Programing fundamentals

- A program is a set of instructions for a computer to follow
- Programs are often used to manipulate data (in all type and formats you discussed last week)
- Simple to complex
    - the scripts you wrote last week (simple)
    - instructions to analyze relationships in census data and visualize them
    - a model of global climate

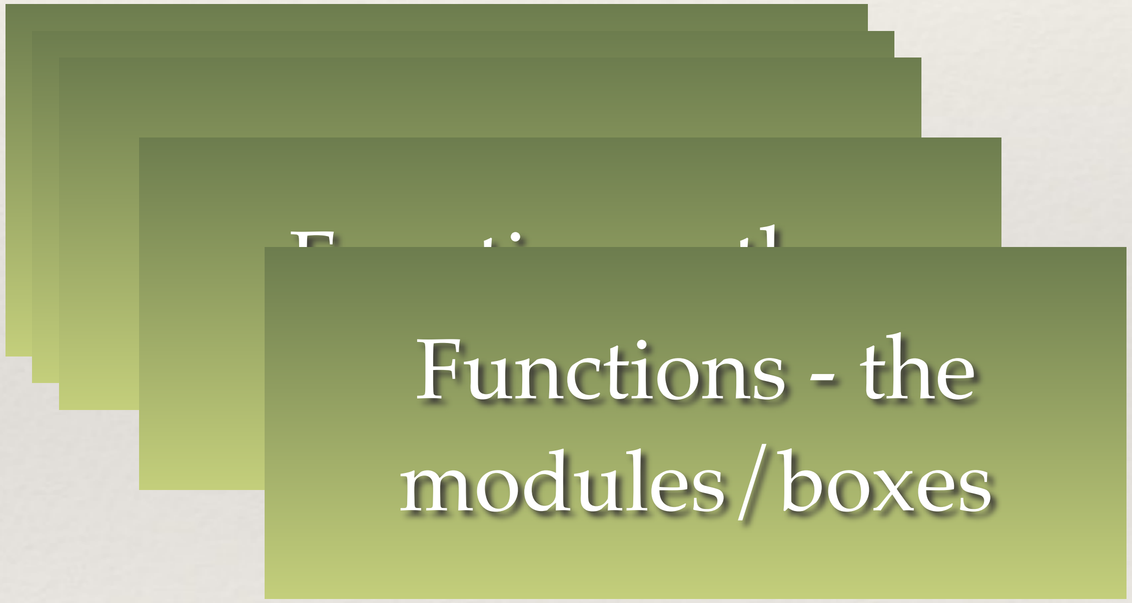
Programing fundamentals

- Operations (=,+,-,…concatenate, copy)
- Data structures (simple variables, arrays, lists…)
- Control structures (if then, loops)
- Modules…

Concepts common to all languages through the syntax may be different

Modularity

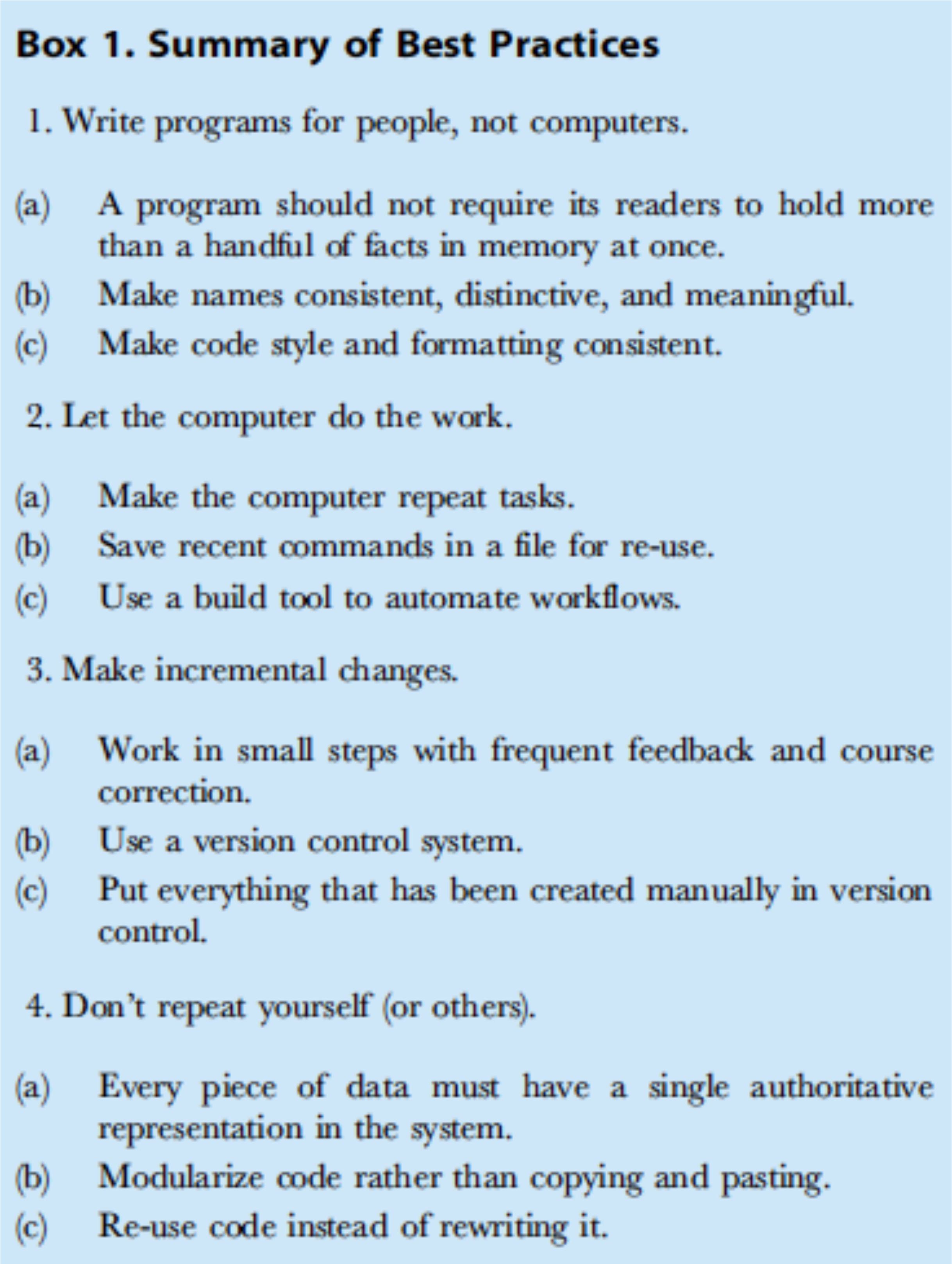
Main controls the overall flow of program- calls to the functions/ modules/building blocks



Functions - the modules/boxes

- A program is often multiple pieces put together
- These pieces or modules can be used multiple times

Programing fundamentals



Box 1. Summary of Best Practices

1. Write programs for people, not computers.

(a) A program should not require its readers to hold more than a handful of facts in memory at once.
(b) Make names consistent, distinctive, and meaningful.
(c) Make code style and formatting consistent.

2. Let the computer do the work.

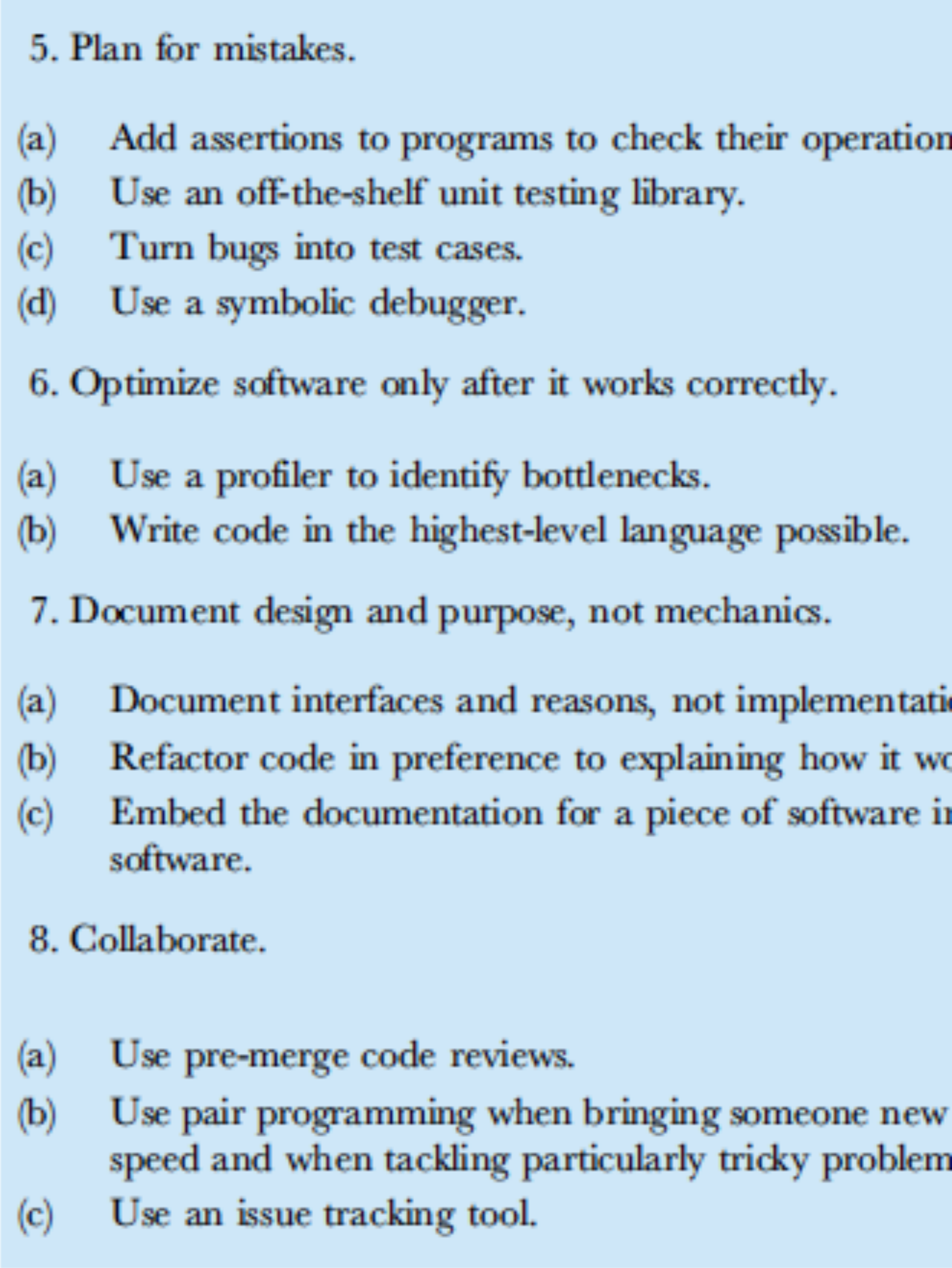
(a) Make the computer repeat tasks.
(b) Save recent commands in a file for re-use.
(c) Use a build tool to automate workflows.

3. Make incremental changes.

(a) Work in small steps with frequent feedback and course correction.
(b) Use a version control system.
(c) Put everything that has been created manually in version control.

4. Don't repeat yourself (or others).

(a) Every piece of data must have a single authoritative representation in the system.
(b) Modularize code rather than copying and pasting.
(c) Re-use code instead of rewriting it.



5. Plan for mistakes.

(a) Add assertions to programs to check their operation
(b) Use an off-the-shelf unit testing library.
(c) Turn bugs into test cases.
(d) Use a symbolic debugger.

6. Optimize software only after it works correctly.

(a) Use a profiler to identify bottlenecks.
(b) Write code in the highest-level language possible.

7. Document design and purpose, not mechanics.

(a) Document interfaces and reasons, not implementati
(b) Refactor code in preference to explaining how it wo
(c) Embed the documentation for a piece of software i software.

8. Collaborate.

(a) Use pre-merge code reviews.
(b) Use pair programming when bringing someone new speed and when tackling particularly tricky problem
(c) Use an issue tracking tool.

Best practices for software development

- Read: Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745
- Blanton, B and Lenhardt, C 2014. A Scientist's Perspective on Sustainable Scientific Software. Journal of Open Research Software 2(1):e17, DOI: http://dx.doi.org/10.5334/jors.ba
- but also
- http://simpleprogrammer.com/2013/02/17/principles-are-timeless-best-practices-are-fads/

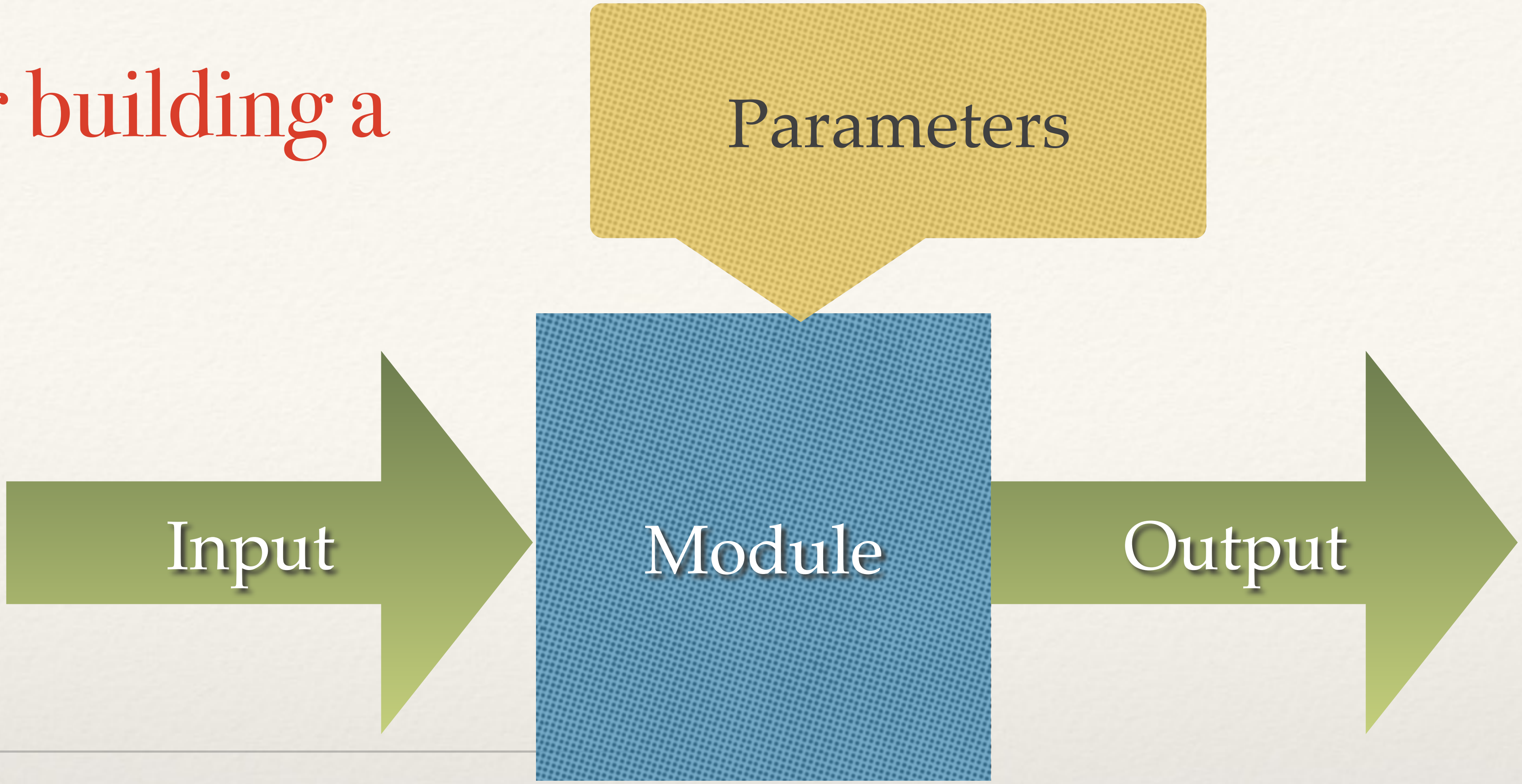
Best practices for model (software) development

- Common problems
    - Unreadable code (hard to understand, easy to forget how it works, hard to find errors, hard to expand)
    - Overly complex, disorganized code (hard to find errors; hard to modify-expand)
    - Insufficient testing (both during development and after)
    - Not tracking code changes (multiple versions, which is correct?)

STEPS: Program Design

1. **Clearly define your goal as precisely as possible, what do you want your program to do**
    1. **inputs/parameters**
    2. **outputs**
2. Implement and document
3. Test
4. Refine

Steps for building a module



Parameters

Input

Module

Output

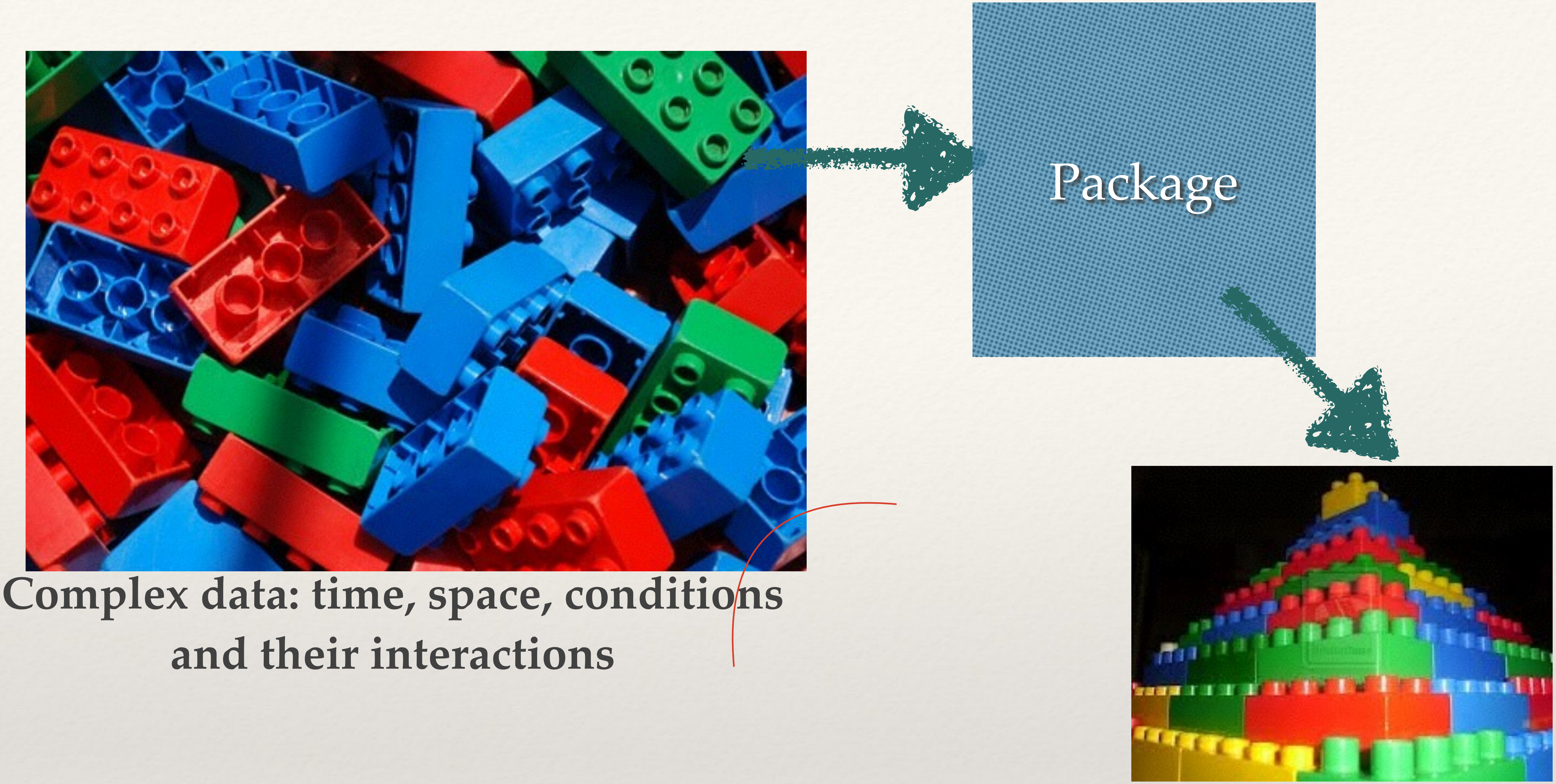
1. Design the program “conceptually” - “on paper” in words or figures
2. Translate into a step by step representation
3. Choose programming language
4. Define inputs (data type, units)
5. Define output (data type, units)
6. Define structure
7. Write program
8. Document the program
9. Test the program
10. Refine…

Best practices for software development

- Automated tools (useful for more complex code development
- (note that GP's often create programs > 100 lines of code)
- Automated documentation
  - http://www.stack.nl/~dimitri/doxygen/
  - http://roxygen.org/roxygen2-manual.pdf
- Automated test case development
  - http://r-pkgs.had.co.nz/tests.html
- Automated code evolution tracking (Version Control)
  - https://github.com/

Designing Programs

- What’s in the box (the program itself) that gives you a relationship between outputs and inputs
  - the link between inputs and output
  - breaks this down into bite-sized steps or calls to other boxes)
  - think of programs as made up building blocks
  - the design of this set of sets should be easy to follow



Package

Complex data: time, space, conditions and their interactions

Building Packages

We often have a project that has a set of different functions and data sets - we can combine these together as a package

Functions in R

- Format for a basic function in R

```
#’ documentation that describes inputs, outputs and what the function does
FUNCTION NAME = function(inputs, parameters) {
body of the function (manipulation of inputs)
return(values to return)
}
```

In R, inputs and parameters are treated the same; but it is useful to think about them separately in designing the model - collectively they are sometimes referred to as arguments

ALWAYS USE Meaningful names for your function, its parameters and variables calculated within the function

Use Lists to return more complex info

```r
#' Summary information about spring climate
#'
#' computes summary information about spring temperature and precipitation
#' @param clim.data  data frame with columns tmax, tmin (C)
#' rain (precip in mm), year, month (integer), day
#' @param months (as integer) to include in spring; default 4,5,6
#' @return returns a list containing, mean spring temperature (mean.springT, (C))
#' year with lowest spring temperature (coldest.spring (year))
#' mean spring precipitation (mean.springP (mm))
#' spring (as year) with highest precip (wettest.spring (year))


spring.summary = function(clim.data, spring.months = c(4:6)) {

  spring = subset(clim.data, clim.data$month %in% spring.months)
  springT = (spring$tmax+spring$tmin)/2.0
  all.springT = aggregate(springT, by =list(spring$year), mean)
  mean.springT = mean(c(spring$tmax, spring$tmin))
  lowyear = spring$year[which.min(spring$tmin)]
  spring.precip = as.data.frame(matrix(nrow=unique(spring$year), ncol=2))
  colnames(spring.precip)=c("precip","year")

  spring.precip = aggregate(spring$rain, by=list(spring$year), sum)


  colnames(spring.precip) = c("year","precip")
  mean.spring.precip = mean(spring.precip$precip)
  wettest.spring = spring.precip$year[which.max(spring.precip$precip)]

  return(list(mean.springT = mean.springT, coldest.spring=lowyear,
              mean.springP=mean.spring.precip,wettest.spring=wettest.spring,
      all.springP = spring.precip, all.springT = all.springT ))
}
```

Packages

- Packages in R are ways to organize code/data
- We've used many packages (e.g dplyr) that contain different functions (e.g manipulate())
- You can create your own package to organize code that you might use for a particular project
  - sharing
  - standardization

Packages

- Packages have a precise directory structure to store your code, data, documentation and tests that is easy for R to read
- A file named DESCRIPTION with descriptions of the package, author, and license conditions - meta data
- in a structured text format that is readable by computers and by people.
- • A man/ subdirectory of documentation files.
- • An R/ subdirectory of R code.
- • A data/ subdirectory of datasets.

- There can be other components but this is a start

Packages

- This package (“classexamples”) is now a directory structure to store your code, data, documentation and tests that is easy for R to read
- A file named DESCRIPTION with descriptions of the package, author, and license conditions
- in a structured text format that is readable by computers and by people.
- • A man/ subdirectory of documentation files.
- • An R/ subdirectory of R code.
- • A data/ subdirectory of datasets.

- There can be other components but this is a start

Packages

- To create a package - in R studio -
  - start a new project
  - create R package
  - at creation you can add things (.R code, .RData data)
  - notice how it creates a project, and subdirectories - any .R files you created will go in R directory

Packages

- use load_all() to load everything in your package into your current workspace
- DESCRIPTION
  - edit this file to describe your function

Packages

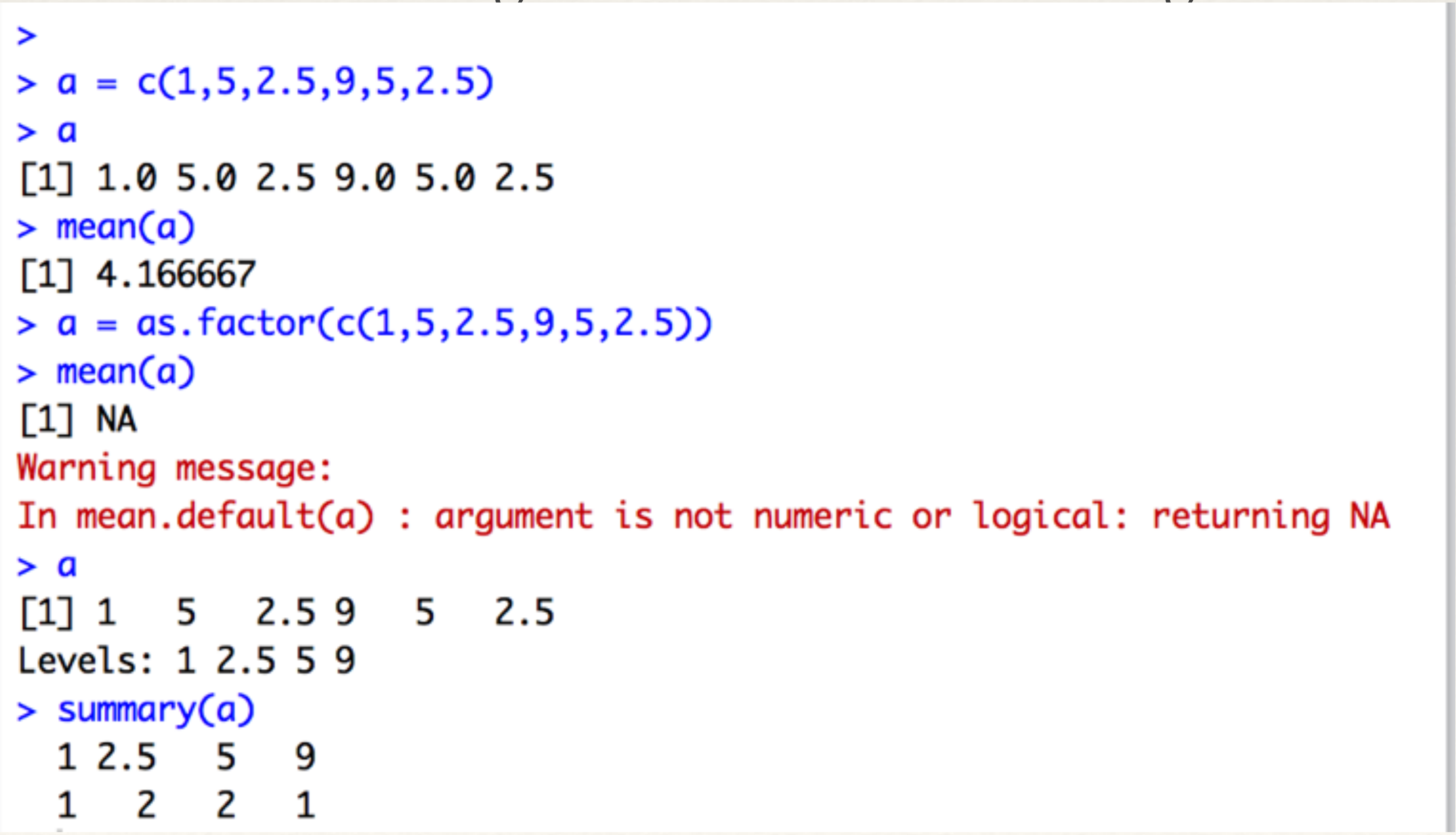
- Data
  - to add data to your package; store as an .RData file in the Data subdirectory
    - use save(name, file=“data/name.RData”)
    - you may have to create data
- R
  - to add code to your package; store as a .R file in the R subdirectory
- See example in esm237examples

Data Structures

- **vectors (c)**
- **matrices, arrays**
- **data frames**
- lists
- factors

Key Programming concepts: Review of data types

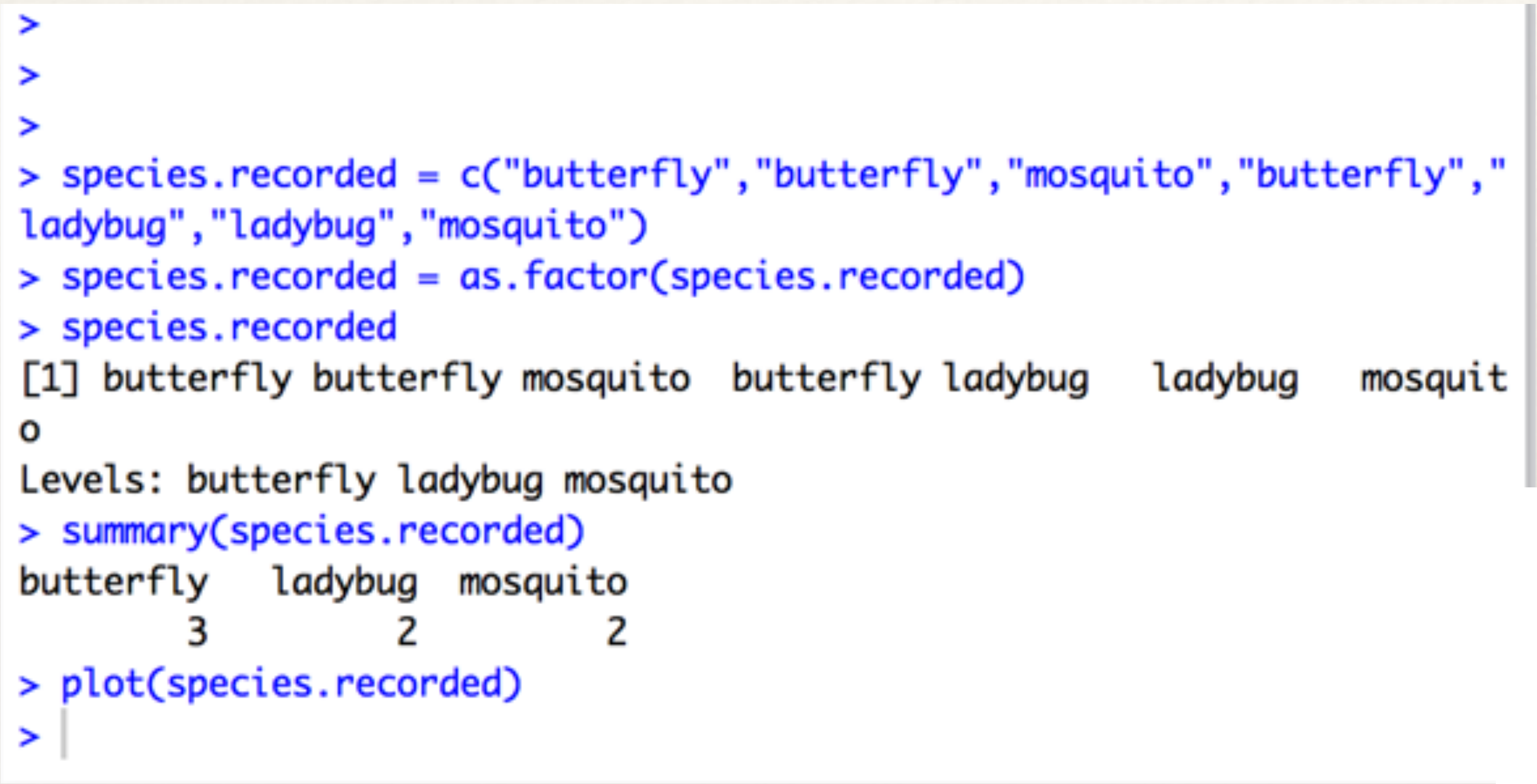
- Factors (a bit tricky, basically a vector of "things" that has different levels (classes); not really numeric - so you can't average them!)
- But can be useful for doing "calculations" with categories



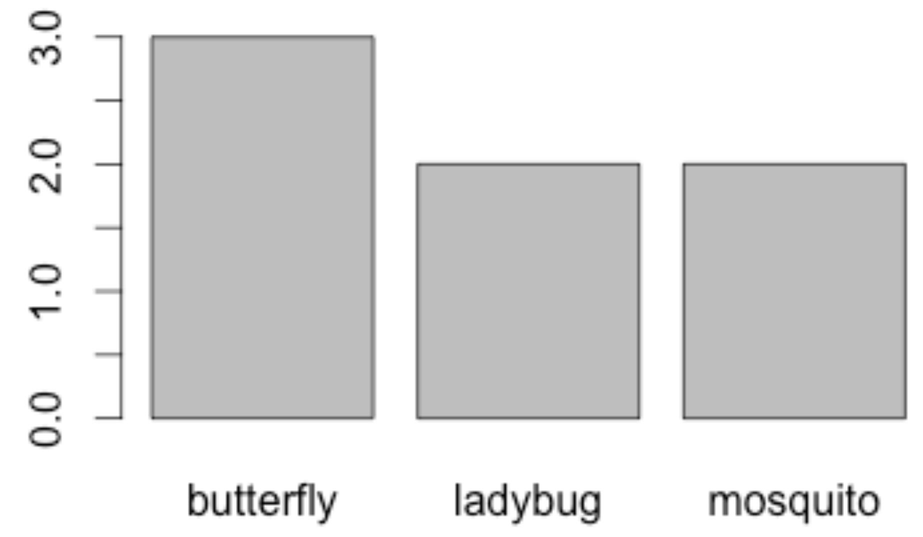
```
>
> a = c(1,5,2.5,9,5,2.5)
> a
[1] 1.0 5.0 2.5 9.0 5.0 2.5
> mean(a)
[1] 4.166667
> a = as.factor(c(1,5,2.5,9,5,2.5))
> mean(a)
[1] NA
Warning message:
In mean.default(a) : argument is not numeric or logical: returning NA
> a
[1] 1   5   2.5 9   5   2.5
Levels: 1 2.5 5 9
> summary(a)
  1 2.5   5   9
  1   2   2   1
```

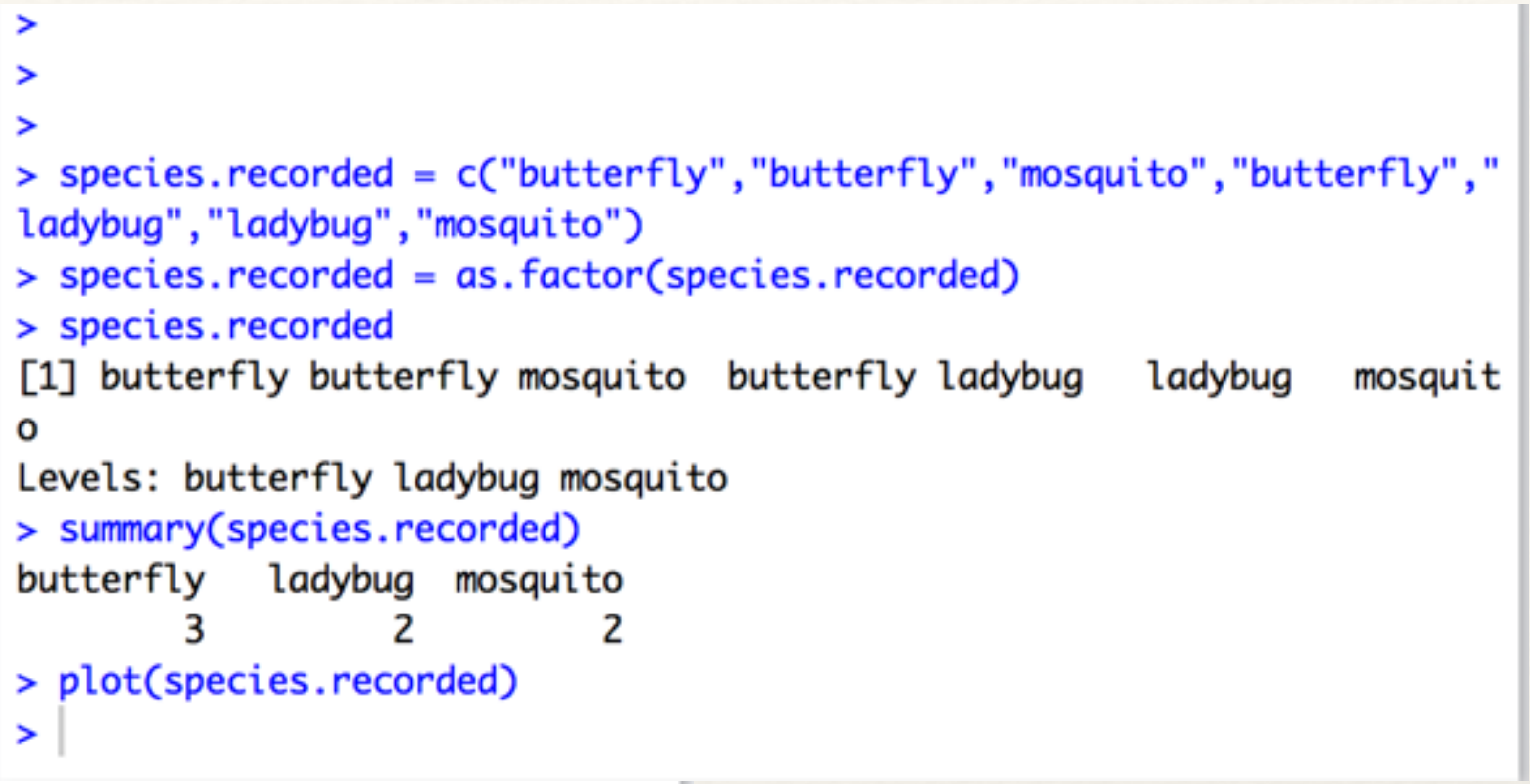
Key Programming concepts: Review of data types

- *summary* can be used with factors to get frequencies in each category (or "level")

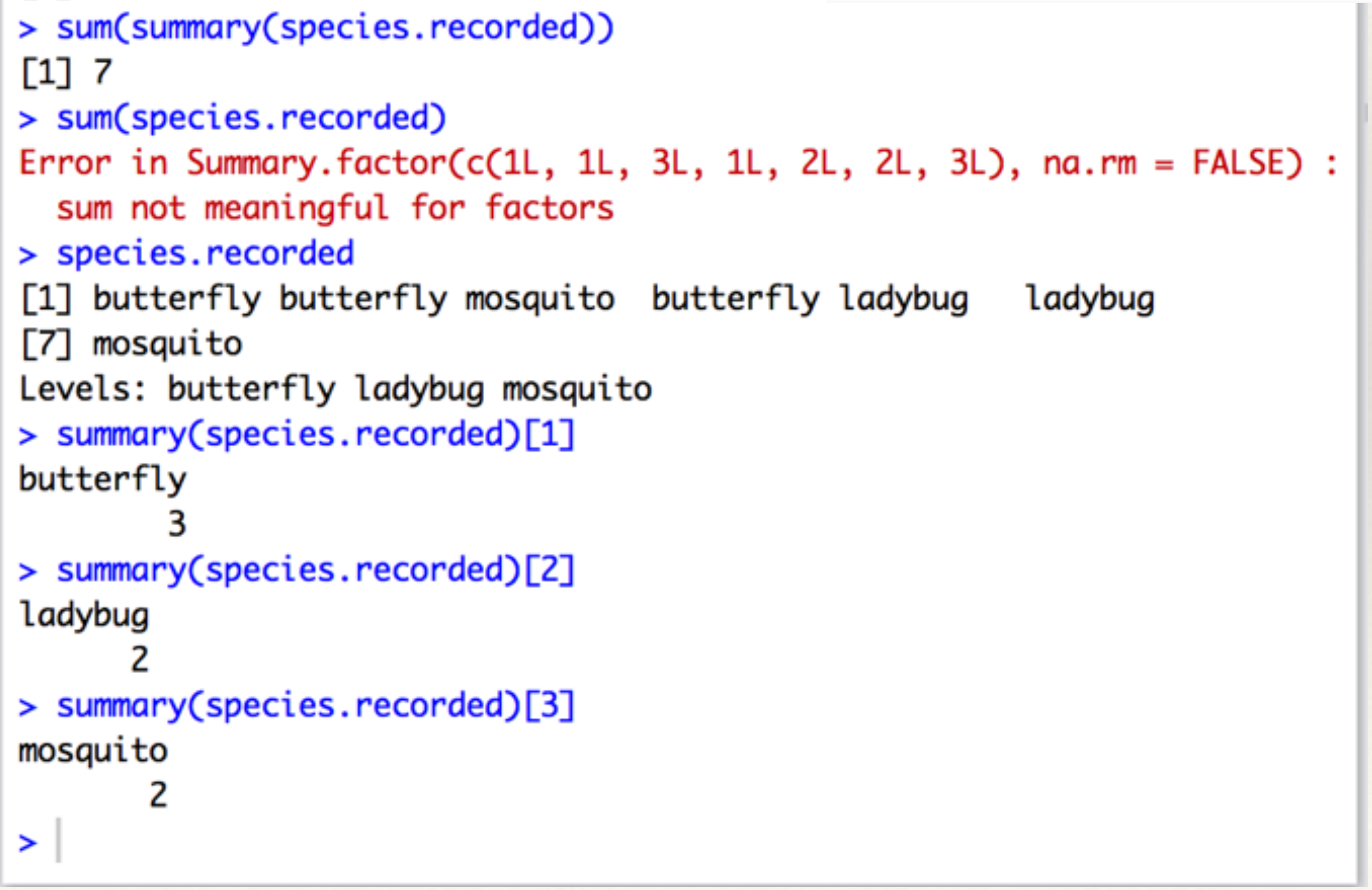


```
>
>
>
> species.recorded = c("butterfly","butterfly","mosquito","butterfly","
ladybug","ladybug","mosquito")
> species.recorded = as.factor(species.recorded)
> species.recorded
[1] butterfly butterfly mosquito  butterfly ladybug   ladybug   mosquit
o
Levels: butterfly ladybug mosquito
> summary(species.recorded)
butterfly   ladybug  mosquito 
        3         2         2 
> plot(species.recorded)
> 
```





```
>
>
>
> species.recorded = c("butterfly","butterfly","mosquito","butterfly","
ladybug","ladybug","mosquito")
> species.recorded = as.factor(species.recorded)
> species.recorded
[1] butterfly butterfly mosquito  butterfly ladybug   ladybug   mosquit
o
Levels: butterfly ladybug mosquito
> summary(species.recorded)
butterfly   ladybug  mosquito 
        3         2         2 
> plot(species.recorded)
> 
```



```
> mean(summary(species.recorded))
[1] 2.333333
> max(summary(species.recorded))
[1] 3
> sum(summary(species.recorded))
[1] 7
> sum(species.recorded)
Error in Summary.factor(c(1L, 1L, 3L, 1L, 2L, 2L, 3L), na.rm = FALSE) : 
  sum not meaningful for factors
> species.recorded
[1] butterfly butterfly mosquito  butterfly ladybug   ladybug  
[7] mosquito 
Levels: butterfly ladybug mosquito
> summary(species.recorded)[1]
butterfly 
        3 
> summary(species.recorded)[2]
ladybug 
      2 
> summary(species.recorded)[3]
mosquito 
       2 
> 
```

You can “do things” (apply functions) to the summary (frequency of each “factor” level

Key Programming concepts: Review of data types

- A simple model that takes advantage of factors
- A model to compute an index of species diversity from a list of recorded species

$$D = \sum (n / N)^2$$

where n is the number of individuals in each species, and N is total number

Key Programming concepts: Review of data types

```
#' Simpson's Species Diversity Index
#'
#' Compute a species diversity index
#' @param species list of species (names, or code)
#' @return value of Species Diversity Index
#' @examples
#' compute_simpson_index(c(“butterfly","butterfly","mosquito","butterfly",
#’ ”ladybug","ladybug")))
#' @references
#' http://www.tiem.utk.edu/~gross/bioed/bealsmodules/simpsonDI.html

compute_simpson_index = function(species) {

species = as.factor(species)
tmp = (summary(species)/sum(summary(species))) ** 2
diversity = sum(tmp)
return(diversity)
}
```

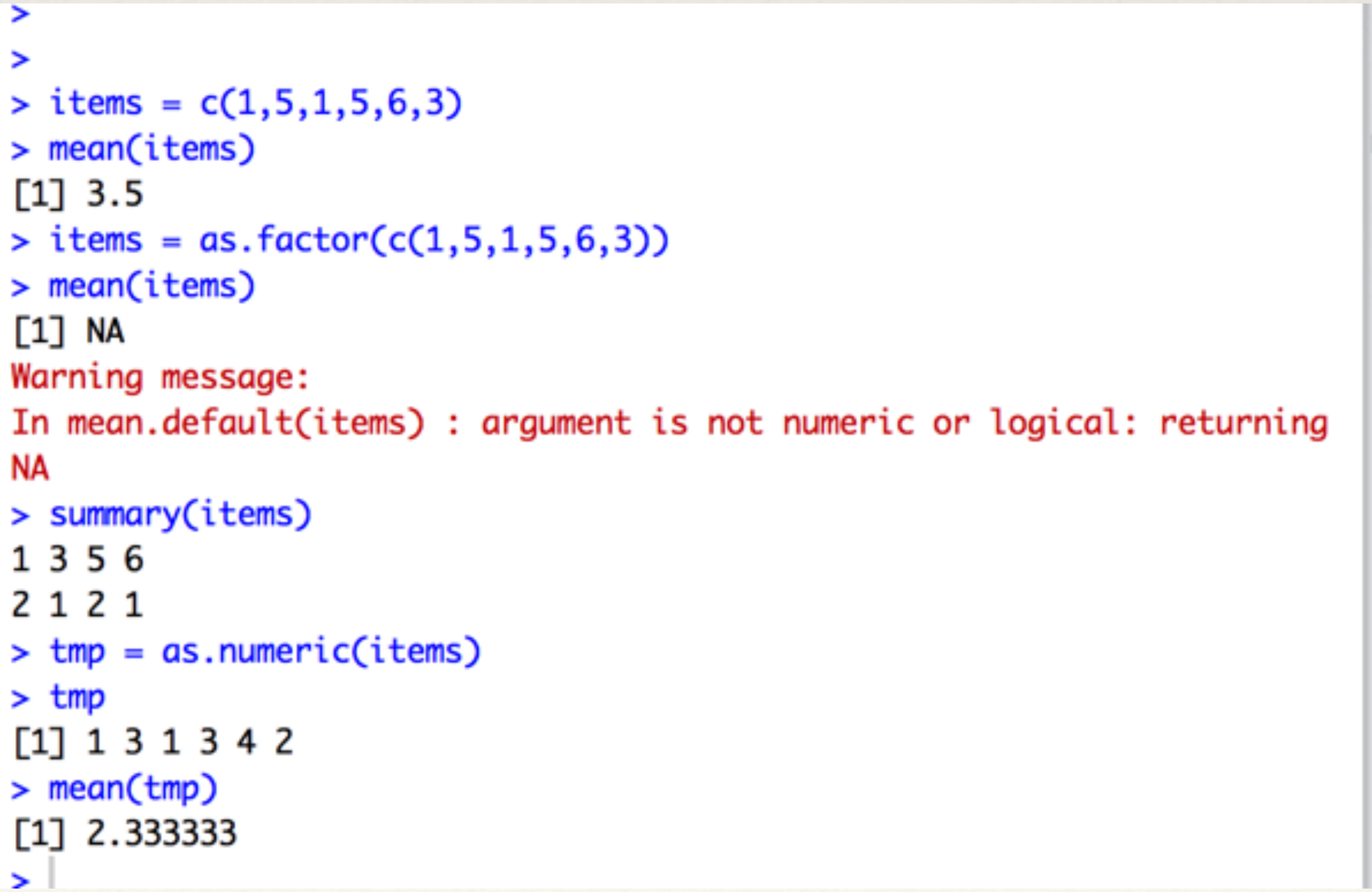
Key Programming concepts: Review of data types

- *lm* is an example of a function that returns a list

```
>
> res = lm(obs$prices~obs$forestC)
> names(res)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> res$coefficients
(Intercept) obs$forestC
 14.9789368   0.1865644
> res$model
  obs$prices obs$forestC
1         23          59
2         44          88
3         60         100
4          4          10
5          2           8
6         33          79
7         59         300
>
```

Data Structures

- a bit more on factors; a list of numbers can also be a factor but then they are not treated as actual numbers - you could think of them as "codes" or addresses or..
- use *as.numeric* or *as.character* to go back to a regular vector from a factor



```
>
>
> items = c(1,5,1,5,6,3)
> mean(items)
[1] 3.5
> items = as.factor(c(1,5,1,5,6,3))
> mean(items)
[1] NA
Warning message:
In mean.default(items) : argument is not numeric or logical: returning
NA
> summary(items)
1 3 5 6
2 1 2 1
> tmp = as.numeric(items)
> tmp
[1] 1 3 1 3 4 2
> mean(tmp)
[1] 2.333333
> 
```

Data Structures

Generating “fake” or example data - sample

```
tmp = c("ponderosa","jack","white","lodgepole","douglasfir","oak")
obs.trees= list(species=sample(tmp, replace=T, size=100))

obs.trees$carbon = runif(min=5, max=20, n=100)

# run our functions
compute_simpson_index(obs.trees$species)

# save data for use in your R package
save(obs.trees, file="data/obstrees.RData")
```

Data Structures

- **vector, (c)**
- **matrices, arrays**
- **data frames**
- lists
- **factors**

Key Programming concepts: Review of data typessd

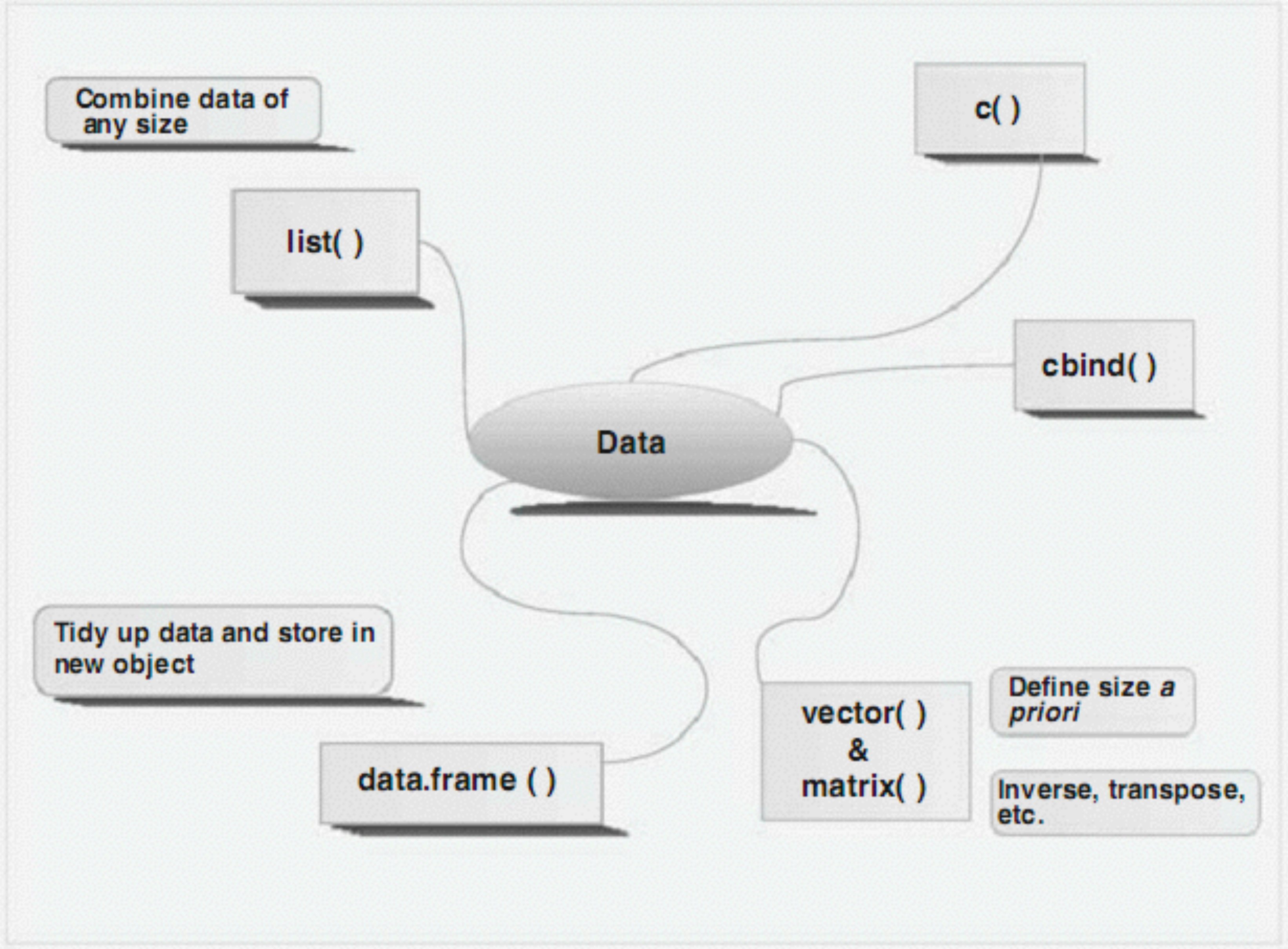
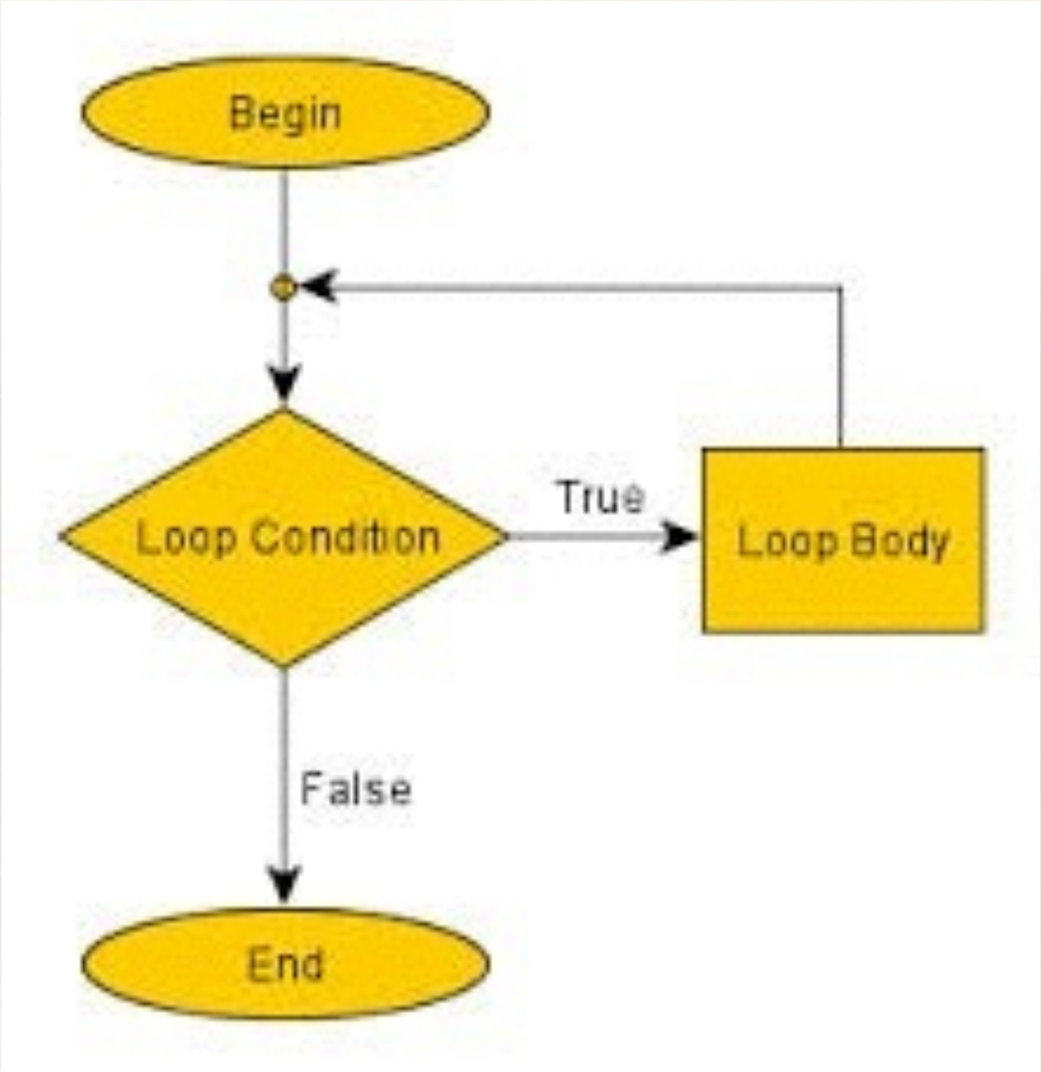


Fig. 2.1 Overview of various methods of storing data. The data stored by `cbind`, `matrix`, or `data.frame` assume that data in each row correspond to the same observation (sample, case)

http://www.simonqueenborough.com/R/basic/figure/data-types.png

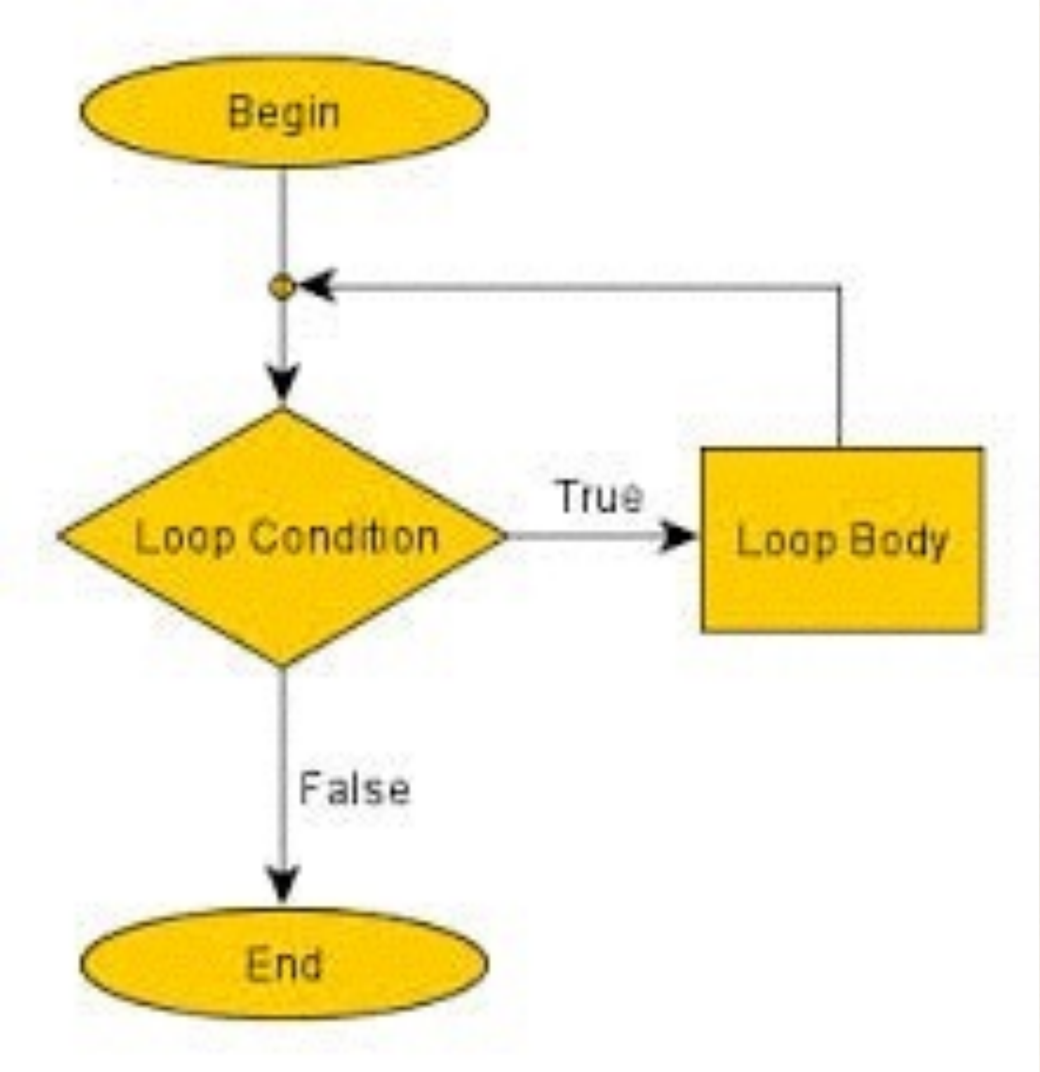
Key Programming concepts: Looping

- Loops are fundamental in all programming languages: and are frequently used in models



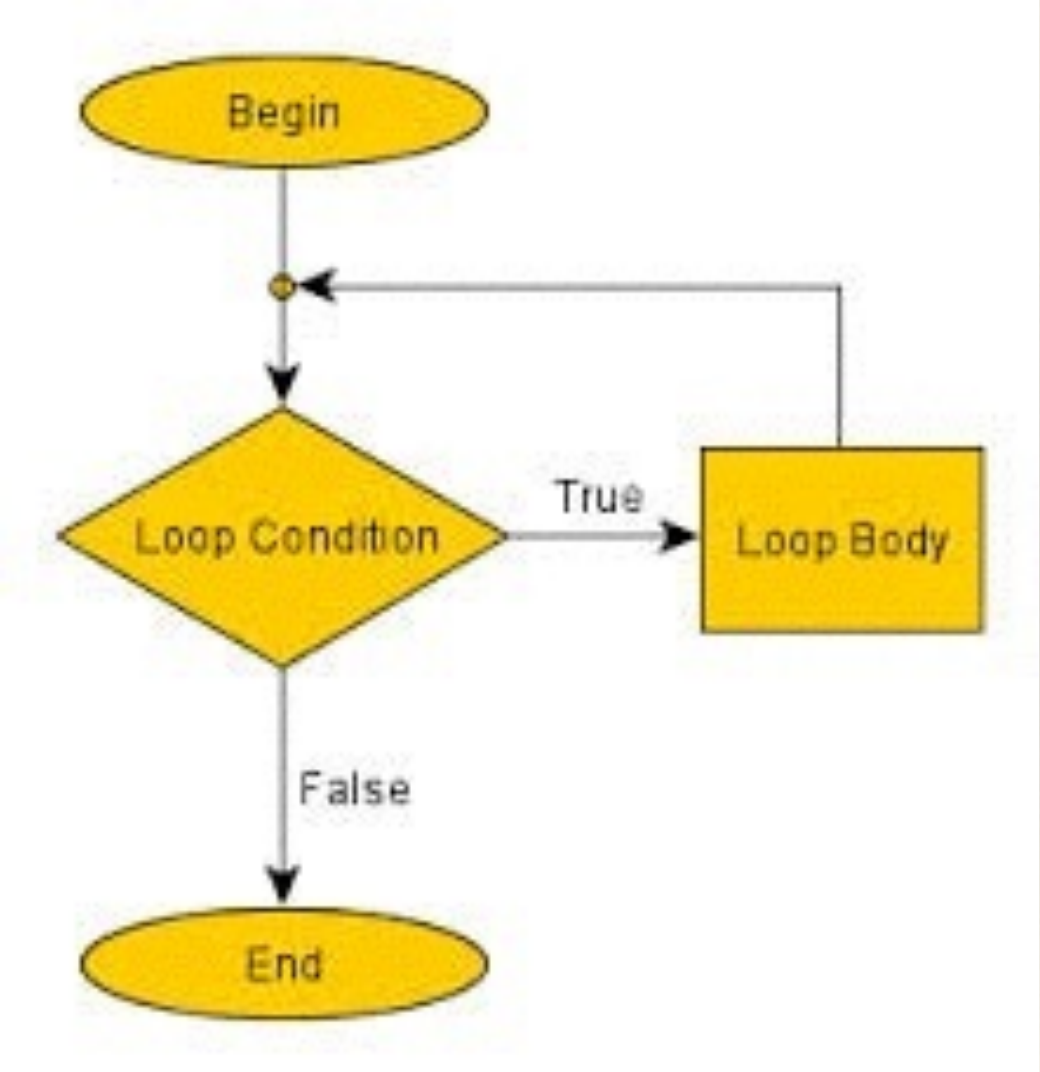
Key Programming concepts: Looping

- Two distinctive reasons for looping
- Apply the same equations (e.g for power generation) over a range of parameter values
- Evolve a variable through time (or space), when the variable's value at the next time step depends on the previous one (e.g growing a population)



Key Programming concepts: Looping

- All loops have this basic structure - repeat statements (loop body) until a condition is true



Key Programming concepts: Looping

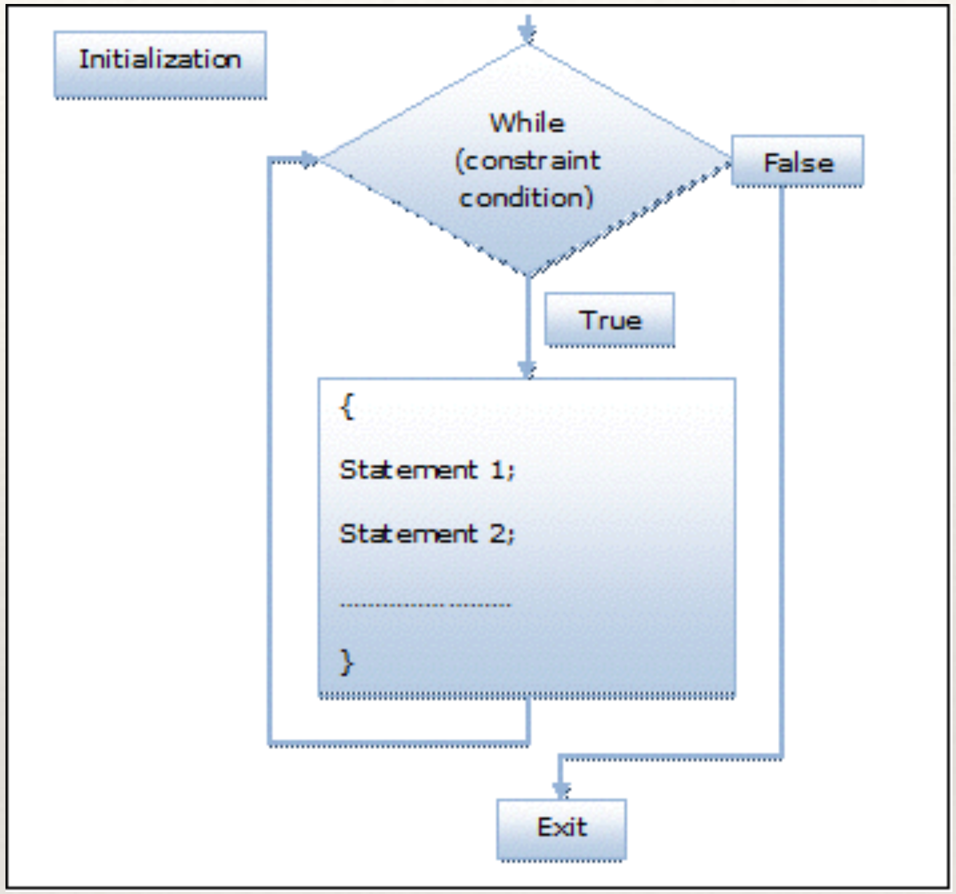
- In R, the most commonly used loop is the ***For*** loop
- ***for (i in 1:n){ statements}***
- In “for” loops the i (or whatever variable you want to use as the counter, is automatically incremented each time the loop is gone through; and the looping ends when i (the counter) reaches n
- What is x? alpha? after this loop is run

```
>x=0
> for (alpha in 1:4) { x = x+alpha}
```

```
>
>
> x=0
> for (alpha in 1:4) { x = x+alpha}
>
>
> alpha
[1] 4
> x
[1] 10
```

Key Programming concepts: Looping

- Another useful looping construct is the ***While*** loop
- keep looping until a condition is met
- Useful when you don't know what "n" in the for 1 in to "n" is
- often used in models where you are evolving
  - accumulate something until a threshold is reached (population, energy, biomass?



Key Programming concepts: Looping

- A simple *while* loop example

```
>
>
> alpha = 0
> x = 0
> while (alpha < 100) { alpha = alpha + x; x = x+1}
> x
[1] 15
> alpha
[1] 105
>
```

- alpha = (1+2+3+4+5+6+7+8+9+10+11+12+13+14) = 105

Key Programming concepts: Looping

- A more useful *while* loop example
- A question: if a metal toxin in a lake increases by 1% per year, how many years will it take for the metal level to be greater than 30 units, if toxin is current at 5 units
- there are other ways to do this, but a while loop would do it

why won’t this work?

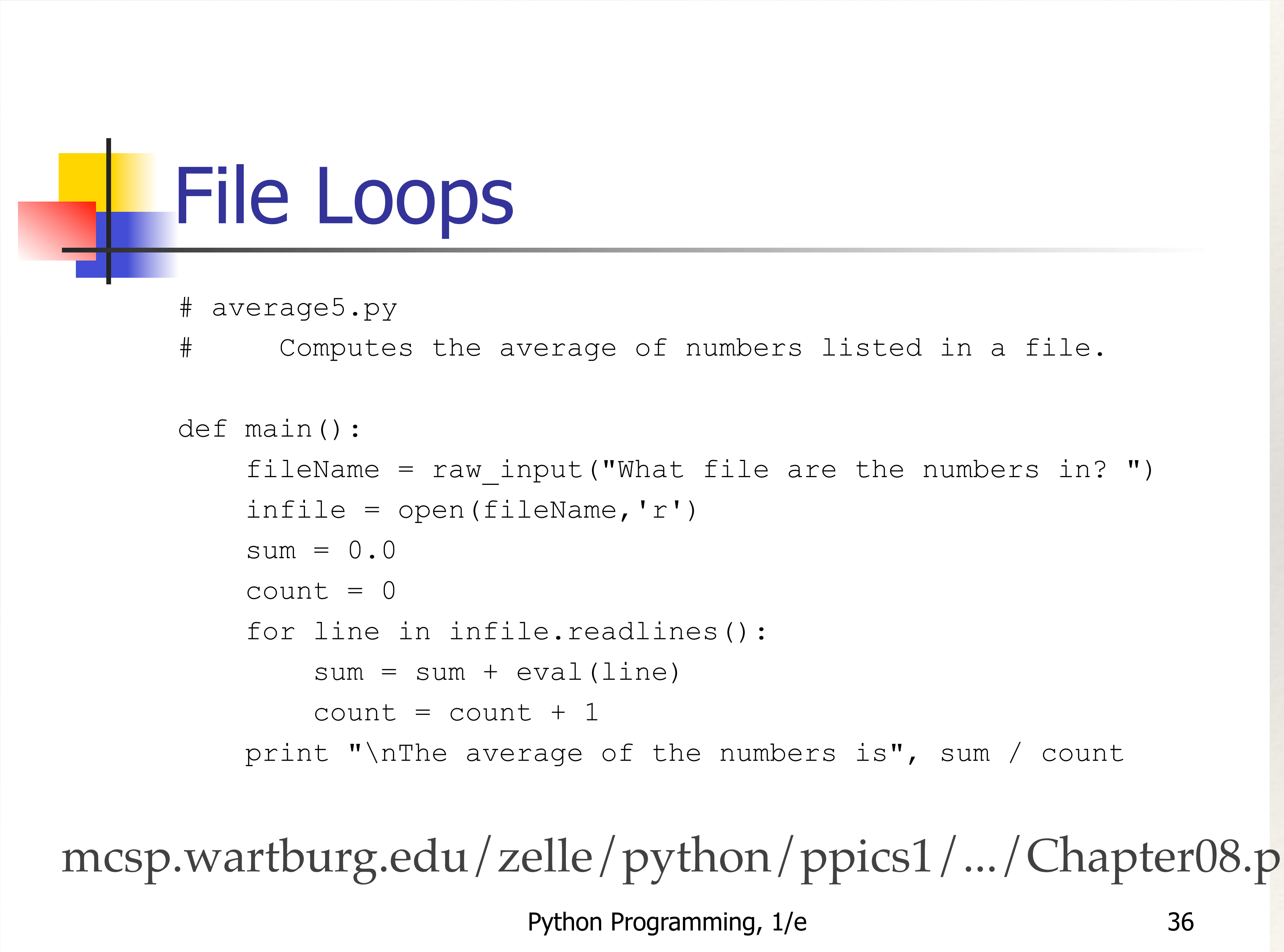
```
> >
> pollutant.level = 5
> while (pollutant.level < 30 ) {
+ pollutant.level = pollutant.level + 0.01* pollutant.level
+ yr = yr + 1
+ }
>
```

Key Programming concepts: Looping

```
> yr=1
> pollutant.level = 5
> while (pollutant.level < 30 ) {
+ pollutant.level = pollutant.level + 0.01* pollutant.level
+ yr = yr + 1
+ }
> > yr
[1] 182
> pollutant.level
[1] 30.2788
```

Key Programming concepts: Looping

- Most programming languages have For and while loops



File Loops

```
# average5.py
#     Computes the average of numbers listed in a file.

def main():
    fileName = raw_input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in infile.readlines():
        sum = sum + eval(line)
        count = count + 1
    print "\nThe average of the numbers is", sum / count
```

Python Programming, 1/e 36

mcsp.wartburg.edu/zelle/python/ppics1/.../Chapter08.p

Key Programming concepts: Control Structures

- *if*(cond) expression

```
>
> a=4
> b=10
> if(a > b) win = "a"
> if(b > a) win = "b"
> win
[1] "b"
>
```

Conditions:
== equal
> greater than
>= greater than or equal to
< less than
<= less than or equal to
%in% is in a list of something

- *ifelse*(cond, true, false)

```
>
> win = ifelse(a > b, "a","b")
> win
[1] "b"
>
>
```

&& AND
| | OR
is.null()

Key Programming concepts: Control Structures

If can also be used to choose what you return from a function

```
compute_seasonal_flow = function(str,kind) {

str$season = ifelse( str$month %in% c(1,2,3,10,11,12),"winter","summer")

tmp = subset(str, str$season=="winter")
if(kind=="mean") winter= mean(tmp$mm)
if(kind=="max") winter= max(tmp$mm)
if(kind=="min") winter=min(tmp$mm)

tmp = subset(str, str$season=="summer")
if(kind=="mean") summer= mean(tmp$mm)
if(kind=="max") summer= max(tmp$mm)
if(kind=="min") summer=min(tmp$mm)


return(list(summer=summer, winter=winter))
}
```

Key Programming concepts: Control Structures

If can also be used to choose what you return from a function

```
>
>
> compute_seasonal_flow(streamflow,"mean")
$summer
[1] 1.538304

$winter
[1] 0.6200728

> compute_seasonal_flow(streamflow,"max")
$summer
[1] 23.66069

$winter
[1] 71.97168
```

Key Programming concepts: Review of data types

```
#' compute_NPV
#'
#' compute net present value
#' @param value/cost ($)
#' @param time in the future that cost/value occurs (years)
#' @param discount rate, default 0.01
#' @return value in $


compute_NPV = function(value, time, discount=0.01) {

  result=0.0
  if (length(value) < length(time) )
    value = rep(value, times=length(time))
  for (i in 1:length(time) ) {
  result = result + value[i] / (1 + discount)**time[i]
  }

  return(result)
}
```

Key Programming concepts: Review of data types

```
#' compute_carbon
#'
#' computes growth given species, and spring temperature and precipitation
#' @param  currentbiomass (mgC)
#' @param  species (name of species)
#' @param  species.parm (data frame with species, maxrate (%C/yr), topt (C), pmax
(mm), pmin(mm)
#' @param springt (C) springtime temperature
#' @param springp (mm) springtime rainfall
#' @return growth (mgC/year)
compute_carbon = function(currentbiomass, species, species.parm, springt, springp)

  idx = match(obs.trees$species, coeff.species.growth$species)
  growth.rate = species.parm$maxrate[idx]
  growth.rate = growth.rate - abs(springt-species.parm$topt[idx])/20
  peffect = (springp -species.parm$pmin[idx])/
      (species.parm$pmin[idx]-species.parm$pmax[idx])*species.parm$maxrate[idx]
  growth.rate = ifelse(springp < species.parm$pmin[idx], 0,
                       ifelse(springp > species.parm$pmax[idx], growth.rate,
                              growth.rate-peffect) )
  new.carbon = currentbiomass*growth.rate
  return(new.carbon)
}
```

```
# load "stuff" in your package including R
load_all()
result = spring.summary(clim)
View(result)

# save data for use in your R package
save(clim, file="data/clim.RData")

# generate data
tmp = c("ponderosa","jack","white","lodgepole","douglasfir","oak")
obs.trees= list(species=sample(tmp, replace=T, size=100))

obs.trees$carbon = runif(min=5, max=20, n=100)

coeff.species.growth = data.frame(species=c("ponderosa","jack","white","lodgepole","douglasfir","oak"),
maxrate=c(1.2,1.1,1.3,1.6,1.9,1.2),
topt = c(9,7,6,5,7,12), pmax = c(300,300,300,400,600,400), pmin = c(100,200,200,250,250,100))

# run our functions
compute_simpson_index(obs.trees$species)
compute_NPV(value=100, time=20, discount=0.01)
compute_carbon(obs.trees$carbon, obs.trees$species, coeff.species.growth, 9, 200)

# save data for use in your R package
save(obs.trees, file="data/obstrees.RData")
save(coeff.species.growth, file=“data/coeff.species.growth.RData")
```

Key Programming concepts: Looping

- Loops can be “nested” - one loop inside the other
- For example, if we want to calculate NPV for a range of different interest rates and a range of damages that may be incurred 10 years in the future
    - using a function called compute_npv
- Steps
    - define inputs (interest rates, damages)
    - define a data structure to store results
    - define function/model (already available)
    - use looping to run model for all inputs and store in data structure

Key Programming concepts: Looping

- Now we can start to build a more complex program
- Lets say we want to figure out the benefits of a forest, that include both carbon storage and biodiversity
- Conceptual model
- Implementation using our building blocks

```
#' Forest Ecosystem Benefit Computer
#'
#' compute_ecobenefit()
#'
#' Computes an estimate of forest ecosystem benefits that include both biodiversity and carbon
#' @param tree dataframe with species and current biomass
#' @param carbonprice ($) price paid for carbon
#' @param biodiversityprice ($) price paid for biodiversity in a given year
#' @param paramters for growth model
#' @param clim dataframe with tmax, tmin and precip for each day
#' @param discount discount rates
#' @return annual.benefit and NPV of all benefits over all years
#' @examples
compute_ecobenefit = function(tree, carbonprice, biodiversityprice, coeff.species.growth, clim, discount) {

        spring = spring.summary(clim)
        benefit = matrix(nrow=nrow(spring$all.springT), ncol=length(tree$species))
        for (i in 1:nrow(spring$all.springT)) {
                benefit[i,]=compute_carbon(tree$carbon,trees$species, coeff.species.growth,
                spring$all.springT$x[i], spring$all.springP$precip[i])
                }
        benefit = as.data.frame(benefit)*carbonprice
        benefit$biodiversity = compute_simpson_index(tree$species)*biodiversityprice

        annual.benefit = apply(benefit,1,sum)
        present.benefit = compute_NPV(value=annual.benefit, time=seq(from=1,to=length(annual.benefit)),
discount)
  return(list(annual.benefit=annual.benefit, NPV=present.benefit))
}
```

Key Programming concepts: Looping

Run our more complex function - different discount rates

```
compute_ecobenefit(obs.trees, 20, 10,  coeff.species.growth, clim, 0.01)
compute_ecobenefit(obs.trees, 20, 10,  coeff.species.growth, clim, 0.05)
```