"         "3.2"         "1.3"         "0.2"         "setosa"
4 "4.6"         "3.1"         "1.5"         "0.2"         "setosa"
5 "5.0"         "3.6"         "1.4"         "0.2"         "setosa"
as.list(matrix(1:10, nrow = 2))
[[1]]
[1] 1
```



```
[[2]]  
[1] 2  
  
[[3]]  
[1] 3  
  
[[4]]  
[1] 4  
  
[[5]]  
[1] 5  
  
[[6]]  
[1] 6  
  
[[7]]  
[1] 7  
  
[[8]]  
[1] 8  
  
[[9]]  
[1] 9  
  
[[10]]  
[1] 10  
is.list(letters[1:4])  
[1] FALSE
```

Useful Links and Resources

- [“R objects” section from *Hands-on programming with R*](#)

Data Structures in Python



Working With Python

We begin with some basics on Python before we talk specifically about data structures.

Dependencies

An essential part of Python is to incorporate dependencies, also known as packages. These are modules of existing code and functions which you can base your code upon. They may actually include code written by you. A wide range of dependencies is available with the Anaconda installation so you will probably only need to activate them. But if you need to install a package with Anaconda you need to use `conda` commands in the QtConsole (macOS) or the Anaconda prompt (Windows). For example, `conda install Quandl`. Sometimes a package is not available as a `conda` package but can be installed by `pip install`, for example `pip install Quandl`.

To find out more about `conda` please consult [this guide](#). That guide also provides a handy cheat-sheet if you want to learn, for example, about updating packages.

To use a dependency in a Python package, the basic code is

```
import _package_ as _alias_
```

For example, the below code is using the alias `np` for the Python package `numpy` and printing the number `\(pi\)`:

```
import numpy as np
print(np.pi)
3.141592653589793
```

Note that the alias `np` above (or any other you want to use) need to be placed before `pi`, i.e. `np.pi`. In fact, using an alias is optional, yet recommended practice. For more on the numpy package see the dedicated set of notes that follows.

Data Types and Structures

Python has a number of built in data types; some of them are given below:

1. `float`
2. `int`
3. Strings
4. Boolean

5. Tuples
6. Lists
7. Sets
8. Dictionaries

float, int, String and Boolean Types

float and int are for real numbers and integers respectively. The type() method can identify the kind of a data type, see the code below for an example:

```
a=4.6
print(type(a))
<class 'float'>

b=10
print(type(b))
<class 'int'>
```

Strings are for text and have type str. The code below demonstrates Python's character processing capabilities along with their output:

```
phrase='All models are wrong, but some are useful.'
print(phrase[0:3]) #print the first three characters
All
print(phrase.find('models')) #Find the index where the word 'models' starts.
4
print(phrase.find('right')) # when the word does not exist
-1
print(phrase.lower()) #set all to lower case
all models are wrong, but some are useful.
print(phrase.upper()) #set all to upper case
ALL MODELS ARE WRONG, BUT SOME ARE USEFUL.
a=phrase.split(',')
print(a)
['All models are wrong', ' but some are useful.']
```

Finally Boolean type is bool and is typically used for the results of true/false statements. For example,

```
k=1>3
print(k)
False
print(type(k))
<class 'bool'>
```

Data Structures

Data structures or collections in Python are containers that are used to store collections of data of potentially different types. There are four collection data types in the Python programming language:

- Tuples are collections which is ordered and unchangeable. A tuple can have duplicate members.
- Lists are collections which is ordered and changeable. A list can have duplicate members.
- Sets are collections which is unordered and unindexed. A set cannot have duplicate members.
- Dictionaries are collections which is unordered, changeable and indexed. A dictionary cannot have duplicate members.

Tuples and Indexing

A tuple is a collection of data types which is ordered and unchangeable. In Python tuples are written with round brackets.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1)
('apple', 'banana', 'cherry')
tuple2=(1.0,3.0,7.0)
print(tuple2)
(1.0, 3.0, 7.0)
```

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

You can access tuple items by referring to the index number, inside square brackets.

Python indexing: Note that in Python the first item of a tuple (or any other collection) has index 0, not 1! Similarly the second item has index 1, the third item has index 2 and so on. This is different to R where the first item is indexed with 1, the second with 2 and so on.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1[0])
apple
print(tuple1[2])
cherry
```

Negative indexing is also possible and it means starting counting from the end; -1 refers to the last item, -2 refers to the second last item etc.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1[-1])
cherry
```

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

For example, if we want to return the third, fourth, and the fifth item, we can use the following code:

```
tuple3 = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(tuple3[2:5])

('cherry', 'orange', 'kiwi')
```

Lists, Sets and Dictionaries

Here we see three other types of Python collections. One of the ways in which they all differ from tuples is that, unlike tuples, they are mutable. In other words you can change the values of a list, set or a dictionary but you cannot do that with tuples.

A list is a collection which is ordered and changeable. In Python lists are defined with square brackets.

```
list1 = ["apple", "banana", "cherry"]
print(list1)

['apple', 'banana', 'cherry']
```

For example if we want to print the second item of the list, we can use the following code:

```
print(list1[1])

banana
```

To illustrate that a list is mutable let us check the following example:

```
list1 = ["apple", "banana", "cherry"]
print(list1)

['apple', 'banana', 'cherry']
list1[1] = "apple"
print(list1)

['apple', 'apple', 'cherry']
```

A set is a collection which is unordered and unindexed. In Python, sets are defined with curly brackets.

```
set = {"apple", "banana", "cherry"}
print(set)

{'banana', 'cherry', 'apple'}
```

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are defined with curly brackets, and they have keys and values.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

```
}  
print(thisdict)  
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Items of a dictionary can be accessed by referring to their key names, inside square brackets:

```
thisdict["model"]  
'Mustang'
```

You can change the value of a specific item by referring to its key name:

For example if we want to change the “year” to 2018, we can use the following code:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["year"] = 2018
```

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)  
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

The `pop()` method removes the item with the specified key name:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")  
'Mustang'  
print(thisdict)  
{'brand': 'Ford', 'year': 1964}
```

Useful Links and Resources

- [Data structures documentation from python.org](#)

Working With NumPy and Pandas

**** Note:** The code chunks below should be run in the following order ******

NumPy

NumPy is the key package for numerical computing in Python. It allows you to perform operations on whole blocks of data without writing loops. The fast numerical processing it permits makes it the basis of most visualisation packages as well more sophisticated scientific and machine learning packages.

To activate it in Spyder scripts or Jupyter notebooks, simply add the following line at the beginning of your script or notebook.

```
import numpy as np
```

Note that the code blocks that follow in this page assume that **NumPy** has been activated. So make sure you execute the above line beforehand.

NumPy Arrays

A **NumPy** array, or else **ndarray**, is a grid of values of the same type, which is indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can create a **ndarray** object by using the **array()** function. Also the function **type()** gives us the type of the array.

```
array1 = np.array([1, 2, 3, 4, 5])
print(array1)
[1 2 3 4 5]
print(type(array1))
<class 'numpy.ndarray'>
```

To create an **ndarray**, we can also pass a list, tuple or any array-like object into the **array()** function, and it will be converted into an **ndarray**:

```
#NumPy arrays from tuples
arr = np.array((1, 2, 3, 4, 5))
print(arr)
[1 2 3 4 5]
```

Arrays may have dimensions of 0 (single numbers, strings or Booleans), 1 (vectors), 2 (matrices) or even 3 (tensors).

```
array0d = np.array(42) #0-dimensional array
print(array0d)
```



```

42
array1d = np.array([1, 2, 3, 4, 5]) #1-dimensional array
print(array1d)
[1 2 3 4 5]
array2d = np.array([[1, 2, 3], [4, 5, 6]]) #2-dimensional array
print(array2d)
[[1 2 3]
 [4 5 6]]

```

NumPy also provides many methods to create some commonly used arrays:

```

a = np.zeros((2,2))    # Create an array of all zeros
print(a)               # Prints "[[ 0.  0.]
                        #           [ 0.  0.]]"

[[0. 0.]
 [0. 0.]]

b = np.ones((1,2))    # Create an array of all ones
print(b)               # Prints "[[ 1.  1.]]"

[[1. 1.]]

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                        #           [ 7.  7.]]"

[[7 7]
 [7 7]]

d = np.eye(2)          # Create a 2x2 identity matrix
print(d)               # Prints "[[ 1.  0.]
                        #           [ 0.  1.]]"

[[1. 0.]
 [0. 1.]]

np.random.seed(10)

e = np.random.random((2,2)) # Create an array filled with random values
print(np.round(e,3))        # print the array with values rounded to 3 d.p.

[[0.771 0.021]
 [0.634 0.749]]

```

In the example below `np.sqrt` is a *vectorised* calculation on a **NumPy** array set up using the function `arange`. In other words the function `np.sqrt()` is applied to all the elements of the array `rng` at the same time rather than each of them separately.

```

rng=np.arange(10)
print(rng)
[0 1 2 3 4 5 6 7 8 9]

```

```
print(type(rng))
<class 'numpy.ndarray'>
print(np.sqrt(rng))
[0.          1.          1.41421356  1.73205081  2.          2.23606798
 2.44948974  2.64575131  2.82842712  3.          ]
```

NumPy Operations

NumPy provides some handy methods to summarise the data. For example

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])
print(a.shape)
(2, 3)
print(a)
[[ 1.  2.  4.]
 [-1.  2. -5.]]
print(a.sum(axis=0))
[ 0.  4. -1.]
print(a.sum(axis=1))
[ 7. -4.]
print(a.sum())
3.0
```

Note the Python indexing feature we saw on Python collections: `axis=0` refers to the first dimension of the 2-D array above; namely the rows. Similarly `axis=1` refers to its second dimension; namely the columns.

NumPy can also be used for matrix operations such as transposing and multiplying matrices:

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])
b=np.transpose(a)    # b is the transpose of a
print(b.shape)
(3, 2)
print(b)
[[ 1. -1.]
 [ 2.  2.]
 [ 4. -5.]]
c=np.dot(a,b)        #c = a * b
print(c)
[[ 21. -17.]
 [-17.  30.]]
c=a @ b              #c = a * b as before
print(c)
```

```
[[ 21. -17.]
 [-17.  30.]]
```

NumPy arrays can be easily subsetted.

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])
print(a)
[[ 1.  2.  4.]
 [-1.  2. -5.]]
print(a[0:2,1:3])
[[ 2.  4.]
 [ 2. -5.]]
```

You can also set the values in a **NumPy** array using simple Boolean expressions:

```
np.random.seed(0)
arr=np.random.random((3,3))
print(np.round(arr,3))
[[0.549 0.715 0.603]
 [0.545 0.424 0.646]
 [0.438 0.892 0.964]]
arr[arr<0.5]=0
print(arr)
[[0.5488135  0.71518937 0.60276338]
 [0.54488318 0.         0.64589411]
 [0.         0.891773   0.96366276]]
print(np.sum((arr<0) & (arr>1))) # The operator '&' stands for 'and'
0
print(np.sum((arr<0) | (arr>0.7))) # The operator '|' stands for 'or'
3
```

Pandas

The **pandas** package provides a major tool in the Python ecosystem. While it is possible to work with data with other Python tools such as the **NumPy** package (see the notes on Data structures with Python), the key difference is that **pandas** is designed for working with *heterogeneous* data. Moreover it gives Python a functionality similar to R Data frames and integrates powerful low level modelling tools such as data import, aggregation and cleaning.

Pandas is included in the Anaconda installation. To activate it in Spyder scripts or Jupyter notebooks, simply add the following lines in the beginning of your program.

```
import pandas as pd
```

Note: The code blocks that follow in this document assume that `Pandas` has been activated. So make sure you execute the above line beforehand. Some of the blocks also require **NumPy** so make sure this is activated as well.

Pandas Series

A `Series` is a one-dimensional labelled array capable of holding any data type (integers, strings, floating point numbers, etc.). The axis labels are collectively referred to as the index. The basic method to create a `Series` is to call:

```
s = pd.Series(data, index=index)
```

The object `data` can be several things, for example a Python dictionary or a `ndarray`. For `ndarrays`, the index must be the same length as data. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
np.random.seed(1)
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
s
a    1.624345
b   -0.611756
c   -0.528172
d   -1.072969
e    0.865408
dtype: float64
```

Pandas Data Frames

A **Pandas** data frame is a 2-dimensional labelled data structure with columns of potentially different types. You can think of it like a spreadsheet or R data frame or a dictionary of `Series` objects. It is generally the most commonly used `pandas` object. Like `Series`, data frames accept many different kinds of input, such as dictionaries, lists, sets, `Series`, `ndarrays` etc.

Along with the data, we can optionally pass index (row labels) and columns (column labels) arguments. If axis labels are not passed, they will be constructed from the input data based on some default options.

We can create a `pandas` data frame from a structured array in the following way

```
data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])
pd.DataFrame(data, index=['first', 'second'])
```

that gives the output

| | A | B | C |
|--------|---|-----|----|
| first | 0 | 0.0 | b" |
| second | 0 | 0.0 | b" |

pandas data frame

Below is how we can create a `pandas` data frame from dictionary of `ndarrays` or lists

```
d = {'one': [1., 2., 3., 4.],
     'two': [4., 3., 2., 1.]}
pd.DataFrame(d)
# repeat with pd.DataFrame(d, index=['a', 'b', 'c', 'd']) instead of pd.DataFrame(d)
```

that gives the output

| | one | two |
|---|-----|-----|
| 0 | 1.0 | 4.0 |
| 1 | 2.0 | 3.0 |
| 2 | 3.0 | 2.0 |
| 3 | 4.0 | 1.0 |

Another pandas Data Frame

Pandas provides a nice tool to get basic summaries from your data set. The following code creates the `pandas` data frame `df` and applies the `describe()` function on it.

```
d = {'one': [1., 2., 3., 4.],
     'two': [4., 3., 2., 1.]}
df=pd.DataFrame(d)
df.describe()
```

that gives the output

| | one | two |
|-------|----------|----------|
| count | 4.000000 | 4.000000 |
| mean | 2.500000 | 2.500000 |
| std | 1.290994 | 1.290994 |
| min | 1.000000 | 1.000000 |
| 25% | 1.750000 | 1.750000 |
| 50% | 2.500000 | 2.500000 |
| 75% | 3.250000 | 3.250000 |
| max | 4.000000 | 4.000000 |

Descriptive statistics from Pandas.

Also, in a very similar manner as with **NumPy**, we can use Boolean indexing to generate a subset of your data:

```
d = {'one': [1., 2., 3., 4.],
      'two': [4., 3., 2., 1.]}
df=pd.DataFrame(d)
print(df[df>0])
```

| | one | two |
|---|-----|-----|
| 0 | 1.0 | 4.0 |
| 1 | 2.0 | 3.0 |
| 2 | 3.0 | 2.0 |
| 3 | 4.0 | 1.0 |

Finally, indexing also works in a similar manner with **NumPy**. The additional feature here is that we can use the rows/column names in addition to the numbered indices.

```
d = {'one': [1., 2., 3., 4.],
      'two': [4., 3., 2., 1.]}
df=pd.DataFrame(d)
print(df['one'])
```

| | |
|---|-----|
| 0 | 1.0 |
| 1 | 2.0 |
| 2 | 3.0 |
| 3 | 4.0 |

Name: one, dtype: float64

```
print(df['two'])
```

| | |
|---|-----|
| 0 | 4.0 |
| 1 | 3.0 |
| 2 | 2.0 |
| 3 | 1.0 |

Name: two, dtype: float64

Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapters 4 and 5.
- [NumPy website](#)
- [10 minutes to pandas from the pandas website](#)

Data Exchange File Formats With Python

**** Note: The code chunks below should be run in the following order ****

File Formats

We have already described several file formats that provide standard ways to encode information to be stored in a file. They can usually be identified by looking at the file extension; e.g. a file saved with name `Data` in CSV format will appear as `Data.csv`. By noticing `.csv` extension we can clearly identify that it is a comma-separated file and data is stored in a specific way, in this case a tabular format.

An essential skill for a data scientist is to be able to understand the underlying structure of various file formats, and their advantages and disadvantages. There may also be situations in which decisions need to be taken on how to store data in order to improved the performance.

Next we will look in various file formats and provide information on how to import their information in Python.

Plain Text (txt)

As the name suggests, plain text file format contain only plain text. Usually, this text is in unstructured form and there is no metadata associated with it.

Let's take a simple example of a text file. The following example shows text file data that contain text:

```
All models are wrong, but some are useful
```

The above text is written in the attached file called `Sentence.txt` and can be read in various ways one of them being with the code below:

```
text_file = open('Sentence.txt', 'r')
lines = text_file.read()
lines
'All models are wrong, but some are useful'
```

Comma-Separated Values (CSV)

As mentioned in previous notes the comma-separated values (CSV) file format is for cases when data is stored in cells that are organized in rows and columns. Each column can be of different types; strings (text characters), dates, integers, etc. Each line in CSV file represents an observation or commonly called a record. Each record may contain one or more fields which are separated by a comma.

The image below shows the `titanic.csv` file (attached to this notes) when opened with TextEdit on a macOS. This file contains information for each of the passengers of the Titanic.

```
titanic.csv
survived,pclass,sex,age,sibsp,parch,fare,embarked,class,who,adult_male,deck,embark_town,alive,alone,
0,3,male,22,1,0,7.25,S,Third,man,TRUE,,Southampton,no,FALSE,
1,1,female,38,1,0,71.2833,C,First,woman,FALSE,C,Cherbourg,yes,FALSE,
1,3,female,26,0,0,7.925,S,Third,woman,FALSE,,Southampton,yes,TRUE,
1,1,female,35,1,0,53.1,S,First,woman,FALSE,C,Southampton,yes,FALSE,
0,3,male,35,0,0,8.05,S,Third,man,TRUE,,Southampton,no,TRUE,
0,3,male,,0,0,8.4583,Q,Third,man,TRUE,,Queenstown,no,TRUE,
0,1,male,54,0,0,51.8625,S,First,man,TRUE,E,Southampton,no,TRUE,
0,3,male,2,3,1,21.075,S,Third,child,FALSE,,Southampton,no,FALSE,
1,3,female,27,0,2,11.1333,S,Third,woman,FALSE,,Southampton,yes,FALSE,
1,2,female,14,1,0,30.0708,C,Second,child,FALSE,,Cherbourg,yes,FALSE,
1,3,female,4,1,1,16.7,S,Third,child,FALSE,G,Southampton,yes,FALSE,
1,1,female,58,0,0,26.55,S,First,woman,FALSE,C,Southampton,yes,TRUE,
0,3,male,20,0,0,8.05,S,Third,man,TRUE,,Southampton,no,TRUE,
0,3,male,39,1,5,31.275,S,Third,man,TRUE,,Southampton,no,FALSE,
0,3,female,14,0,0,7.8542,S,Third,child,FALSE,,Southampton,no,TRUE,
1,2,female,55,0,0,16,S,Second,woman,FALSE,,Southampton,yes,TRUE,
0,3,male,2,4,1,29.125,Q,Third,child,FALSE,,Queenstown,no,FALSE,
1,2,male,,0,0,13,S,Second,man,TRUE,,Southampton,yes,TRUE,
0,3,female,31,1,0,18,S,Third,woman,FALSE,,Southampton,no,FALSE,
1,3,female,,0,0,7.225,C,Third,woman,FALSE,,Cherbourg,yes,TRUE,
0,2,male,35,0,0,26,S,Second,man,TRUE,,Southampton,no,TRUE,
1,2,male,34,0,0,13,S,Second,man,TRUE,D,Southampton,yes,TRUE,
1,3,female,15,0,0,8.0292,Q,Third,child,FALSE,,Queenstown,yes,TRUE,
1,1,male,28,0,0,35.5,S,First,man,TRUE,A,Southampton,yes,TRUE,
0,3,female,8,3,1,21.075,S,Third,child,FALSE,,Southampton,no,FALSE,
1,3,female,38,1,5,31.3875,S,Third,woman,FALSE,,Southampton,yes,FALSE,
0,3,male,,0,0,7.225,C,Third,man,TRUE,,Cherbourg,no,TRUE,
0,1,male,19,3,2,263,S,First,man,TRUE,C,Southampton,no,FALSE,
1,3,female,,0,0,7.8792,Q,Third,woman,FALSE,,Queenstown,yes,TRUE,
```

A view of a CSV file.

To load data in Python from a CSV file, you can use the **Pandas** package. The code below activates the `pandas` package (first line), reads the data from the attached file `titanic.csv`, stores them in the `pandas` data frame `pd` (second line), which we then view by simply typing its name (third line).

```
import pandas as pd
df = pd.read_csv('titanic.csv')
df
```

The output from the program above is shown below:

| | survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male |
|-----|----------|--------|--------|------|-------|-------|---------|----------|--------|-------|------------|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | man | True |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | woman | False |
| 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | woman | False |
| 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | woman | False |
| 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | man | True |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 886 | 0 | 2 | male | 27.0 | 0 | 0 | 13.0000 | S | Second | man | True |
| 887 | 1 | 1 | female | 19.0 | 0 | 0 | 30.0000 | S | First | woman | False |
| 888 | 0 | 3 | female | NaN | 1 | 2 | 23.4500 | S | Third | woman | False |
| 889 | 1 | 1 | male | 26.0 | 0 | 0 | 30.0000 | C | First | man | True |
| 890 | 0 | 3 | male | 32.0 | 0 | 0 | 7.7500 | Q | Third | man | True |

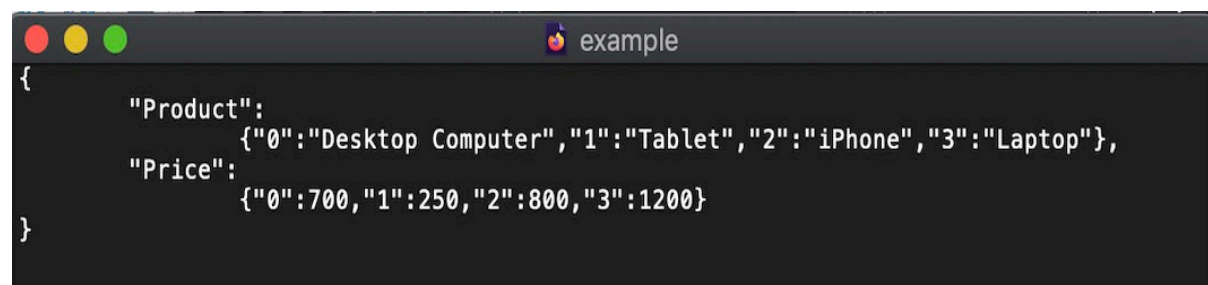
A view of the previous CSV file as a pandas data frame.

To export `pandas` data frames to csv files the following line can be used `df.to_csv(r'Path to store the exported CSV file\File Name.csv')` e.g.

```
df.to_csv(r'Path to store the exported CSV file\export_dataframe.csv', index = False, header=True)
```

JavaScript Object Notation (JSON)

Recall that JavaScript Object Notation (JSON) is a standard format for sending structured data over the web. The JSON file format can be easily read in any programming language because it is language-independent data format. It is a much more flexible data format than a tabular text form like CSV. Here is an example of a JSON file:



```
{
  "Product":
    {"0":"Desktop Computer","1":"Tablet","2":"iPhone","3":"Laptop"},
  "Price":
    {"0":700,"1":250,"2":800,"3":1200}
}
```

A view of a JSON file.

Here is how it can be imported into Python using `pandas`.

```
import pandas as pd
```

```
df = pd.read_json('example.json')
df
```

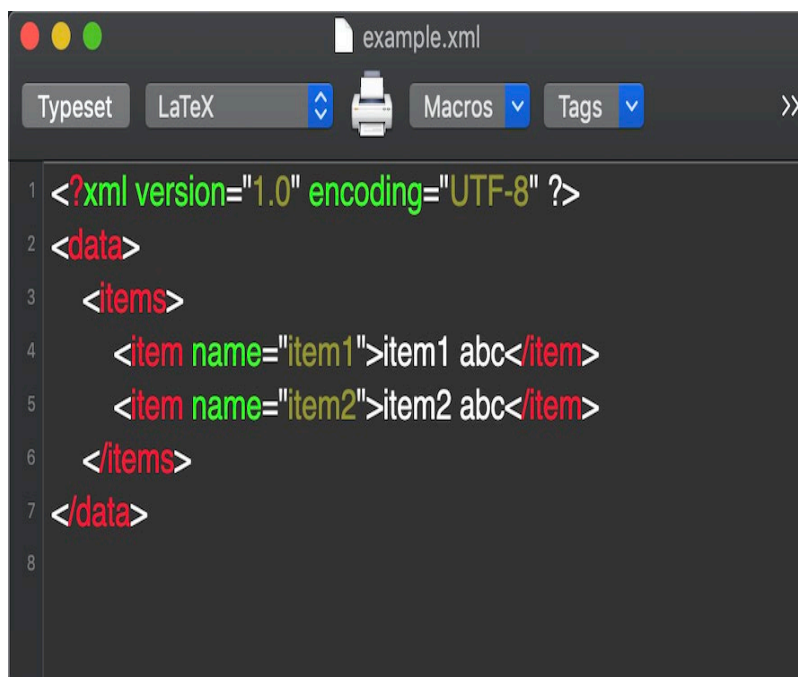
| | Product | Price |
|---|------------------|-------|
| 0 | Desktop Computer | 700 |
| 1 | Tablet | 250 |
| 2 | iPhone | 800 |
| 3 | Laptop | 1200 |

A view of the previous json file as a pandas data frame.

To export `pandas` Data Frames to JSON files the following line can be used `df.to_json(r'Path to store the exported JSON file\File Name.json')`

Extensible Markup Language (XML)

As mentioned in Week 3, Extensible Markup Language (XML) file format is a markup language that has certain rules for encoding data. The following example shows an xml document which is attached as `example.xml`:



A view of a XML file.

To read this file into Python, the `ElementTree` package can be used for example with the code below:

```
import xml.etree.ElementTree as et
tree = et.parse('example.xml')
root = tree.getroot()
```

XML files contain a single root that contains all the data. In the above code the **ElementTree** package is activated (1st line), the file is read and parsed into the object `tree` (2nd line) and the root of the xml file is extracted (3rd line).

The code below illustrates how to read and access the data from the XML file.

```
print('Item #1 attribute: ',root[0][0].attrib) # one specific item attribute
Item #1 attribute:  {'name': 'item1'}

print('\Item #2 data: ', root[0][1].text)# one specific item's data
\Item #2 data:  item2 abc
```

Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 6.
- [Article from Ankit Gupta on most commonly used file formats using Python.](#)
- [Documentation of the ElementTree package:](#)