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
  - scripts that you save in R-Markdown
  - instructions to analyze relationships in census data and visualize them
  - a model of global climate

Programing fundamentals

- Programs can be written in many different languages (all have their strengths and weakness)
- Languages expect instructions in a particular form (syntax) and then translate them to be readable by the computer
- Languages have evolved to make it help users write programs that are easy to understand, re-use, extend, test, run quickly, use lots of data…

Programing fundamentals

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

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

- Modularity
  - breaking your instructions down into individual pieces
  - identifying instructions that can be reused
    - an ecosystem model might re-use instructions for calculating how a species grows
    - an accounting program might re-use instructions for computing net present value from interest rates
  - modules often become ‘black boxes’ which hides detail that might make understanding the program overly complex
  - most languages have lots of black boxes already written and most allow you to write your own

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/

Programming issues that you have had?

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.

STEPS: Program Design

1. **Clearly define your goal as precisely as possible, what do you want your program to do**
    1. **inputs/parameters/data**
    2. **outputs**
    3. **break into functional units (flow charts, conceptual diagrams)**
2. Implement and document
3. Test - Internal
4. Refine
5. Distribute
6. Test - Other users

STEPS: Levels of testing

1. **Unit testing**
2. **Integration Testing**
3. **System Testing**
4. **Acceptance Testing**
    -

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
-

Best practices for software development

- Structured practices that ensures
  - clear, readable code
  - modularity (organized “independent” building blocks)
  - testing as you go and after
  - code evolution is documented

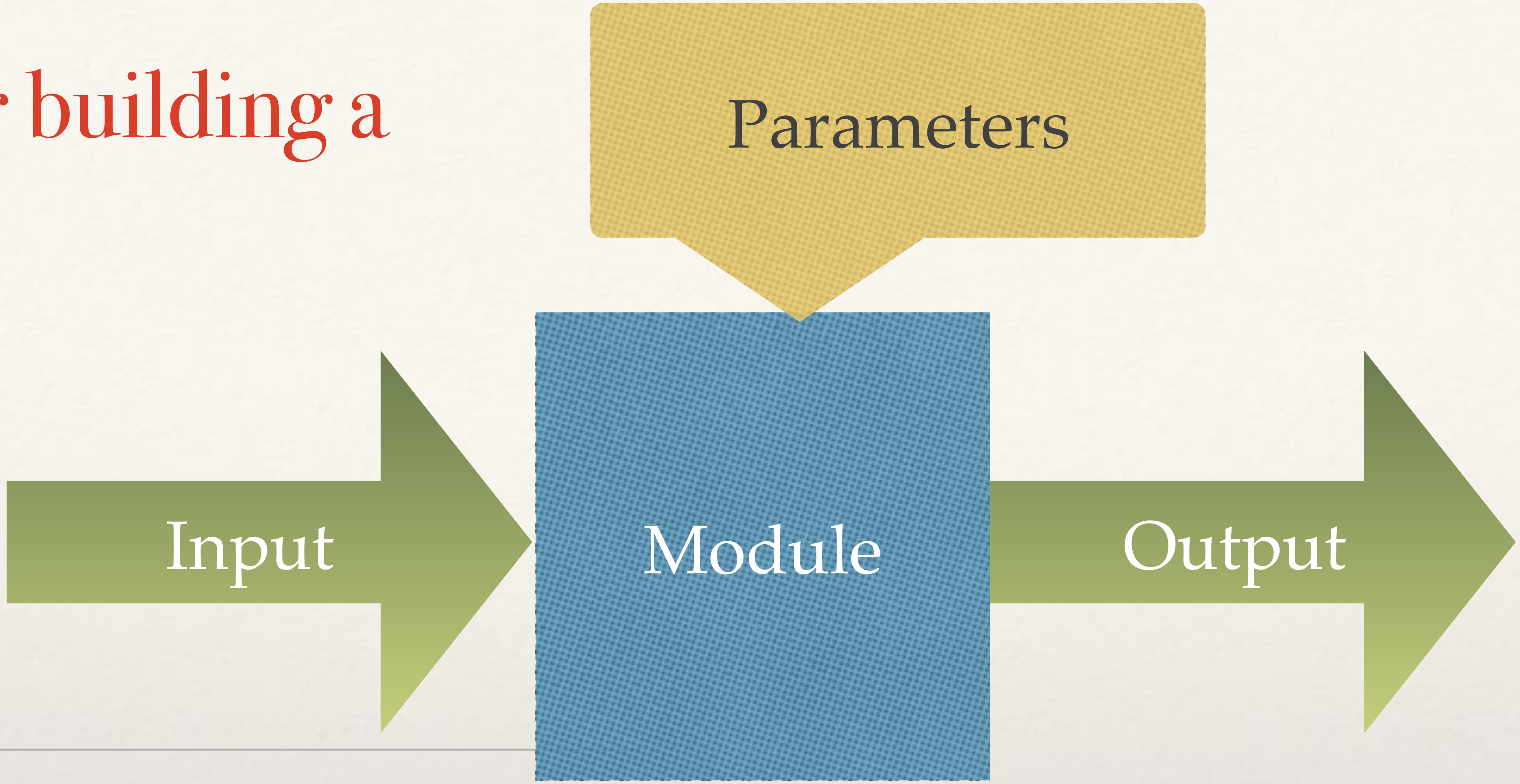
Inside Building Blocks

- Instructions inside the building blocks/box
    - Numeric data operators
        - +,-,/,*, %*%
    - Non-Numerica Data operations
        - searching,
        - concatenating
        - organizing
    - Math
        - sin, cos, exp, min, max…
        - these are themselves programs - boxes
    - Flow control
        - if else
        - looping
    - R-reference card is useful!

Building Blocks

- Functions (or objects or subroutines)!
- The basic building blocks
- Functions can be written in all languages; in many languages (object-oriented) like C++, Python, functions are also objects
- Functions are the “box” of the model - the transfer function that takes inputs and returns outputs
- More complex models - made up of multiple functions; and nested functions (functions that call/user other functions)
  - functions
  - main program that ‘calls’ the functions

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…

Functions

- Write down what the function will do - given different inputs and parameters
- simple
  - input (temperature); output (growth rate)
- more complex
  - inputs (temperature, organism type)
  - output (if animal, respiration; if plant, growth)

Designing Programs

- Inputs - sometimes separated into input data and parameters
    - input data = the "what" that is manipulated
    - parameters determine "how" the manipulation is done
    - "result = sort(BOD[,"demand"], decreasing=TRUE, method="quick")"
        - sort is the program - set of instructions - its a black box
        - input is BOD[,"demand"]
        - parameters are *decreasing* and *method*
        - output is a sorted version of saved to "*result*"
    - 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)

- Write a contract for a function to compute net present value
- Write a contract for a function to estimate the impact of pollution concentration on microbial biomass

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: $K_{Efficiency}$, ϱ (density of water), g (acceleration due to gravity)

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

P is Power in watts, ϱ is the density of water (~1000 kg/m3), h is height in meters, r is flow rate in cubic meters per second, g is acceleration due to gravity of 9.8 m/s2, $K_{Efficiency}$ is a coefficient of efficiency ranging from 0 to 1.

Building Models

- Inputs/parameters are height, flow, rho, g, and K
- For some (particularly parameters) we provide default values by assigning them a value (e.g Keff = 0.8), but we can overwrite these
- Body is the equations between { and }
- *return* tells R what the output is

```
power_gen = function(height, flow, rho=1000, g=9.8, Keff=0.8) {

result = rho * height * flow * g * Keff
return(result)
}
```

Building Models: Using the model

```
> power_gen(20,1)
[1] 156800
> power_gen(height=20,flow=1)
[1] 156800
> power.guess = power_gen(height=20,flow=1)
> power.guess
[1] 156800
> power.guess = power_gen(flow=1, height=20)
> power.guess
[1] 156800
```

Arguments to the function follow the order they are listed in your definition
Or you can specify which argument you are referring to when you call the program

```
power_gen = function(height, flow, rho=1000, g=9.8, K=0.8) {

# calculate power
result = rho * height * flow * g * K
return(result)
}
```

Building Models

- Always write your function in a text editor and then copy into R
- By convention we name files with functions in them by the name of the function.R
  - so power_gen.R
- you can have R read a text file by *source("power_gen.R")* - make sure you are in the right working directory
- Eventually we will want our function to be part of a package (a library of many functions) - to create a package you must use this convention (name.R)
- place all function in a directory called "**R**"

Building Models: Using the model

```
> power_gen(height=20, flow=1)
[1] 156800
> power_gen(height=20, flow=1, Keff=0.8)
[1] 156800
> power_gen(height=20, flow=1, Keff=0.5)
[1] 98000
> power_gen(height=10, flow=1, Keff=0.5)
[1] 49000
```

Defaults take the value they were assigned in the definition, but can be overwritten

```
power_gen = function(height, flow, rho=1000, g=9.8, Keff=0.8) {

# calculate power
result = rho * height * flow * g * Keff
return(result)
}
```

Scoping

The scope of a variable in a program defines where it can be “seen”

Variables defined inside a function cannot be “seen” outside of that function

There are advantages to this - the interior of the building block does not ‘interfere’ with other parts of the program

```
> power_gen
function(height, flow, rho=1000, g=9.8, Keff=0.8) {

   # calculate power
   result = rho * height * flow * g * K
   return(result)
}
> result
Error: object 'result' not found
> K
Error: object 'K' not found
>
```

Scoping

Build in R…example

Try

Write a function that returns net present value of a cost/profit in the future….

Functions: Error

- what will your function do if user gives you garbage data
- Two options
  - error-checking
    - if temperature < -100 or > 100, or NA, output warning
  - assume user reads the contract :)
    - return unrealistic values
    - so if input -999.0, will still try to output growth rate
- Error-checking is helpful if you are going to build a model made up of many functions- why?

Building Models

- Add error checking

```
power_gen = function(height, flow, rho=1000, g=9.8, Keff=0.8) {

  # make sure inputs are positive
    if (height < 0) return(NA)
    if (flow < 0) return(NA)
    if (rho < 0) return(NA)

  # calculate power
    result = rho * height * flow * g * Keff


  return(result)
}
```

Functions - R style

- Documentation style that allows automatic generation of help pages (we will get there)
- Save the function as a SEPARATE file - named by the name of the function.R (autopower.R)
- Don’t include steps to run the function in that document (put these in another R script file or R markdown)

One of the equations used to compute automobile fuel efficiency is as follows this is the power required to keep a car moving at a given speed

$$Pb = c_{rolling} * m *g*V + 1/2\ A*p_{air}*c_{drag}*V^3$$

where $c_{rolling}$ and c_{drag} are rolling and aerodynamic resistive coefficients, typical values are 0.015 and 0.3, respectively.
V: is vehicle speed (assuming no headwind) in m/s (or mps)
m: is vehicle mass in kg
A is surface area of car (m2)
g: is acceleration due to gravity (9.8 m/s2)
p_{air} = density of air (1.2kg/m3)
Pb is power in Watts

Write a function to compute power, given a truck of m=31752 kg (parameters for a heavy truck) for a range of different highway speeds
plot power as a function of speed
how does the curve change for a lighter vehicle

Note that 1mph=0.477m/s