
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 implementation.
- (b) Refactor code in preference to explaining how it works.
- (c) Embed the documentation for a piece of software in the software.

8. Collaborate.

- (a) Use pre-merge code reviews.
- (b) Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
- (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

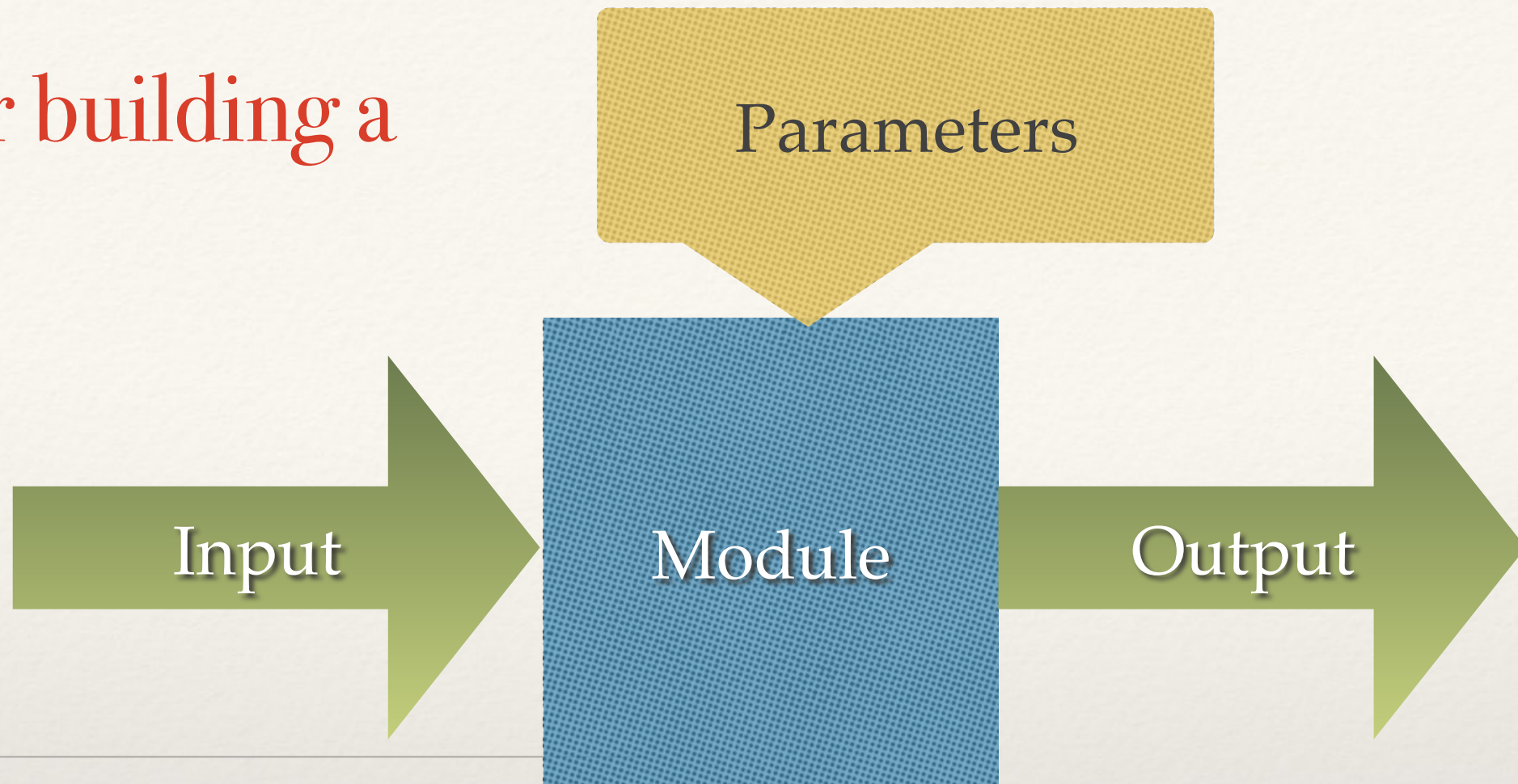
❖ Common problems

- ❖ 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



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

- ❖ Inputs - sometimes separated into input data and parameters
 - ❖ input data = the “what” that is manipulated
 - ❖ parameters determine “how” the manipulation is done
 - ❖ “sort -n file.txt”
 - ❖ sort is the program - set of instructions - its a black box
 - ❖ input is file.txt
 - ❖ parameters is -n
 - ❖ output is a sorted version of file.txt
- ❖ my iphone app for calculating car mileage
 - ❖ inputs are gallons and odometer readings at each fill up
 - ❖ graph of is miles / gallon over time
 - ❖ parameters control units (could be km / liter, output couple be presented as a graph or an average value)

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

Building Blocks

- ❖ Instructions inside the building blocks/box
 - ❖ Numeric data operators
 - ❖ $+, -, /, *, \% * \%$
 - ❖ Strings
 - ❖ substr, paste..
 - ❖ Math
 - ❖ sin, cos, exp, min, max...
 - ❖ these are themselves programs - boxes
- ❖ R-reference card is useful!

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

A simple program: Example

- ❖ Input: Reservoir height and flow rate
- ❖ Output: Instantaneous power generation (W / s)
- ❖ Parameters: $K_{\text{Efficiency}}$, ρ (density of water), g (acceleration due to gravity)

$$P = \rho * h * r * g * K_{\text{Efficiency}};$$

P is Power in watts, ρ is the density of water ($\sim 1000 \text{ kg/m}^3$), h is height in meters, r is flow rate in cubic meters per second, g is acceleration due to gravity of 9.8 m/s^2 , $K_{\text{Efficiency}}$ is a coefficient of efficiency ranging from 0 to 1.

Simple Functions

```
#' Power Required by Speed
#'  
#' This function determines the power required to keep a vehicle moving  
at  
#' a given speed  
#' @param cdrag coefficient due to drag default=0.3  
#' @param crolling coefficient due to rolling/friction default=0.015  
#' @param v vehicle speed (m/2)  
#' @param m vehicle mass (kg)  
#' @param A area of front of vehicle (m2)  
#' @param g acceleration due to gravity (m/s) default=9.8  
#' @param pair (kg/m3) default =1.2  
#' @return power (W)  
  
power = function(cdrag=0.3, crolling=0.015,pair=1.2,g=9.8,V,m,A) {  
P = crolling*m*g*v + 1/2*A*pair*cdrag*v**3  
return(P)  
}  
  
v=seq(from=0, to=100, by=10)  
plot(v, power(V=0.447*v, m=31752, A=25))  
lines(v, power(V=0.447*v, m=61752, A=25))
```

Use Lists to return more complex info

```
#' 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$month %in% spring.months)
  mean.springT = mean(c(spring$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 ))
}
❖
```




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

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

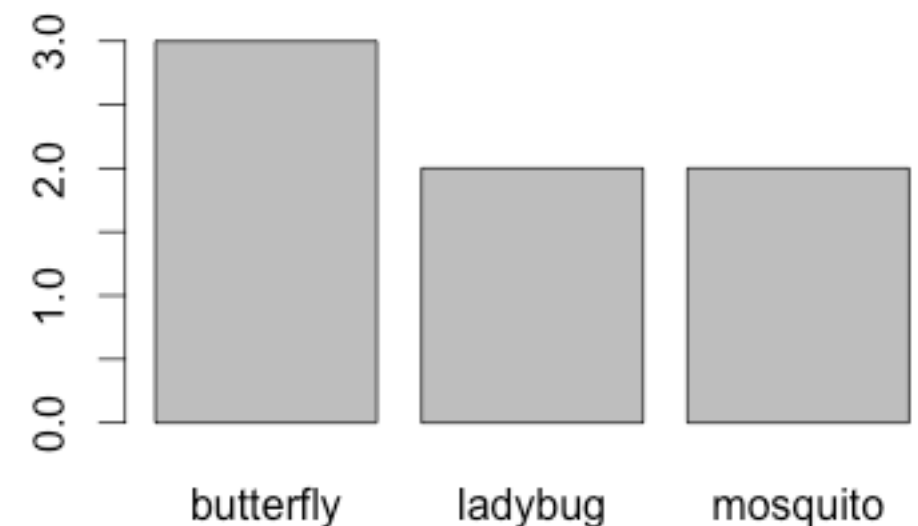
- ❖ Factors (a bit tricky, basically a vector of “things” that has different levels (classes); not really numeric - so you can’t average them!)
- ❖ But can be useful for doing “calculations” with categories

```
>
> 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
0
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
```

```
> |
```

```
>
```

```
>
```

```
>
```

```
> 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)
```

```
> |
```

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

```
# ' 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 = value / (1 + discount)**time
  result
}
```

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
>
```


Key Programming concepts: Review of data types

```
#' computes profit from price for forest plot and Mg/C in that plot
#' @param price ($)
#' @param carbon (MgC)
#' @return list with mean, min, and max prices
compute_carbonvalue = function(price, carbon) {

  cost.per.carbon = price/carbon
  a = mean(cost.per.carbon)
  b = max(cost.per.carbon)
  c = min(cost.per.carbon)

  result = list(avg=a, min=c, max=b)
  return(result)
}
```

Key Programming concepts: Review of data types

❖ example: returning lists from a function

```
#' Describe diversity based on a list of species
#'\n
#'\n
#' Compute a species diversity index
#' @param species list of species (names, or code)
#' @return list with the following items
#'\describe{\n
#'\item{num}{ Number of distinct species}\n
#'\item{simpson}{Value of simpson diversity index}\n
#'\item{dominant}{Name of the most frequently occurring species}\n
#'\}\n
#'\n
#' @examples\n
#'\n
compute_diversity(c("butterfly","butterfly","mosquito","butterfly","ladybug",\n"ladybug"))\n
#'\n
#' @references\n
#'\n
http://www.tiem.utk.edu/~gross/bioed/bealsmodules/simpsonDI.html\n
\n
compute_diversity = function(species) {\n
\n
species = as.factor(species)\n
tmp = (summary(species)/sum(summary(species))) ** 2\n
diversity = sum(tmp)\n
nspecies = length(summary(species))\n
tmp = which.max(summary(species))\n
dominant = names(summary(species)[tmp])\n
return(list(num=nspecies, simpson=diversity, dominant=dominant))\n
}
```


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$carbon.avg = runif(min=5, max=20, n=100)  
obs.trees$carbon.wet = obs.trees$carbon.avg * 1.2  
obs.trees$carbon.dry = obs.trees$carbon.avg * 0.8
```

```
# run our functions  
compute_simpson_index(obs.trees$species)  
compute_carbonvalue(2.5, obs.trees$carbon)  
compute_NPV(value=100, time=20, discount=0.01)
```

```
# 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 types

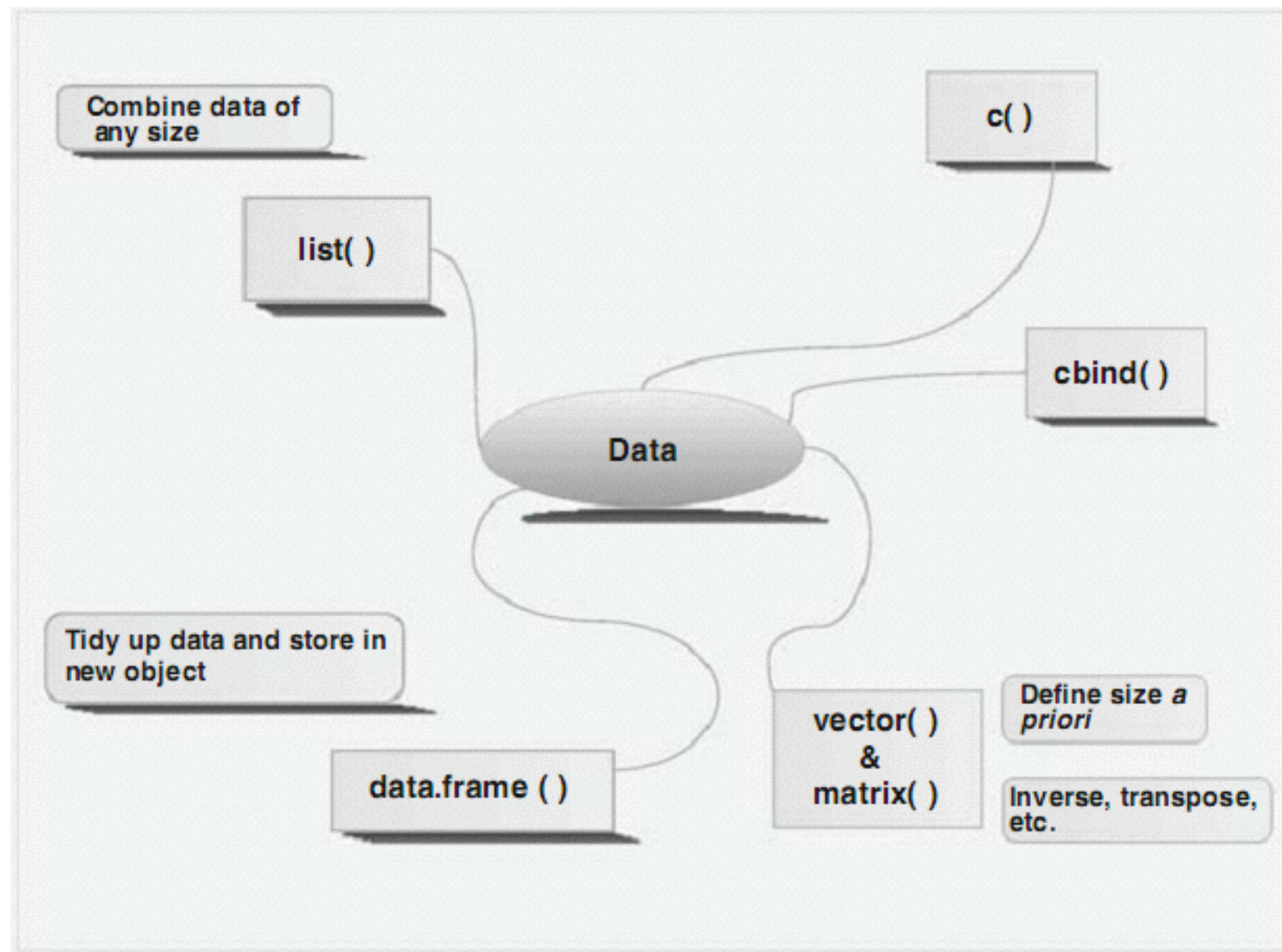


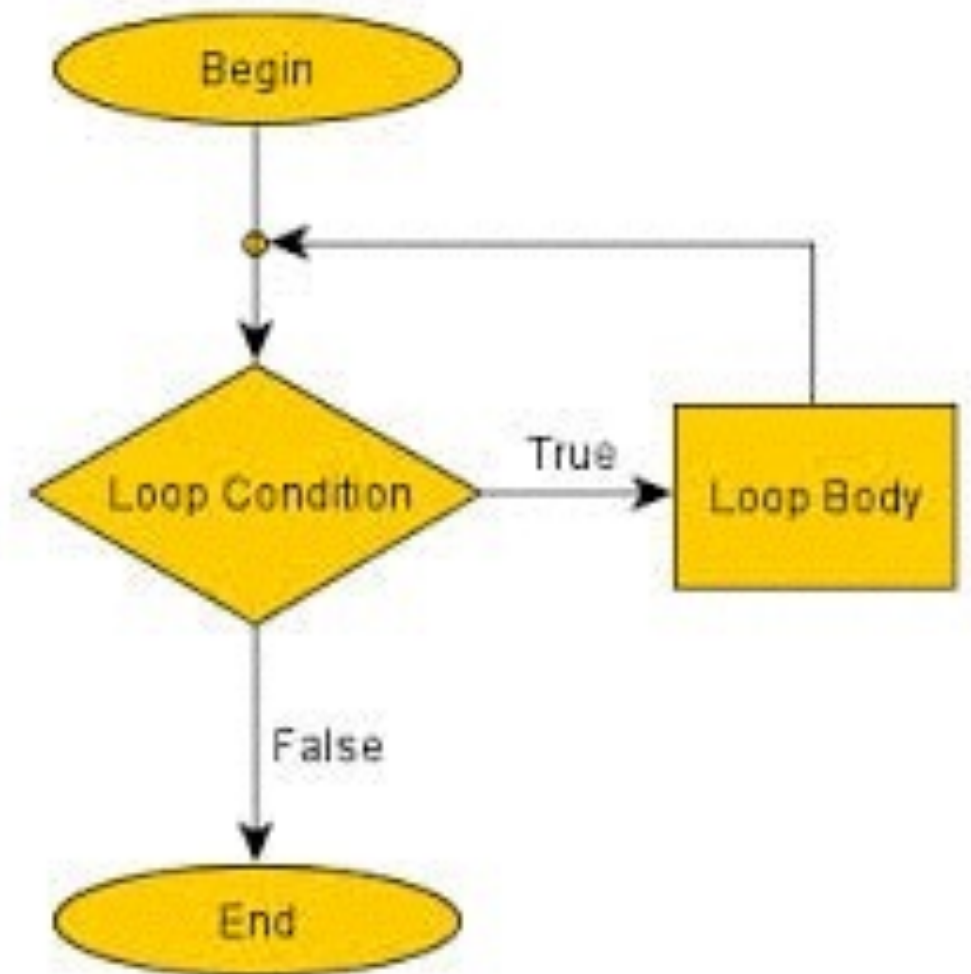
Fig. 2.1 Overview of various methods of storing data. The data stored by `cbind`, `matrix`, or `data.frame` assume that data in each row correspond to the same observation (sample, case)

Key Programming concepts: Looping

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

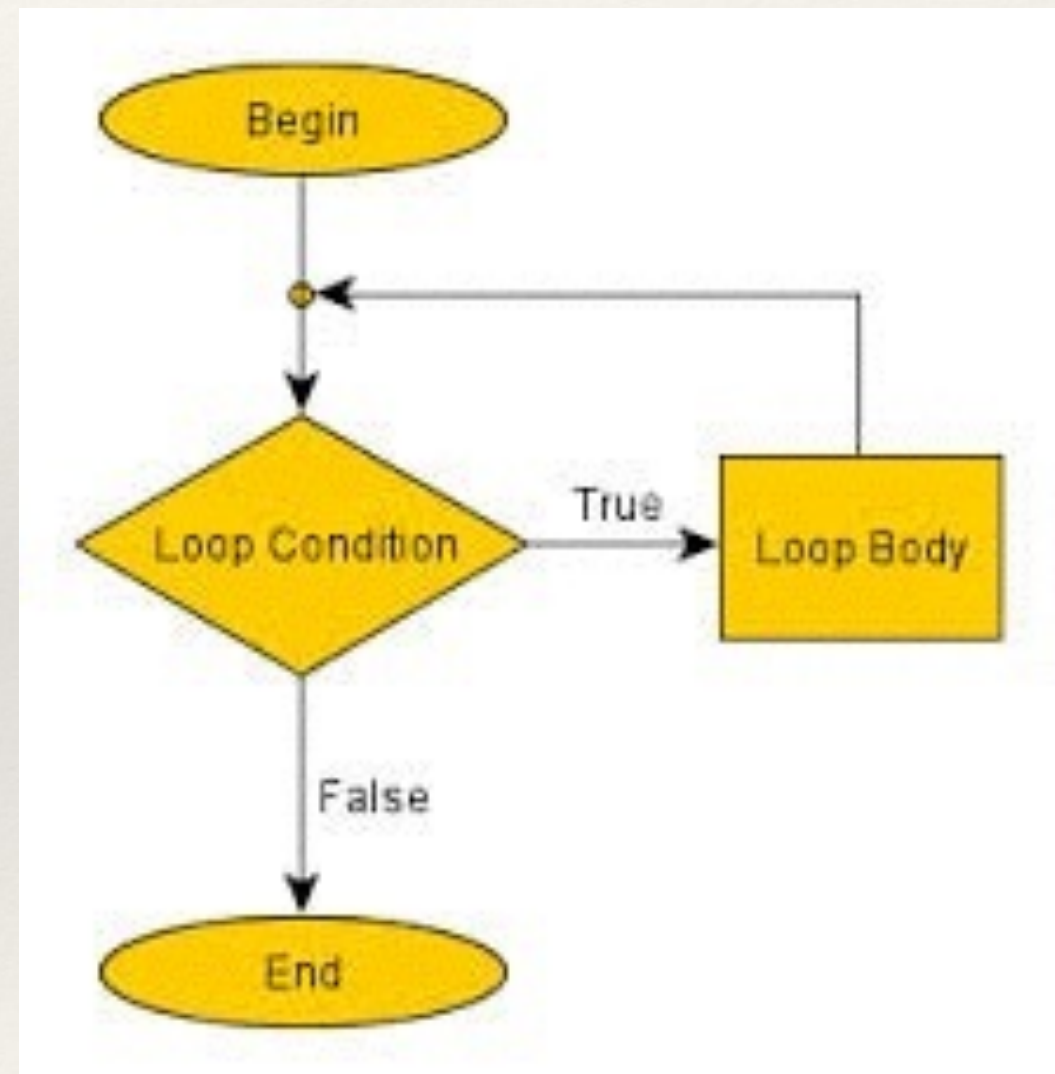


LOOPS REPEAT
ACTIONS...
SO YOU DON'T HAVE TO



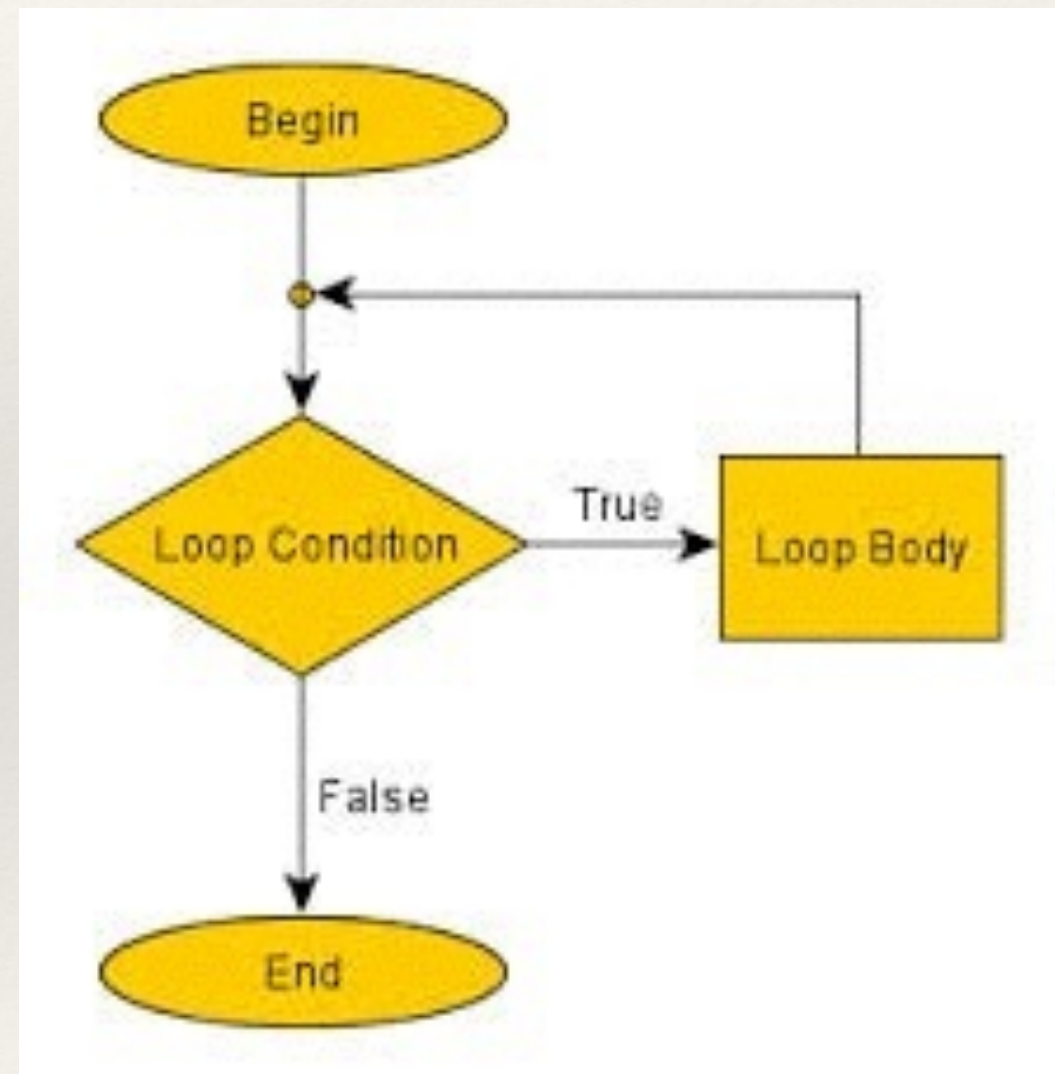
Key Programming concepts: Looping

- ❖ 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

- ❖ 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

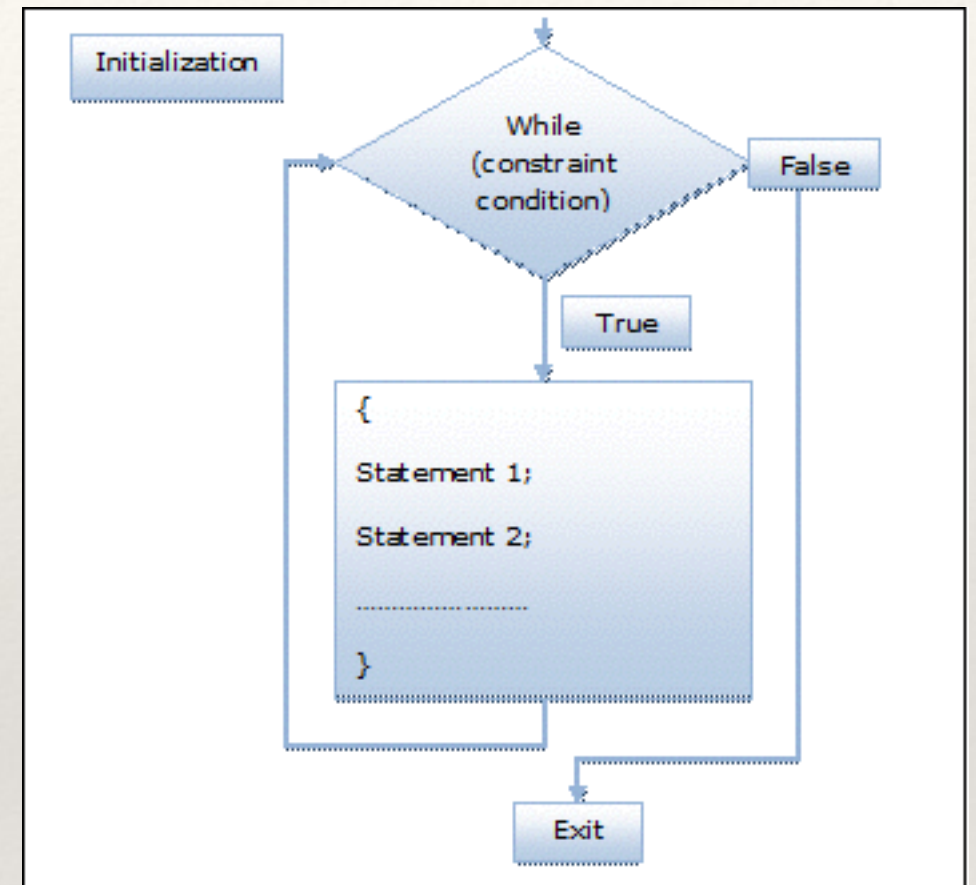
Key Programming concepts: Looping

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

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
```

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"  
>  
>
```

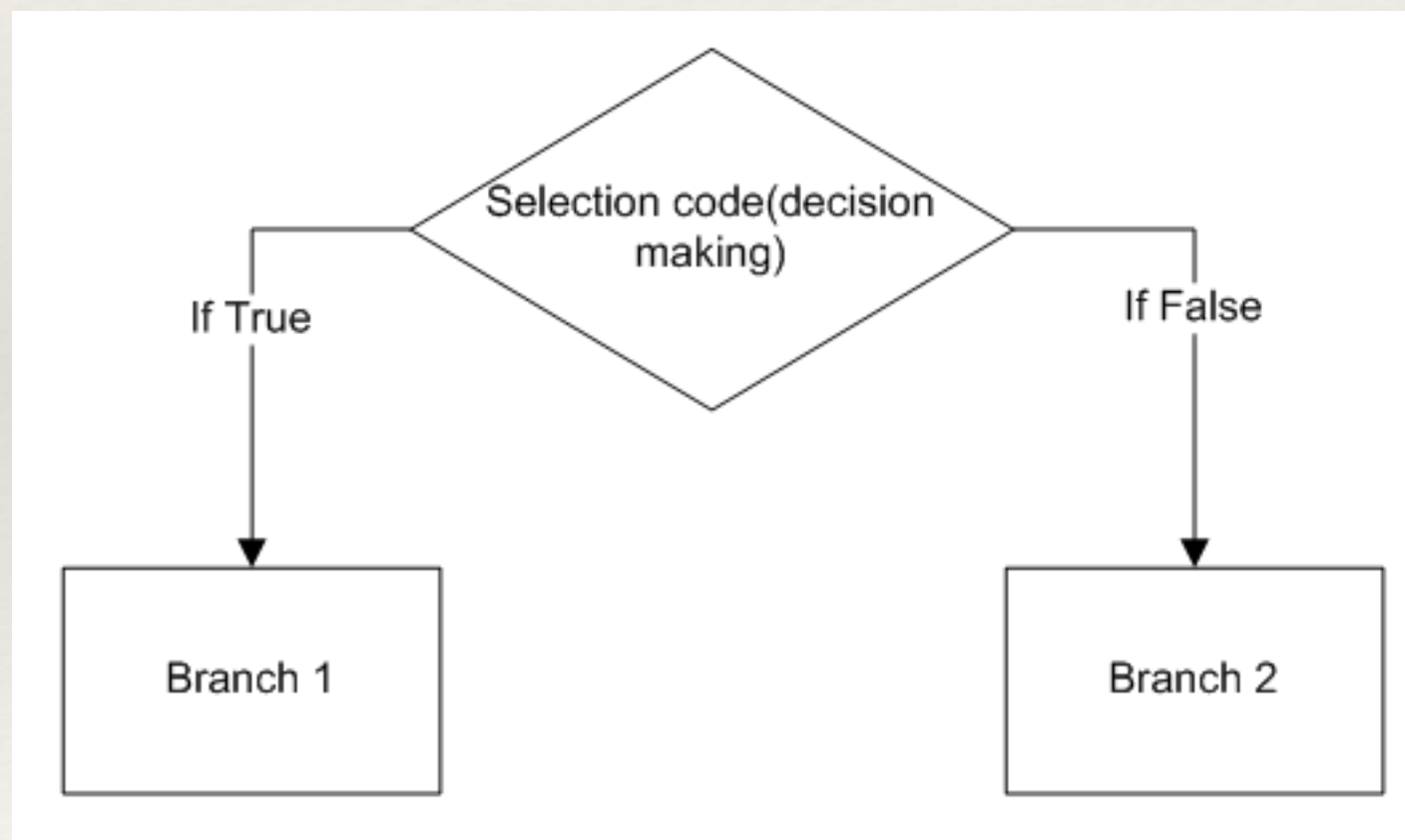
&& AND

|| OR

is.null()

Key Programming concepts: Looping

- ❖ Inside functions *if*(cond) {expression} *else* {expression}
- ❖ the expression can always have multiple statements using {}
- ❖ *If* can be useful for branching in your model



Key Programming concepts: Control Structures

```
#' compute annual yield
#'
#' Function to compute yeild of different fruits as a function of annual temperature and precipitation
#' @param T annual temperature (C)
#' @param P annual precipitation (mm)
#' @param ts slope on temperature
#' @param tp slope on precipitation
#' @param intercept (kg)
#' @param irr Y or N (default N)
#' @return yield in kg
```

```
compute_yield = function(T, P, ts, tp, intercept, irr="N") {
```

```
  if (irr=="N"){
    yield = tp*P + ts*T + intercept
  }
  else {
    yield = ts*T + intercept
  }

  return(yield)
}
```

```
>
> compute_yield(32,200, 0.2, 0.4, 500)
[1] 586.4
> compute_yield(32,200, 0.2, 0.4, 500, "N")
[1] 586.4
> compute_yield(32,200, 0.2, 0.4, 500, "Y")
[1] 506.4
```

```
#' @param str data frame with columns month and mm (mm is daily streamflow)
compute_seasonal_meanflow = function(str) {

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

  tmp = subset(str, str$season=="winter")
  mean.winter = mean(tmp$mm)

  tmp = subset(str, str$season=="summer")
  mean.summer = mean(tmp$mm)
  return(list(summer=mean.summer, winter=mean.winter))
}
```

```
> compute_seasonal_meanflow(streamflow)
$summer
[1] 1.538304

$winter
[1] 0.6200728

>
```


Key Programming concepts: Control Structures

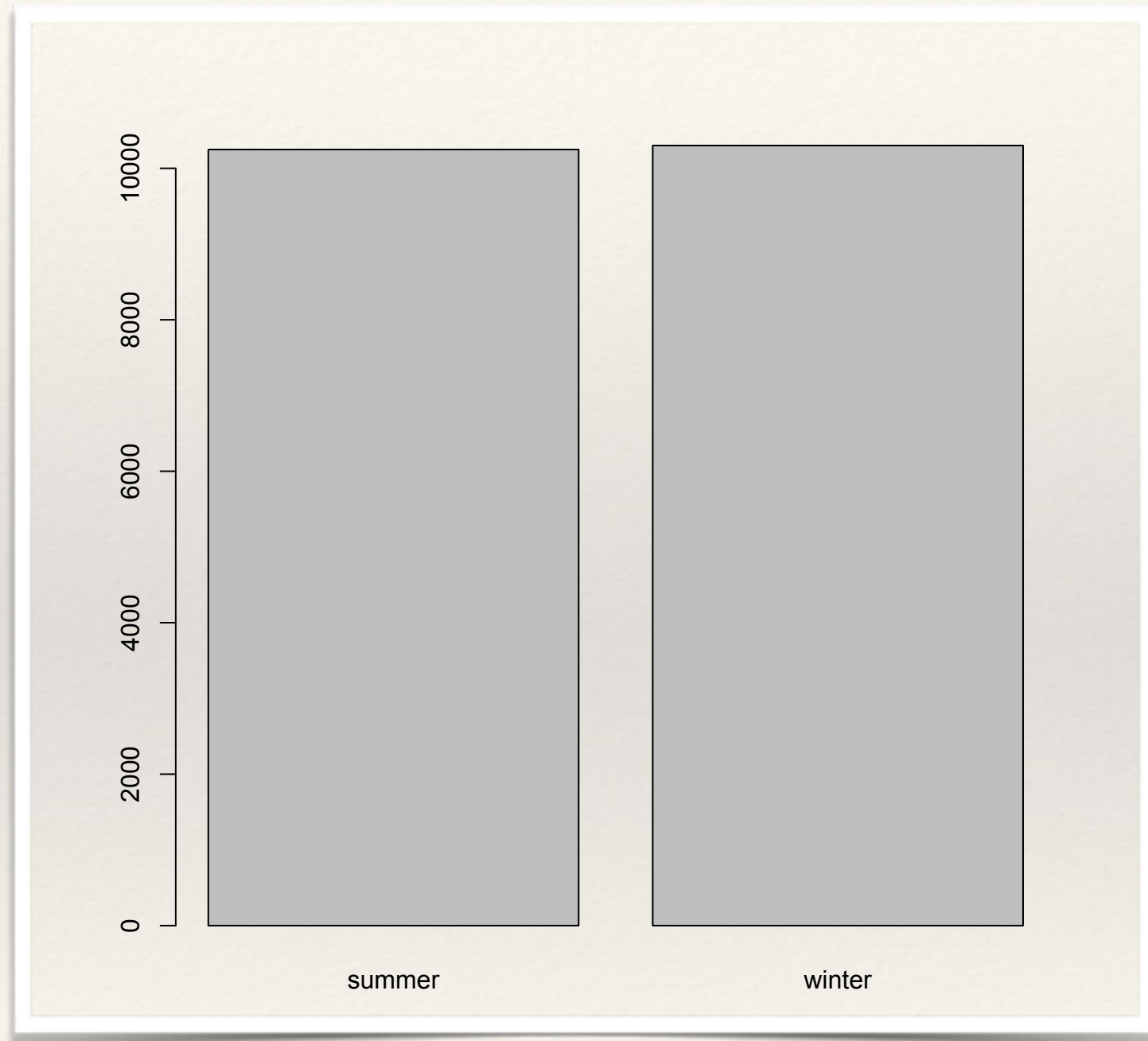
If can also be used to choose what you return from a function

```
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))  
}
```

```
>
```

```
> plot(as.factor(streamflow$season))
```

```
>
```



```
#' @param str data frame with columns month and mm (mm is daily streamflow)
compute_seasonal_meanflow = function(str) {

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

  tmp = subset(str, str$season=="winter")
  mean.winter = mean(tmp$mm)

  tmp = subset(str, str$season=="summer")
  mean.summer = mean(tmp$mm)
  return(list(summer=mean.summer, winter=mean.winter))
}
```

```
> compute_seasonal_meanflow(streamflow)
$summer
[1] 1.538304

$winter
[1] 0.6200728

>
```


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))  
}
```

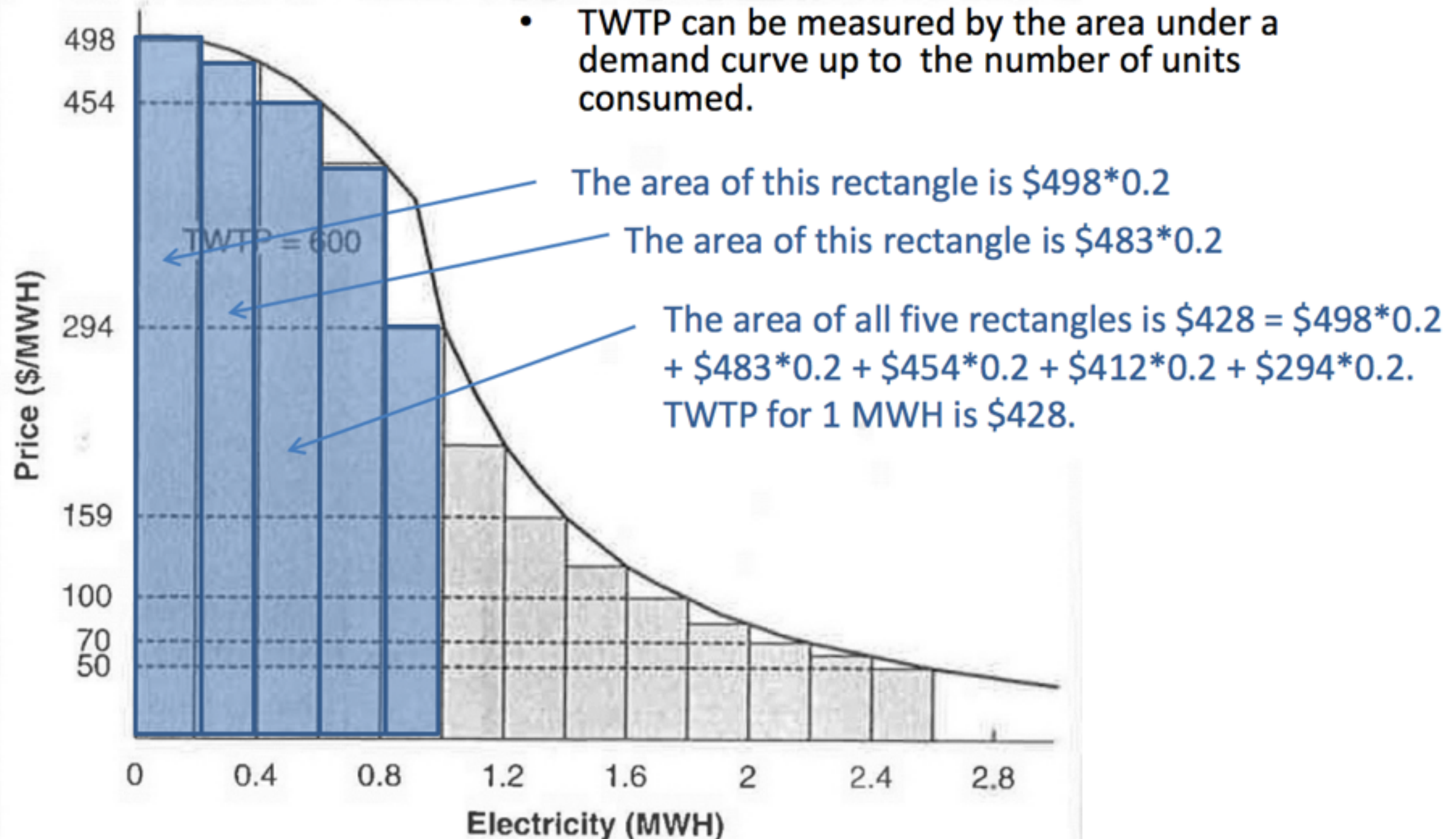
Key Programming concepts: Control Structures

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
```

Measuring TWTP

- TWTP can be measured by the area under a demand curve up to the number of units consumed.



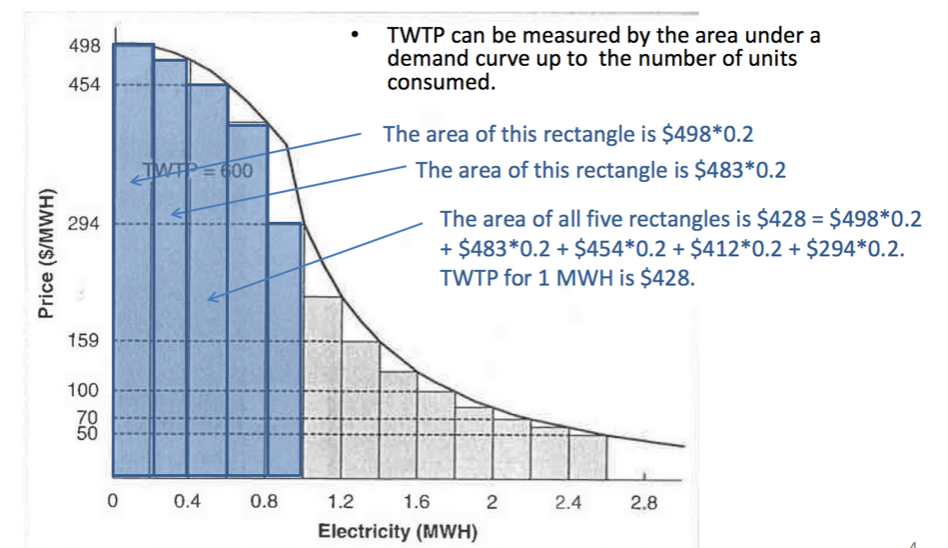
Example

- ❖ Lets say our demand curve is as follows:
- ❖ Electricity = c(0.1, 0.2, 0.4, 0.5, 0.6) MW
- ❖ Marginal willingness to pay = c(300,250,200,50, 20) \$ / MW
- ❖ Write a function that takes as input a data frame with values for marginal willingness to pay for different electricity values and returns total willingness to pay for a given electricity requirement (in MW)
- ❖ Take as input to the function a data frame called curve that has two columns elect, mwtop (electricity and marginal willingness to pay), and the electricity requirement

Key Programming concepts: Control Structures

```
compute_TWP = function(curve, req) {  
  
  total = 0  
  i=1  
  prev.elect = 0  
  while (( curve$select[i] <= req) && (i < length(curve$select))) {  
    total = total + (curve$select[i]-prev.elect)*curve$mwtop[i]  
    prev.elect = curve$select[i]  
    i = i+1  
  }  
  
  return(total)  
}
```

Measuring TWTP



```

compute_TWP = function(curve, req) {

total = 0
i=1
prev.elect = 0
while (( curve$elect[i] <= req) && (i < length(curve$elect))) {
total = total + (curve$elect[i]-prev.elect)*curve$mwtop[i]
prev.elect = curve$elect[i]
i = i+1
}

return(total)
}

```

```

>
> example.curve = data.frame(elect = c(0.1,0.2,0.4,0.5,0.6), mwtop=c(300,250,200,50,20))
> example.curve
  elect mwtop
1  0.1   300
2  0.2   250
3  0.4   200
4  0.5    50
5  0.6    20
> compute_TWP(example.curve, req=0.2)
[1] 55
> (0.1*300)+(0.1*250)
[1] 55
>

```



```
compute_TWP = function(curve, req) {
```

```
  total = 0
```

```
  i=1
```

```
  prev.elect = 0
```

```
  while (( curve$select[i] <= req) && (i < length(curve$select))) {
```

```
    total = total + (curve$select[i]-prev.elect)*curve$mwtop[i]
```

```
    prev.elect = curve$select[i]
```

```
    i = i+1
```

```
  }
```

```
  return(total)
```

```
}
```

```
> example.curve
```

```
  elect mwtop
```

```
1  0.1  300
```

```
2  0.2  250
```

```
3  0.4  200
```

```
4  0.5   50
```

```
5  0.6   20
```

```
> > compute_TWP(example.curve, req=0.7)
```

```
[1] 100
```

```
> compute_TWP(example.curve, req=0.6)
```

```
[1] 100
```

```
> compute_TWP(example.curve, req=0.9)
```

```
[1] 100
```

```
> compute_TWP(example.curve, req=0.3)
```

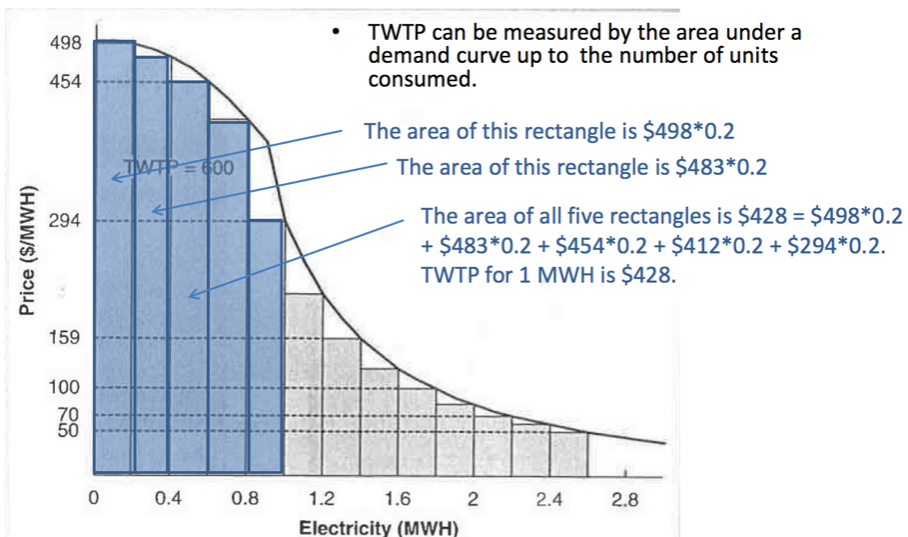
```
[1] 55
```

```
> compute_TWP(example.curve, req=0.4)
```

```
[1] 95
```

```
> >
```

Measuring TWTP



Example

- ❖ If the user asks for an electricity requirement beyond what was described by the demand curve, return NA from the function
- ❖ How would you implement that

```
compute_TWP = function(curve, req) {  
  
  total = 0  
  i=1  
  prev.elect = 0  
  while (( curve$select[i] <= req) && (i < length(curve$select))) {  
    total = total + (curve$select[i]-prev.elect)*curve$mwtop[i]  
    prev.elect = curve$select[i]  
    i = i+1  
  }  
  
  return(total)  
}
```

Example

```
compute_TWP = function(curve, req) {
```

```
  highest.elect = max(curve$select)
```

```
  if (req <= highest.elect) {
```

```
    total = 0
```

```
    i=1
```

```
    prev.elect = 0
```

```
    while (( curve$select[i] <= req) && (i < length(curve$select))) {
```

```
      total = total + (curve$select[i]-prev.elect)*curve$mwtop[i]
```

```
      prev.elect = curve$select[i]
```

```
      i = i+1
```

```
    }
```

```
  }
```

```
  else { total="NA" }
```

```
  return(total)
```

```
}
```

```
> example.curve
```

```
  elect mwtop
```

```
1  0.1  300
```

```
2  0.2  250
```

```
3  0.4  200
```

```
4  0.5   50
```

```
5  0.6   20
```

```
>
```

```
> compute_TWP(example.curve, req=0.4)
```

```
[1] 95
```

```
> compute_TWP(example.curve, req=0.9)
```

```
[1] "NA"
```