# Computational Skills for Biostatistics I: Lecture 1

Amy Willis, Biostatistics, UW

26 March, 2019

# Why bother?

- As a statistician working in any capacity, you will need to know some programming
- As a MS/PhD student in Biostats, you will do some serious programming!
- Good programming practices will help you with research, collaborating, your job search, and your long-term career... whichever path you choose!

# Welcome!

Biost 561 covers modern/advanced R and other programming skills - It is designed for graduate students in Biostatistics, and is tailored to their statistics and programming background

- If you don't know a lot of base R already, you will learn it today and in Homework 1

# Structure and expectations

- Weekly lectures
- Weekly homeworks
- Weekly office hours: HSB F-657. Mondays TBD
- Office hours by appointment

Public folder of course materials at
`https://github.com/adw96/biostat561`

# Topics

Subject to change

- 4/3 Lecture 1: Intro to version control (i.e. `git`), basic `R` (types, methods), writing loops and functions, LaTex/markdown, RStudio projects
- 4/10 Lecture 2: `dplyr` & `magrittr` (i.e. pipes %>%)
- 4/17 Lecture 3: `ggplot2`, more `dplyr` & `magrittr` practice
- 4/24 Lecture 4: Simulation studies
- 5/1 Lecture 5: fast computation (e.g., apply, `do.call`, mc*apply family), STAN
- 5/8 Lecture 6: Writing `R` packages, namespaces/environments
- 5/15 Lecture 7: knitr, shiny, debugging, profiling
- 5/22 Lecture 8: unix, shell, regex
- 5/29 Lecture 9: cluster computing at UW, computing with AWS *
- 6/5 Lecture 10: C++ and Python in `R` *, recap

* indicates likely guest lecture by one of your classmates. A great opportunity to learn the latest, from the greatest!

# Resources

- Available via github
  `https://github.com/adw96/biostat561`
  - Syllabus
  - Slides (& source code)
  - Examples
  - Homeworks
  - Policies: inclusivity, accessibility, academic integrity
- Available via github classroom
  - Homework submission
- Available via email
  - Announcements

# Expectations

What you should expect of me

- I will make your learning a priority
- I will give you timely feedback on your homeworks
- I will treat you as adults, I will treat you with respect
- I will talk slowly (tell me if I'm speaking too fast!)
- I will try to make class engaging and fun!
- I will teach you the way that I program, or (contemporary) alternatives

# Expectations

What I will expect of you

- You make attending class a priority
- You submit your best work for homework: your own work, on time
- You engage in classroom discussion
- You learn from the class; you learn to teach yourself programming skills
- You treat me, guest lecturers, and each other with respect

# Assessment

The only assessment in this course is homework.

- 10 or fewer homeworks
- You must submit a good attempt at every homework to receive credit for this course

I won't record attendance, but if you consistently do not show up you will not receive credit.

# About me

**AmyW**
adw96

Block or report user

statistician, biodiversity enthusiast, assistant professor

University of Washington
Seattle, WA
statisticaldiversitylab.com

**Overview** Repositories 24 Projects 0 Stars 24 Followers 87 Following 38

Pinned

**breakaway**
Species richness with high diversity
R ★18 8

**CatchAll**
Species richness estimation in R
R ★1 1

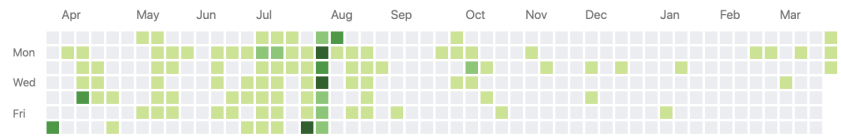
**TreeUncertainty**
Incorporating uncertainty in tree space
R ★4

**biostat561**
Course materials for BIOST561
★97 24

**DivNet**
diversity estimation under ecological networks
R ★13 4

**stamps2018**
Amy's teaching materials for STAMPS @ MBL in 2018
R ★5 2

358 contributions in the last year




Learn how we count contributions.
Less More

# About you

Everyone is here with different backgrounds in programming and computing. Let's get statistical!

?? responses to class survey

- ?? Mac-Windows users
- ??% have used R
- ??% use the apply() family
- ??% pipe, write packages
- ??% use git for **version control**

Is data missing at random?

# Today's class

1. Version control with git
2. Intro to base `R` (types, methods)
3. RStudio & RStudio projects
4. Writing loops and functions
5. LaTex, RMarkdown and knitr

# Version control

What problems can you see with the following approach to version control?

- papersims-v1.R
- papersims-v2.R
- papersims-thea-comments.R
- papersims-hellfire.R
- papersims-v5.R
- papersims-final.R

# Version control

1. How many versions until this becomes intractable?
2. *Date Modified* sorting does not always help
3. Tracking the changes is very difficult; what happens if you need to revert?
4. Exponential file number growth with multiple collaborators!
5. The dreaded computer crash

Dropbox can help with some of these issues, but generally not (3) or (4)!

# Git

- git is an open source version control system (VCS):
  - Track changes to code and documents. *What* changes by *who* and *when*?
  - Share code and collaborate
- github is a website that uses git's VCS:
  - Collaborate with others effectively
  - Distribute code
  - Solicit improvements (*pull* requests)
  - Track issues & feature requests
  - (Build your coding portfolio!)

# Git with R

Git & github are popular with R developers

- Integration with RStudio
- Easy distribution of packages
  - Circumvents CRAN moderators; for better or worse
- Always get the latest features (`devtools`; `install_github`) (in addition to all the other great things about good version control)

# Getting started with git

- Download/update: `https://git-scm.com/downloads`
- Intro:
  `https://guides.github.com/activities/hello-world/`
  - Do this with Homework 1 and a blank pdf, not hello-world
- With RStudio: File/New Project
- Questions? Error messages? The internet is a great resource!
- A great habit to get into early!

Homework 1 Question 0 will get you started using git. More later in the semester!

# R: Class

There are many different *classes* of objects in R

```
x <- c(1, 2, 5)
y <- c("a", "b")
z <- as.factor(y)
c(class(x), class(y), class(z), class(c))
```

```
## [1] "numeric"   "character" "factor"    "function"
```

Others include `logical` (`TRUE`, `FALSE`), complex numbers...

# Modes

R has different modes. The mode tells the way a variable is stored.

```
mode(x)
```

```
## [1] "numeric"
```

```
mode(y)
```

```
## [1] "character"
```

```
mode(z) # factors are stored as numerics
```

```
## [1] "numeric"
```

```
mode(c)
```

```
## [1] "function"
```

# Modes

`is.[class]` asks about the class. Normally you will be interested in the class, not the mode.

```
is.numeric(x)
```

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

```
is.factor(z)
```

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

```
is.numeric(z) # we asked about class, not the mode
```

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

# Data structures

R can store data in various *objects*

- vector: one-dimensional, all data points have same mode
- matrix: two-dimensional, all data points have same mode
- data frame: two-dimensional, all data points in same column have same mode
- list: one-dimensional, data points can be of any type

# Matrices

Matrices vs data frames: all elements have same mode in matrices

```
cbind(c(1,2), c("a", "b"))
```

```
##      [,1] [,2]
## [1,] "1"  "a"
## [2,] "2"  "b"
```

# Matrices

```
aa <- matrix(c(1, 2, 3, 5), nrow = 2, byrow = T)
bb <- c(0.5, 2)
aa
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    5
```

```
bb
```

```
## [1] 0.5 2.0
```

# Matrices

Be very careful with matrix operations!

```
aa %*% bb # matrix multiplication
```

```
##      [,1]
## [1,]  4.5
## [2,] 11.5
```

```
aa * bb # careful! pointwise
```

```
##      [,1] [,2]
## [1,]  0.5    1
## [2,]  6.0   10
```

# Data frames

```r
data.frame(c(1,2), c("a", "b"))
```

```
##   c.1..2. c..a....b..
## 1       1           a
## 2       2           b
```

```r
dd <- data.frame("ID"=c(1,2), "name"=c("a", "b")) # better
dd
```

```
##   ID name
## 1  1    a
## 2  2    b
```

```r
str(dd) # structure: compact info about frame & variables
```

```
## 'data.frame':    2 obs. of  2 variables:
##  $ ID  : num  1 2
##  $ name: Factor w/ 2 levels "a","b": 1 2
```

# Vectorization

Vectorization: doing many calculations with a single command

```
x <- c(0.5, 2, 3, 6)
x^2
```

```
## [1]  0.25  4.00  9.00 36.00
```

```
y <- c(3, 1, 2, 1)
x^y
```

```
## [1] 0.125 2.000 9.000 6.000
```

# Vectorization

The slow way: with loops. Avoid where possible!

```
z <- rep(NA, 4)
for (i in 1:4) {
  z[i] <- x[i]^y[i]
}
z
```

Code is easier to read, and usually easier to debug, when vectorised

# Recycling

In many tasks, R recycles elements of one input until it has enough to match the other

```
aa
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    5
```

```
bb
```

```
## [1] 0.5 2.0
```

```
aa + bb # vectors are treated as columns!
```

```
##      [,1] [,2]
## [1,]  1.5  2.5
## [2,]  5.0  7.0
```

```
cc <- c(1, 2, 3, 4)
aa + cc # the silent killer
```

# Speed comparison

vectorization can cause major speed-ups, because task is optimised and precompiled in C/Fortran, not interpreted R.

```
dd <- matrix(rnorm(1e6), nrow = 1000)
cor(dd)

ee <- matrix(NA, 1000, 1000)
for (i in 1:1000) {
  for (j in 1:1000) {
    ee[i, j] <- cor(ee[, i], ee[, j])
  }
}
```

Speed up factor of vectorization: 36!

# Lists

Lists store information of many different types. Names are optional, but recommended!

```
amy <- list(office.num = 657, pets = TRUE,
            pets.names = c("Princess Jaws", "Friendly", "Mohawk",
                           "Canada", "USA", "Regina George"),
            is.cat = c(TRUE, rep(FALSE, 5)))
amy
```

```
## $office.num
## [1] 657
##
## $pets
## [1] TRUE
##
## $pets.names
## [1] "Princess Jaws" "Friendly"      "Mohawk"        "Canada"
## [5] "USA"           "Regina George"
##
## $is.cat
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

# Lists

Double square brackets pull out individual elements. Single square brackets pull out subsets of the list. I recommend using names wherever possible!

```
amy[[3]] # subset third element
```

```
## [1] "Princess Jaws" "Friendly"      "Mohawk"        "Canada"
## [5] "USA"           "Regina George"
```

```
amy[3] # third element -- a list!
```

```
## $pets.names
## [1] "Princess Jaws" "Friendly"      "Mohawk"        "Canada"
## [5] "USA"           "Regina George"
```

# Lists

```
amy[2:3] # second and third elements -- a list!
```

```
## $pets
## [1] TRUE
##
## $pets.names
## [1] "Princess Jaws" "Friendly"      "Mohawk"        "Canada"
## [5] "USA"           "Regina George"
```

```
amy$office # can also refer by name
```

```
## [1] 657
```

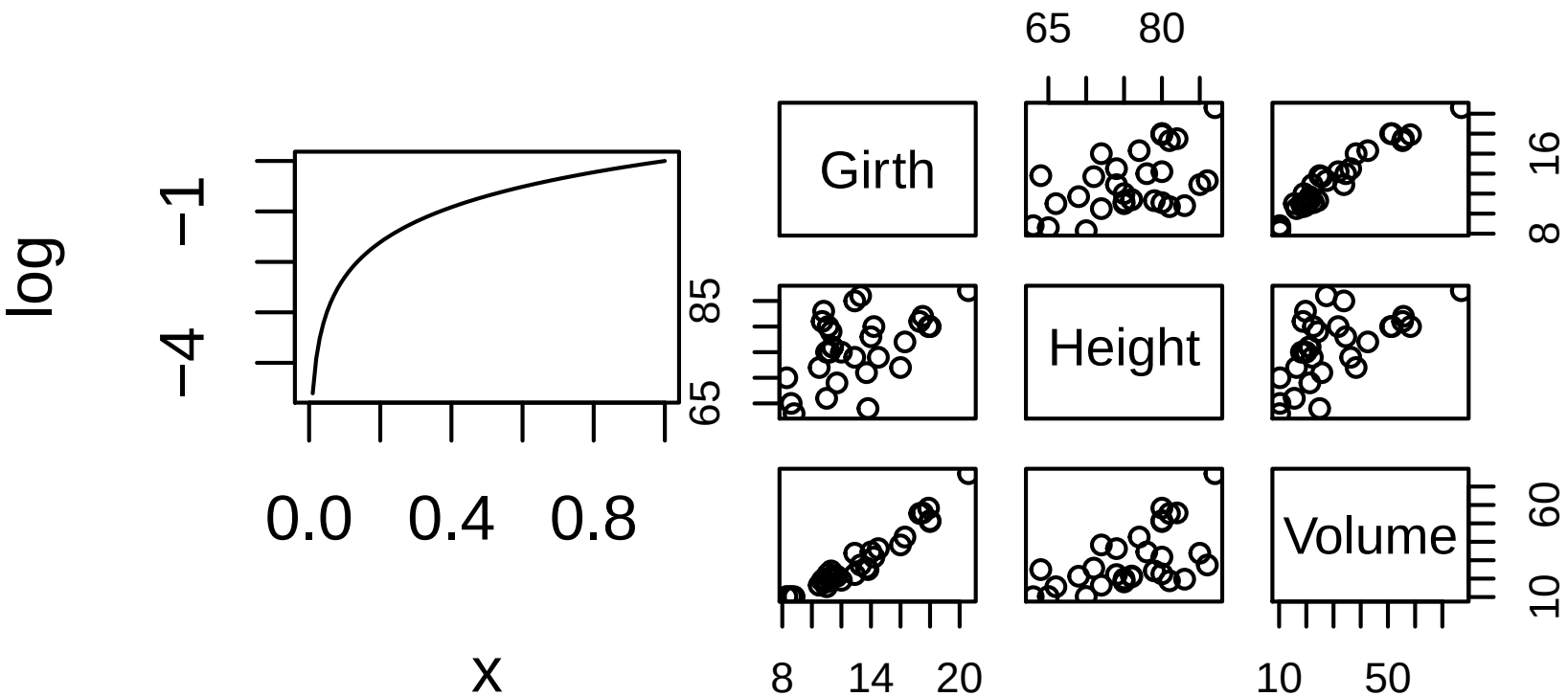
# Generic functions

The same function can apply to objects of different classes. How does R know what to do?

```
c(class(log), class(trees))
```

```
## [1] "function"   "data.frame"
```

```
layout(t(1:2), widths = c(3,1)); plot(log); plot(trees)
```



# Generic functions

`plot` is a *generic* function. Generic functions don't do anything themselves – they call *methods*, which are tailored to the class.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fa9ed18a9b8>
## <environment: namespace:graphics>
```

```
methods("plot") # lists all types R knows how to plot
```

```
##  [1] plot.acf*              plot.data.frame*    plot.decomp
##  [4] plot.default           plot.dendrogram*    plot.densit
##  [7] plot.ecdf              plot.factor*        plot.formul
## [10] plot.function          plot.hclust*        plot.histog
## [13] plot.HoltWinters*      plot.isoreg*        plot.lm*
## [16] plot.medpolish*        plot.mlm*           plot.ppr*
```

# Generic functions

To find the functions that apply to a class

```
methods(class = "lm")
```

```
##  [1] add1             alias            anova            case.n
##  [5] coerce           confint          cooks.distance   devian
##  [9] dfbeta           dfbetas          drop1            dummy.
## [13] effects          extractAIC       family           formul
## [17] hatvalues        influence        initialize       kappa
## [21] labels           logLik           model.frame      model.
## [25] nobs             plot             predict          print
## [29] proj             qr               residuals        rstand
## [33] rstudent         show             simulate         slotsF
## [37] summary          variable.names   vcov
## see '?methods' for accessing help and source code
```

# Generic functions

To see the code for a generic function, type `[function].[class]`

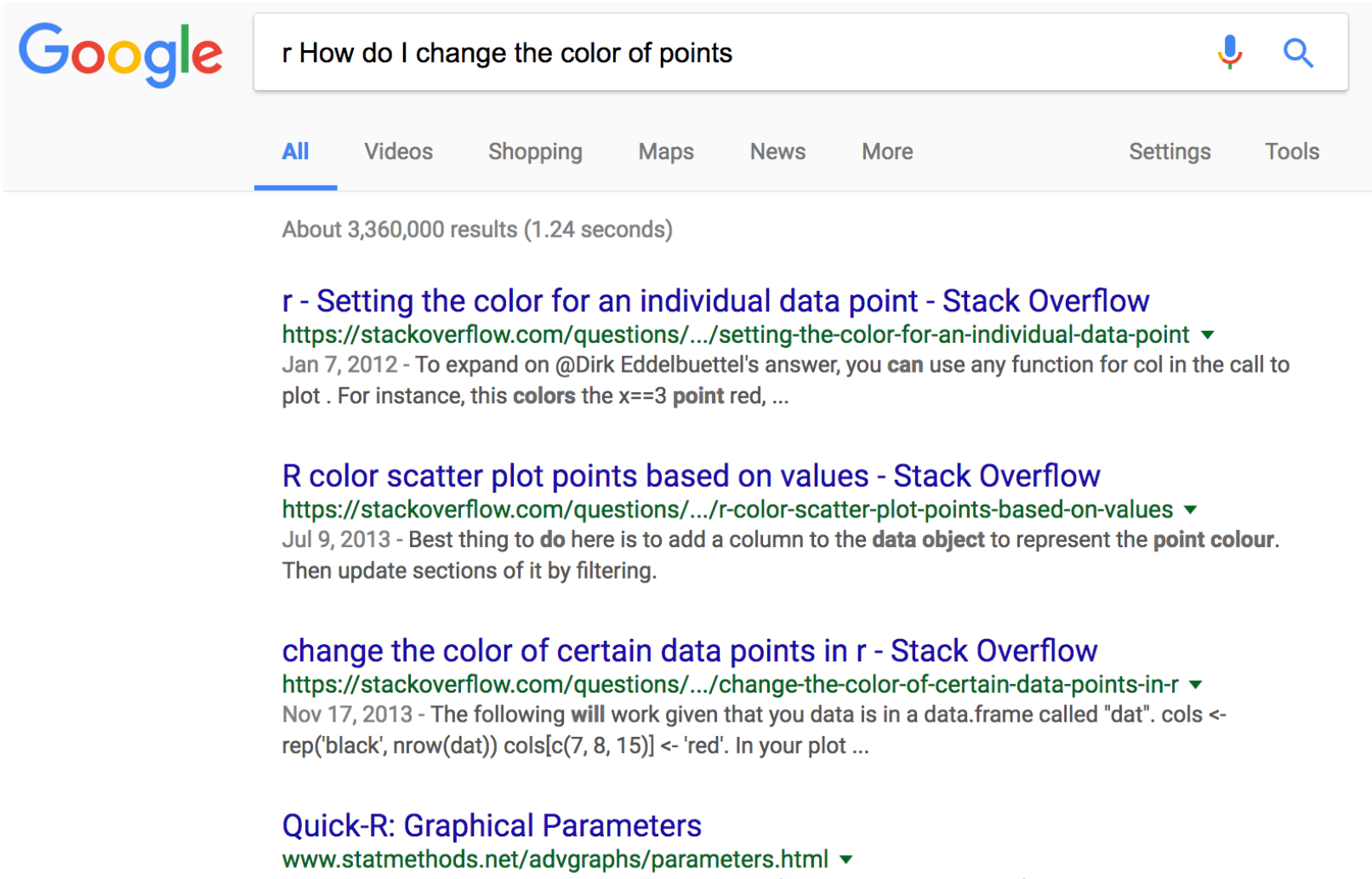
```
dimnames.data.frame
```

```
## function (x)
## list(row.names(x), names(x))
## <bytecode: 0x7fa9ed97b740>
## <environment: namespace:base>
```

# Help

There are many ways to get help with using functions or debugging code

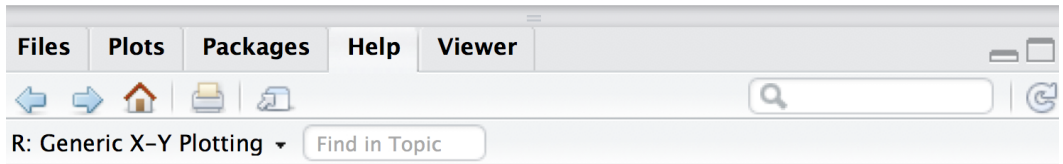
1. The internet



# Help

2. `?fn` shows the documentation for `fn`...

```
> ?plot
>
```




plot {graphics} R Documentation

## Generic X-Y Plotting

### Description

Generic function for plotting of `R` objects. For more details about the graphical parameter arguments, see `par`.

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many `R` objects, including `function`s, `data.frame`s, `density` objects, etc. Use `methods(plot)` and the documentation for these.

### Usage

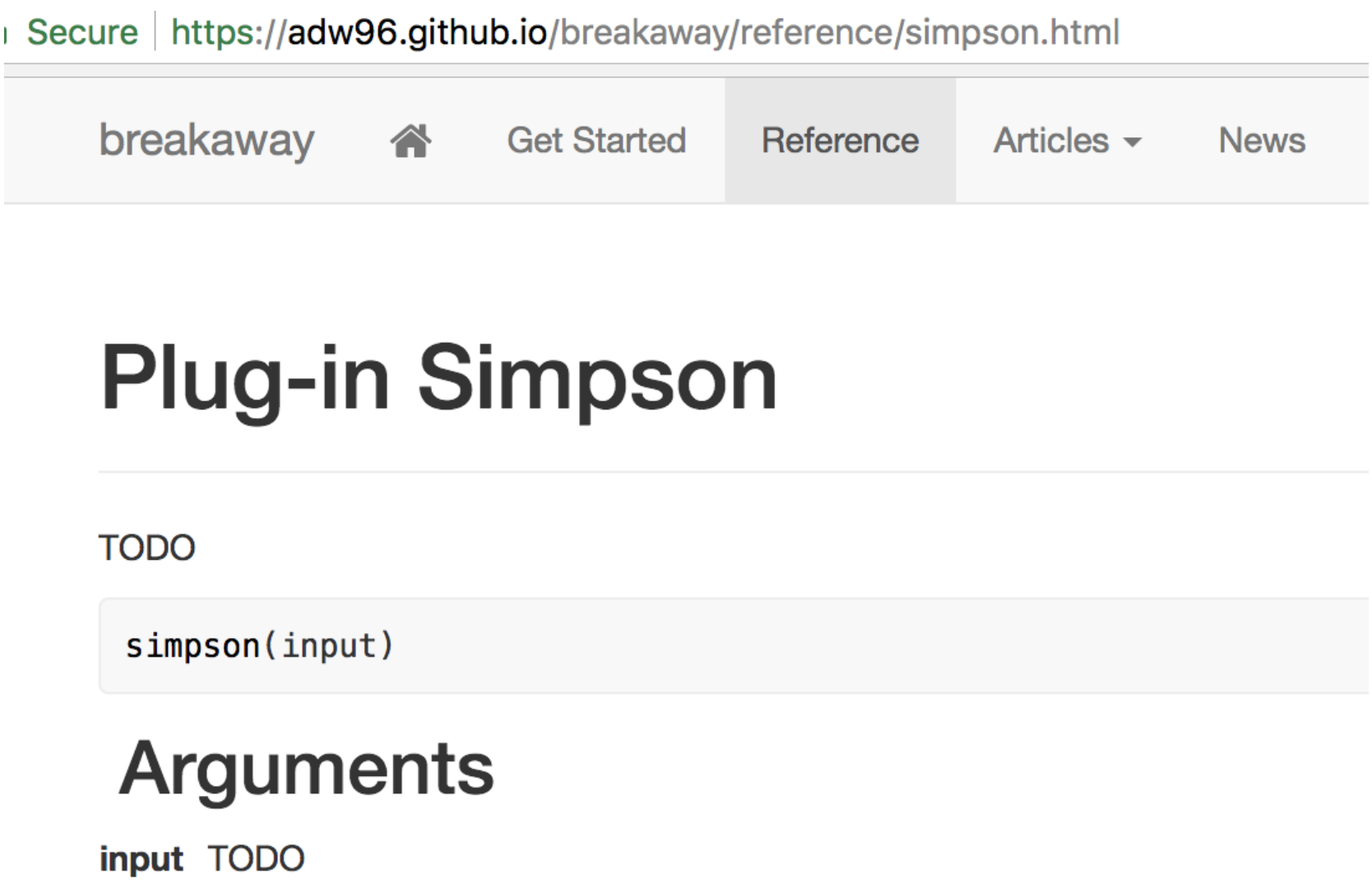
```
plot(x, y, ...)
```

### Arguments

| | |
|---|---|
| `x` | the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any `R` object with a `plot` method* can be provided. |
| `y` | the y coordinates of points in the plot, *optional* if `x` is an appropriate structure. |
| `...` | Arguments to be passed to methods, such as graphical parameters (see `par`). Many methods will accept the following arguments: |

# Help

2. . . . if it exists!



# Help

3. `help.search("topic")` searches help pages for "topic"

```
> help.search("points")
>
```


## Search Results

**Code demonstrations:**

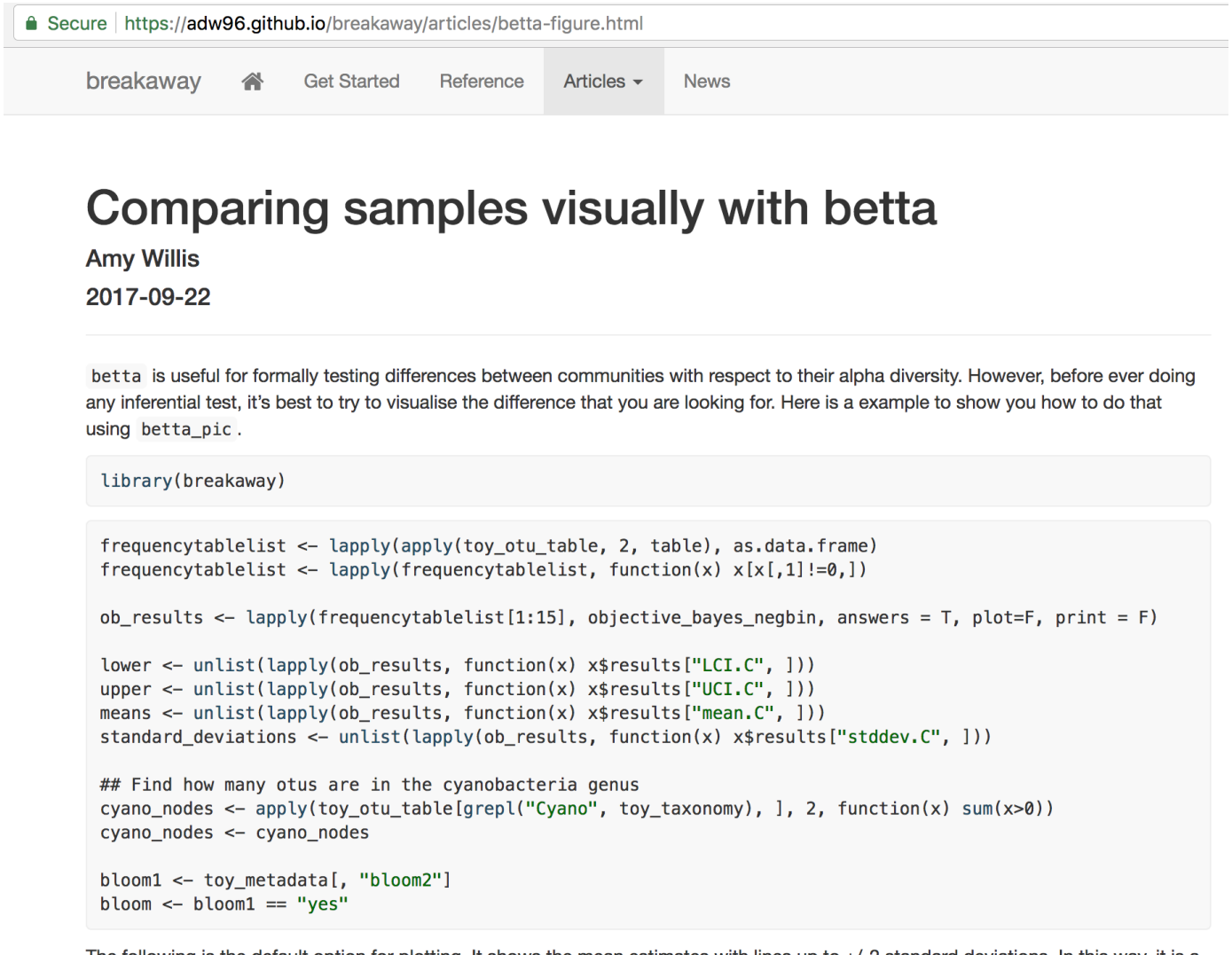
| | | |
|---|---|---|
| tcltk::tkcanvas | Creates a canvas widget showing a 2-D plot with data points that can be dragged with the mouse. | (Run demo in console) |

**Help pages:**

| | |
|---|---|
| ade4::ichtyo | Point sampling of fish community |
| ade4::procuste | Simple Procruste Rotation between two sets of points |
| ade4::s.class | Plot of factorial maps with representation of point classes |
| ade4::triangle.class | Triangular Representation and Groups of points |
| ade4::triangle.plot | Triangular Plotting |
| base::pretty | Pretty Breakpoints |
| base::utf8ToInt | Convert Integer Vectors to or from UTF-8-encoded |

# Examples

The documentation pages often show examples (`example(plot)`) and have demos (`demo(plotmath)`). `vignette()` opens longer worked examples that are great for playing with new packages.



# Keep in mind

- The user of a function assumes responsibility for giving arguments in the correct form
- arguments are ordered
  - Unnamed arguments are allocated as first arguments
  - Named arguments can be anywhere in ordering
- Not supplied arguments assume default value
  - Not supplying arguments without a default gives an error message

Don't get bogged down in reading *all* the documentation – experiment and learn from your mistakes instead!

# Debugging

1. Stare at it until you identify the problem a.k.a. psychic debugging
2. Breakdown the components until you find the problem (bisection method converges linearly!)
3. `traceback()` – covered later in the course

# Homework 1 and next week

- Slides:
  - `https://github.com/adw96/biostat561/lecture1/lecture1.pdf`
- Homework 1 is due next Wednesday at 2:30 p.m.
  - `https://github.com/adw96/biostat561/lecture1/homework1.pdf`
- Complete Question 0 by next Tuesday (Office hours!)
- Submission via github classroom (instructions included):
  - `https://classroom.github.com/classrooms/32249780-biost-561`