CHAPTER 2

Introduction to R and accessing Social Web data

1. Algorithms in R

Hello World.

```
Hello <- function() {
    print(”Hello World”)
}
```

```
Hello()
```

```
## [1] ”Hello World”
```

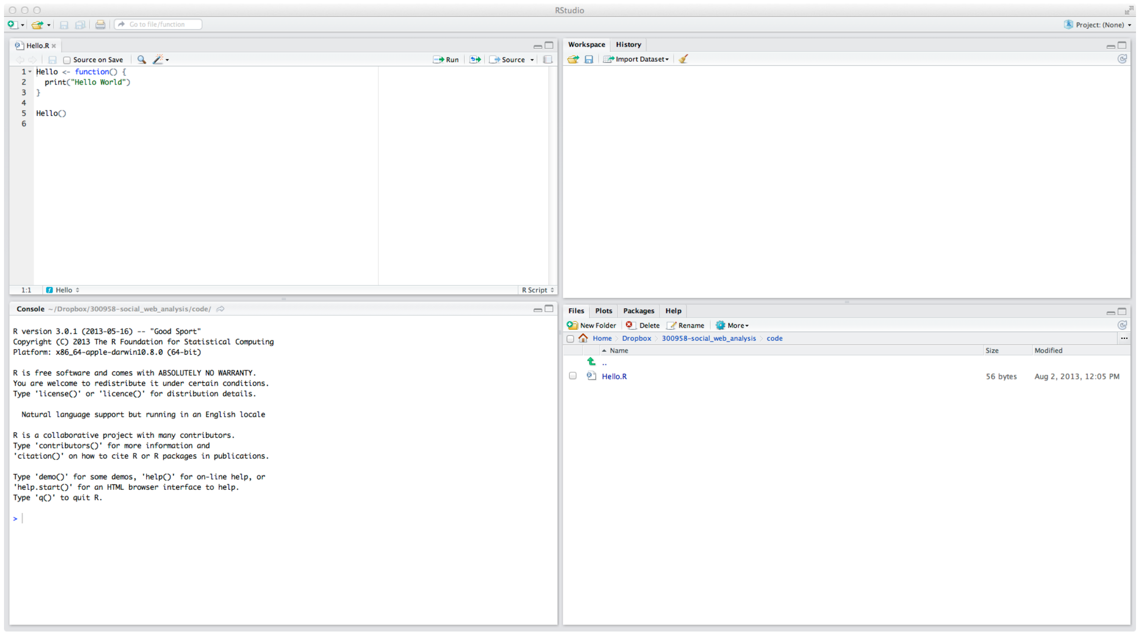
R Scripts. In R commands or programs are stored in text files called “scripts”.

They are usually given the file extension “.r” or “.R”

R itself has a menu item (under “File”) to open a script or create a new one.

Rstudio also does.

R Studio. R studio is an IDE (Integrated Development Environment)



```r
Hello <- function() {
  print("Hello World")
}

Hello()

```

```
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

R Studio.

- The top left pane is an editor for scripts (and other files)
- The bottom left is the R console
- The top right has workspace tab and history tab
  - Workspace is the list of objects created in R
  - History contains the commands you have run
- The bottom right has 4 tabs
  - Files - a listing of the *current working directory*
  - Plots - any plots made in R go here
  - Packages - the available add-on packages
  - Help - help and documentation

R Syntax - Basic data. R has several basic data types

- numbers - 1234, 3.14, 1e5
- strings - “Hello”
- factors - factor(c(“H”,“H”, “T”, “H”), levels=c(“H”,“T”))

R Syntax - Operators. It has operators

- Numeric, + - * / ^
- Comparison, < > <= >= == !=
- Logical, & | && ||
- Assignment <- = ->

Problem. Given two real numbers x and y, what is the R code to test if:

$$\log(x^2 + \pi) \geq y$$

R Syntax - Objects and Expressions. Objects have names made up of letters, numbers, . and _

A letter comes first, case matters

Values can be assigned to them

```
x = 1
y <- ”String”
```

An R expression is combination of objects, operators and data items (possibly with brackets)

```
1 + 1
```

```
## [1] 2
```

```
2 * (1 + x)
```

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

R Syntax - Vectors. Vectors are set of items stored together. They have length and content.

They are constructed with “c”

```
x <- c(1, 3, 7)
length(x)
```

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

The “:” operator can be used to set up a sequence of integers

```
x <- 2:7
x
```

```
## [1] 2 3 4 5 6 7
```

R Syntax - Vectors. “seq” is more general (see help)

```
x <- seq(2, 12, 2)
x
```

```
## [1]  2  4  6  8 10 12
```

Individual entries are accessed with “[” (square brackets) and can be assigned to

```
x[3]
```

```
## [1] 6
```

```
x[4] <- 10
x
```

```
## [1]  2  4  6 10 10 12
```

R Syntax - Vectors.

Problem. How many ways can you think of creating the R vector:

$$[\,0.1\;0.2\;0.3\;0.4\;0.5\;0.6\;0.7\;0.8\;0.9\;1.0\,]$$

R Syntax - Lists. Vectors contain things that are of the same type. Lists are more general.

```
x <- list(1, ”Hello World”, 1:5)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] ”Hello World”
##
## [[3]]
## [1] 1 2 3 4 5
```

R Syntax - Lists. Entries are accessed (and assigned to) with “[[”

```
x[[2]]
```

```
## [1] ”Hello World”
```

```
x[[2]] <- ”Good-bye!”
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] ”Good-bye!”
##
## [[3]]
## [1] 1 2 3 4 5
```

R functions. R has Functions both built in and you can write your own.

Built in functions can be called on objects or vectors

```
sqrt(2)
```

```
## [1] 1.414
```

```
x <- 1:4
sqrt(x)
```

```
## [1] 1.000 1.414 1.732 2.000
```

Note that many functions are *vectorized*

R functions. Functions are called using brackets “()” (no space)

Any arguments are in the brackets, arguments can be named.

```
sum(1:10)  ### Sum of 1 upto 10
```

```
## [1] 55
```

```
sum(x = 1:10)  ### Same
```

```
## [1] 55
```

R functions. To define a new function use the “function” keyword

```
Hello <- function() {
    print(”Hello World”)
}
```

Note this function has no arguments. But we still need the brackets to call it

```
Hello()
```

```
## [1] ”Hello World”
```

R functions.

Problem. Write a function that takes two arguments x and y and returns a vector, containing all of the numbers between x and y, with a 0.1 interval (i.e. returns the set $z = x + 0.1n$ where n is a positive integer and $z \leq y$).

R Control structures. Control structures control how the function proceeds

if statements.

```
if (condition) {
    expression ~ 1
} else {
    expression ~ 2
}
```

Where expression~1 is executed if the condition is TRUE or non-zero and expression~2 is executed otherwise

R Control structures. Loops allow things to be repeated. In R loops run through vectors.

for loops.

```
for (i in vector) {
    expression
}
```

expression is executed with i set to each value in vector in turn

Example 1 - Factorial.

```
Factorial <- function(n) {
  fac <- 1
  for(i in 1:n) {
    fac <- fac * i
  }
  return(fac)
}

Factorial(5)
```

```
## [1] 120
```

Example 2 - Recursive Factorial.

```
RFactorial <- function(n) {
  if(n==1) {
    return(n)
  } else {
    return(n * RFactorial(n - 1))
  }
}

RFactorial(5)
```

```
## [1] 120
```

R Control structures. There are break and next statements:

- break: breaks out of a loop
- next: skips to the next entry

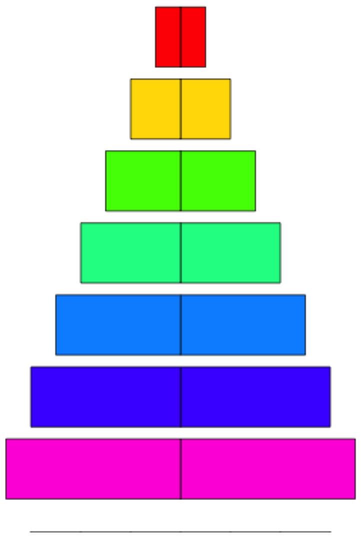
There are also while loops

while loops.

```
while (condition) {
    expression
}
```

Keep executing the expression while the condition is TRUE (or nonzero)

Towers of Hanoi example? Tower of Hanoi



Towers of Hanoi example?

```
move.hanoi <- function(k, from, to, via) {
  if (k > 1) {
    move.hanoi(k - 1, from, via, to)
    move.hanoi(1, from, to, via)
    move.hanoi(k - 1, via, to, from)
  } else {
    cat("Move ", LETTERS[tower[[from]][1]],
        " from ", from,
        " to ", to, "\n")
  }
}

move.hanoi(n, 1, 3, 2)
```

2. Accessing Information from the Social Web

How can my program access the Web? Web pages are designed for reading and easy navigation, and so it is difficult for a program to extract information from them.

Most large sites have a database back end. Our programs would prefer to talk to the database.

In this section, we will investigate how to automatically gather information from Web sites.

Scraping Web pages. If you can view information in a Web page, then you are able to write a program to scrape the information into your database.

Scraping data requires knowledge of the HTML structure of the page.

Example Web page scraping process

(1) Download the HTML from the desired URL
(2) Parse the HTML into some data structure
(3) Extract the portions at the desired positions (e.g. the first paragraph of the second div block).

The Python library BeautifulSoup is an example of software that assists in Web page scraping.

Problems with Web page scraping. This Python program scrapes the date from the Bureau of Meteorology Web site:

```
import urllib2
from BeautifulSoup import BeautifulSoup
bom_url = ”http://www.bom.gov.au/nsw/forecasts/sydney.shtml”

# download the HTML
html = urllib2.urlopen(bom_url).read()
# parse HTML
soup = BeautifulSoup(html)
# find the temperature in the HTML
max_temperature = soup.html.body.findAll(’em’,{”class” : ”max”})[0].string
# print the result
print ”The maximum temperature for today is ” + max_temperature + ” degrees.”
```

The code is dependent on the formatting of the Web page and so will break if the formatting changes.

Parts of a URL.

`scheme://domain:port/path?query_string#fragment_id`

- scheme: http, https, ftp, ftps, ssh, webdav, ...
- domain: either a domain name or IP address
- port: port number, defaults to 80 if not provided
- path: path to file on Web server
- query string: arguments for dynamic files
- fragment_id: tag identifying section of the file

Problem. Identify each part of the URL:

`http://www.google.com/search?client=safari&q=hello`

Accessing an API. An *Application Programming Interface* (API) provides us access to the information in a structured format, allowing us to use the information in other applications.

Web sites that provide an API, do so using a specific URL.

The SCEM seminar site has a two access points:

- The human readable form: `http://seminars.scem.uws.edu.au/detail.php?id=20`
- The computer readable form using the API:
  `http://seminars.scem.uws.edu.au/api.php?id=20`

For an API to be effective, it must provide a consistent URL and data format that allows backward compatibility.

API data formats. An API should present data in a standard format to allow us to use existing tools for parsing.

The main formats used are:

- XML
- YAML
- JSON

Each of these formats allow us to represent complex data structures.

XML. The eXtensible Markup Language (XML) is a flexible language for representing information. It allows us to store data in a human readable form making information more accessible. Many organisations have adopted XML for use in their file formats (the Microsoft format .docx is an XML file compressed using Zip).

XML structure.

Tags.

- Start tag: <custard>
- End tag: </custard>

Elements. An XML element everything from (including) the element's start tag to (including) the element's end tag: <password>This is my password</password>. Note that elements can be nested to create a hierarchy of elements by placing elements within the tags of another element.

Attributes. XML attributes allow us to add information to a tag: <password computer="barney.scem.uws.edu.au">Password for barney</password>

Example XML.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<characters>
    <character id="Frodo">
        <name>Frodo Baggins</name>
        <race>Hobbit</race>
    </character>
    <character id="Aragorn">
        <name>Aragorn, son of Arathorn</name>
        <race>Human</race>
    </character>
</characters>
```

Problem. List all of the tags, elements and attributes of the above XML.

YAML. YAML (YAML Ain't Markup Language) was designed as a data representation language that can store data types from most high level programming languages, and formatted so that it is easily read by humans.

YAML structure.

Tags. A YAML tag is a sequence of characters appearing before the associated data. A tag and its associated data are separated by a colon ':'.

Elements. A YAML element consists of a tag and associated data, all indented at the same level, for example "password: this is my password". Note that elements can be nested to create a hierarchy of elements by placing elements within the data portion of another element.

Lists. A list of elements begin with a dash '-' and all elements are indented at the same level.

Example YAML.

```
%YAML 1.1 # Characters from Middle Earth
---
characters:
    - character:
        id: Frodo
        name: Frodo Baggins
        race: Hobbit
    - character:
        id: Aragorn
        name: Aragorn, son of Arathorn
        race: Human
```

Problem. Identify all of the tags, elements and lists of the above YAML.

JSON. JavaScript Object Notation (JSON) is the format in which data structures are printed from javascript. Since many Web applications are written in javascript, JSON has become a standard for sharing information between Web applications.

JSON structure.

Tags. A JSON tag is similar to a YAML tag, but it is encased in double quotes. E.g. "password".

Elements. JSON elements, like YAML elements, consist of a tag and associated data, but indentation is not needed. Note that elements can be nested to create a hierarchy of elements by placing elements within the data portion of another element.

Lists. A list of elements is encased in square brackets, e.g. ['apple', 'banana', 'carrot']

Example JSON.

```
{
    ”characters”:
    [
        {
            ”id”:”Frodo”,
            ”name”:”Frodo Baggins”,
            ”race”:”Hobbit”
        },
        {
            ”id”:”Aragorn”,
            ”name”:”Aragorn, son of Arathorn”,
            ”race”:”Human”
        }
    ]
}
```

Problem. Identify all of the tags, elements and lists of the above JSON.

Authentication. Web sites offering an API usually require authentication to keep all users accountable. Without authentication, a user could constantly download information from a site, draining the sites resources and limiting its bandwidth, without detection.

If all users are authenticated, then a site can limit the access time or downloads, ensuring all users have a consistent quality of service.

Traditional authentication is to load the login page and provide your username and password.

Insecurity. APIs are for access by programs that perform a specific task (e.g. list today's tweets). A program should not have full access to your account.

OAuth. OAuth is a standard for authorisation, that provides temporary key for restricted access to a service.

To obtain the OAuth key, we must manually access the site and generate a key with given access priviliages (read only, read and write, or write only). This key can then be provided to a program to automatically access our account.

ROAuth. We will be using the R library ROAuth to authenticate users through R.

Facebook Graph API. Facebook offers an interface to test out queries before inserting them in a program: `https://developers.facebook.com/tools/explorer`

To obtain an access token (for OAuth), you must login to a Facebook account.

For more information on the Graph API: `https://developers.facebook.com/docs/reference/api/`

Example queries.

```
https://graph.facebook.com/me?access_token=...
https://graph.facebook.com/me/friends?access_token=...
https://graph.facebook.com/me/mutualfriends/you?access_token=...
```

Note: “me” can be replaced with an ID, “you” must be replaced with an ID.

Twitter API. Unfortunately, there is no Twitter explorer page. To get information from Twitter, we must send a HTTP request containing our ID, the access key and the query. The results are returned in JSON format.

There are many programming languages the provide access to the Twitter API. We will use the R interface throughout this unit.

More information on the Twitter API is here: `https://dev.twitter.com/docs/api/1.1`

Accessing Information. Note that we have access to most of the information from Twitter, but we usually only have access to our information and our friends information from Facebook.

TwitteR. We will be using the R library twitteR to access the Twitter API through R.

Demonstration. We will now show how to set up access to Twitter information using R and the Twitter API.

Steps:

1. Set up a Twitter App
2. Obtain the App OAuth keys
3. Give R the OAuth keys to authorise it to connect to the App
4. Start downloading tweets

Try to do this yourself later. We will require Twitter information in the later lab classes.

Summary.

- We can use R to write programs that interface with the Web.
- Data can be scraped from Web pages or accessed using their API.
- Web APIs provide data in either XML, YAML or JSON format.
- To ensure a specific quality of service, Web sites require authentication when using their API.
- Facebook and Twitter provide powerful APIs.

Next Week. Simple Exposure Analysis