BASICS OF PROGRAMMING

OUTLINE

1. Programming Fundamentals
2. Code Components
3. Designing Using Pseudo-Code
4. From Pseudo-Code to Runnable Code

PROGRAMMING FUNDAMENTALS

BASICS OF PROGRAMMING

GOAL AND LEARNING OBJECTIVES OF THIS SECTION

1. Provide you with fundamental concepts that you can apply to *any* programming language
2. Give you insight into what is common across all computer languages
3. Help you to *learn* any programming language, through reference to these common fundamentals
4. Prepare you for your first lab – when you will be designing and implementing computer programs in R and/or Python

GROUP DISCUSSION

What is computer code?

What is a computer program?

```c
#include <stdio.h>
int main()
{
        double firstNumber, secondNumber, temporaryVariable;

        printf("Enter first number: ");
        scanf("%lf", &firstNumber);

        printf("Enter second number: ");
        scanf("%lf",&secondNumber);

        // Value of firstNumber is assigned to temporaryVariable
        temporaryVariable = firstNumber;

        // Value of secondNumber is assigned to firstNumber
        firstNumber = secondNumber;

        // Value of temporaryVariable (which contains the initial value of firstNumber)
        secondNumber = temporaryVariable;

        printf("\nAfter swapping, firstNumber = %.2lf\n", firstNumber);
        printf("After swapping, secondNumber = %.2lf", secondNumber);

        return 0;
}
```

COMPUTER PROGRAM: EXAMPLE IN C

An algorithm written in a computer language, providing instructions to a computer for carrying out a series of operations

COMPUTER PROGRAM: DEFINITION

An **algorithm**, written in a **computer language**, that provides instructions to a computer for carrying out a **sequence** of **operations**.

It can be **compiled** or **interpreted** as a series of hardware operations, carried out by the **electrical components** of a computer.

SOME FUNDAMENTAL CONCEPTS

Algorithm

Computer Language

(Formal) Language

FORMAL LANGUAGE: EXAMPLE

Alphabet: {'a', 'b', 'C', 'D','!'}

Rules (Grammar):

- letters may be placed to the left or right of another letter
- a letter instance must always have another instance of the same letter to either the left or the right
- upper case letters must always have a lower case letter to the left or right

FORMAL LANGUAGE: DEFINITION

In a formal language:

- words are created from a pre-defined alphabet
- a grammar provides rules about how letters may be combined to form words

COMPUTER LANGUAGE: DEFINITION

A (formal) language constructed to provide instructions **to a computer**, such that it can be compiled into low-level instructions that the computer processor can **carry out**.

In the lexical and syntax rules given below, BNF notation characters are written in green.

- Alternatives are separated by vertical bars: i.e., '*a* | *b*' stands for "*a* **or** *b*".
- Square brackets indicate optionality: '[*a*]' stands for an optional *a*, i.e., "*a* | *epsilon*" (here, *epsilon* refers to the empty sequence).
- Curly braces indicate repetition: '{ *a* }' stands for "*epsilon* | *a* | *aa* | *aaa* | ..."

---

1. Lexical Rules

letter	::=	`a	b	...	z	A	B	...	Z`
digit	::=	`0	1	...	9`
id	::=	*letter* { *letter* | *digit* | _ }
intcon	::=	*digit* { *digit* }
charcon	::=	' *ch* ' | '**\n**' | '**\0**', where *ch* denotes any printable ASCII character, as specified by **isprint()**, other than **** (backslash) and **'** (single quote).
stringcon	::=	**"**{*ch*}**"**, where *ch* denotes any printable ASCII character (as specified by **isprint()**) other than **"** (double quotes) and the newline character.
Comments		Comments are as in C, i.e. a sequence of characters preceded by **/*** and followed by ***/**, and not containing any occurrence of ***/**.

COMPUTER LANGUAGE: FORMAL DEFINITION OF C

A language constructed to provide instructions to a computer

COMPUTER PROGRAM: DEFINITION

An **algorithm**, written in a **computer language**, that provides instructions to a computer for carrying out a **sequence** of **operations**.

It can be **compiled** or **interpreted** as a series of hardware operations, carried out by the **electrical components** of a computer.

ALGORITHM: EXAMPLE

1. Pour ½ cup flour into bowl
2. Break one egg into bowl
3. Pour 3 tablespoons oil into bowl
4. Pour 1 teaspoon baking powder into bowl
5. Mix with spoon until smooth
6. Pour mixture into muffin tins
7. Bake for 15 minutes at 350 degrees Fahrenheit

ALGORITHM: DEFINITION

A sequence of instructions which have one or more well defined stopping points.

COMPUTER PROGRAM: DETAILS

Higher level computer languages are compiled into (or interpreted as) ‘machine code’ – a series of very basic instructions that tell the computer hardware how to behave.

When the computer is carrying out the instructions, we say it is ‘running’ the program – as a **process**

We can instruct a computer to run a program. Computers can also tell themselves to run programs!

```
MONITOR FOR 6802 1.4          9-14-80   TSC ASSEMBLER   PAGE    2

 C000                       ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START       LDS    #STACK

                ****************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA      EQU    %00010011
 0011           CTLREG      EQU    %00010001

 C003 86 13     INITA       LDA A  #RESETA    RESET ACIA
 C005 B7 80 04              STA A  ACIA
 C008 86 11                 LDA A  #CTLREG    SET 8 BITS AND 2 STOP
 C00A B7 80 04              STA A  ACIA

 C00D 7E C0 F1              JMP    SIGNON     GO TO START OF MONITOR

                ****************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal
```

COMPUTER PROGRAMS: THE BIG PICTURE

All computers operate by running (compiled) computer programs.

The internet is a collection of computers connected by physical wires or radio wave transmitters and receivers.

Computers transmit to, and receive signals from, other computers on this network.

The signals sent from computer to computer, and what is done with received signals, are based on what programs the computers are running.

The cloud is a part of the internet loosely defined as a collection of computers used mainly to store and serve content to other computers.

CODE COMPONENTS

BASICS OF PROGRAMMING

GROUP DISCUSSION

What are the fundamental elements of computer code?

ELEMENTS OF COMPUTER CODE

Variables

Data Structures

Operators

Statements and Expressions

Blocks (and Scope)

Functions

Logical (Control) Flow

Libraries/Packages/Modules

Inputs/Outputs

Interpreters/Compilers

load additional functions (package/module/library) from outside the current code

```
library(igraph)
```

user-defined function with three arguments

variable

block

```
my_graph_function <- function(my_number_nodes, my_colour, my_density)
{

my_graph <- sample_gnp(my_number_nodes, my_density, directed = FALSE, loops = FALSE)

if(ecount(my_graph) >= my_number_nodes){V(my_graph)$color <-  my_colour}

plot(my_graph, layout=layout.fruchterman.reingold, vertex.color=V(my_graph)$color)

}
```

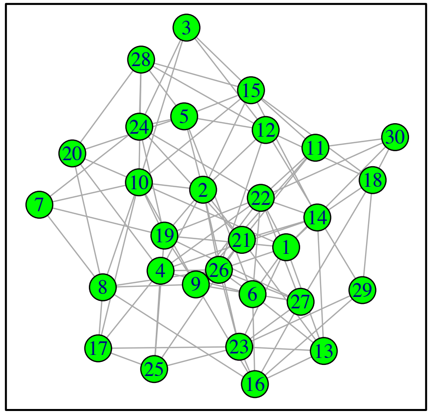
creating a data structure/object (a graph)

conditional logic statement (control flow)

```
my_graph_function(30,"green",0.3)
```

calling the user-defined function

generating output (a visualization of the graph)



READY TO START PROGRAMMING???

Not so fast!

DESIGNING CODE (USING PSEUDO-CODE)

BASICS OF PROGRAMMING

GROUP DISCUSSION

What does it mean to design an algorithm, or a program?

DESIGN COMPONENTS

In designing an algorithm, we need to specify:

- input
- output
- how the input should be transformed to provide the output

From a bigger picture perspective, we can also talk about the function, or purpose of the algorithm.

PSEUDO-CODE: EXAMPLE

```
j_cluster(array_of_points, max_n_neighbour_distance)
{
    for each point[i] in array_of_points
    {
        for each remaining point[j] in array_of_points
        {
            distance_between_ij = distance(point[i], point[j])
            if distance_between_ij <= max_n_neighbour_distance
            then neighbours[i] = add_to_neighbours(point[i],point[j])
        }
. . .
```

PSEUDOCODE: WHAT IT REALLY LOOKED LIKE!

i-cluster
min_neighbour_distance
~~DBSCAN~~ (array_of_points,)

for each point[i] in array_of_points
{
for each remaining point[j]
{
distance_between_ij = distance
(point[i], ~~remaining~~
}

}

if distance_between_ij ≤ max_neighbour_di
neighbours[i] = add_to_neighbours[point[i], p

}

⋮

PSEUDO-CODE: DESCRIPTION

Pseudo-code is the term for a rough sketch of an algorithm which indicates the general expected input, output and steps, but which 'black boxes' the details of the functions.

Keeping in mind the main elements of any computer language (e.g. variables, functions, logical flow, etc.) we can design an algorithm without using a specific language.

PSEUDO-CODE: STRATEGY

Define an input

Define an output

Write a set of programmatic instructions that will take you from input to output

Remember that you can 'black box' parts of the code – describing functionality at a high level

PSEUDO-CODE: LEVEL OF ABSTRACTION

Getting a feel for the right level of detail in pseudocode takes practice.

To some extent, it depends on the **level of abstraction** of the programming language you will (likely) be using:

- High-level languages – have a lot of built in functions
- Low-level languages – many details and functions must be programmed 'by hand'

High-level languages let you program at a higher level of abstraction.

At the same time, you may sacrifice utility for understanding.

EXERCISE IN PSEUDO-CODE AND ALGORITHM DESIGN: SORTING

Your input is a list of numbers in unknown order

Your output is the same list of numbers sorted in the right order

Write pseudo-code that will take you from input to output

Remember that you can 'black box' parts of the code – describing functionality at a high level

In terms of level of detail – take into account the manipulation of individual numbers or groups of numbers within the list.

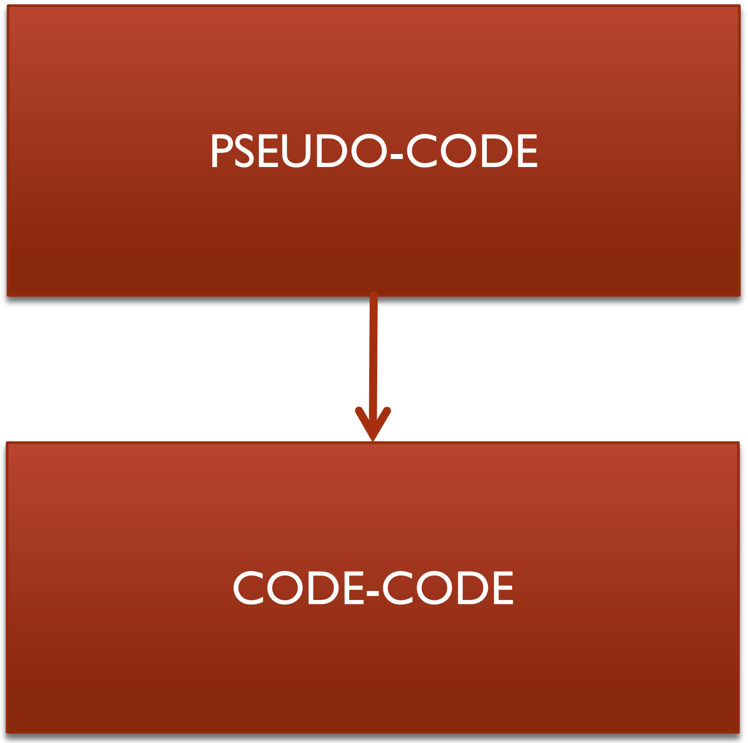
FROM PSEUDO-CODE TO RUNNABLE CODE

BASICS OF PROGRAMMING

THE REAL DEAL

To go from pseudo-code to real code, there are a number of steps:

- Determine the appropriate syntax of the language you want to use and rewrite your pseudo-code as real code in this language
- Replace black box functions with real code
- Determine how to connect your piece of code (the software) up to the computer, so your code can be compiled/interpreted, run by the computer, receive input and generate output



FROM CODE TO COMPUTER

Many **roadblocks** can arise when going from code – which is just text files – to having code that runs on your computer. These roadblocks can include:

- libraries
- input/output + file system
- compilers/interpreters

In broad terms, a certain amount of infrastructure must be in place!

We are taking care of much of this for you by setting up notebooks for you.

PROGRAMMING RESOURCES

Much of the information about how to use a particular computer language or how to make code run on a particular hardware configuration, **is not written down** in any single **authoritative reference manual**.

This is likely because the world of coding and computers changes so quickly.

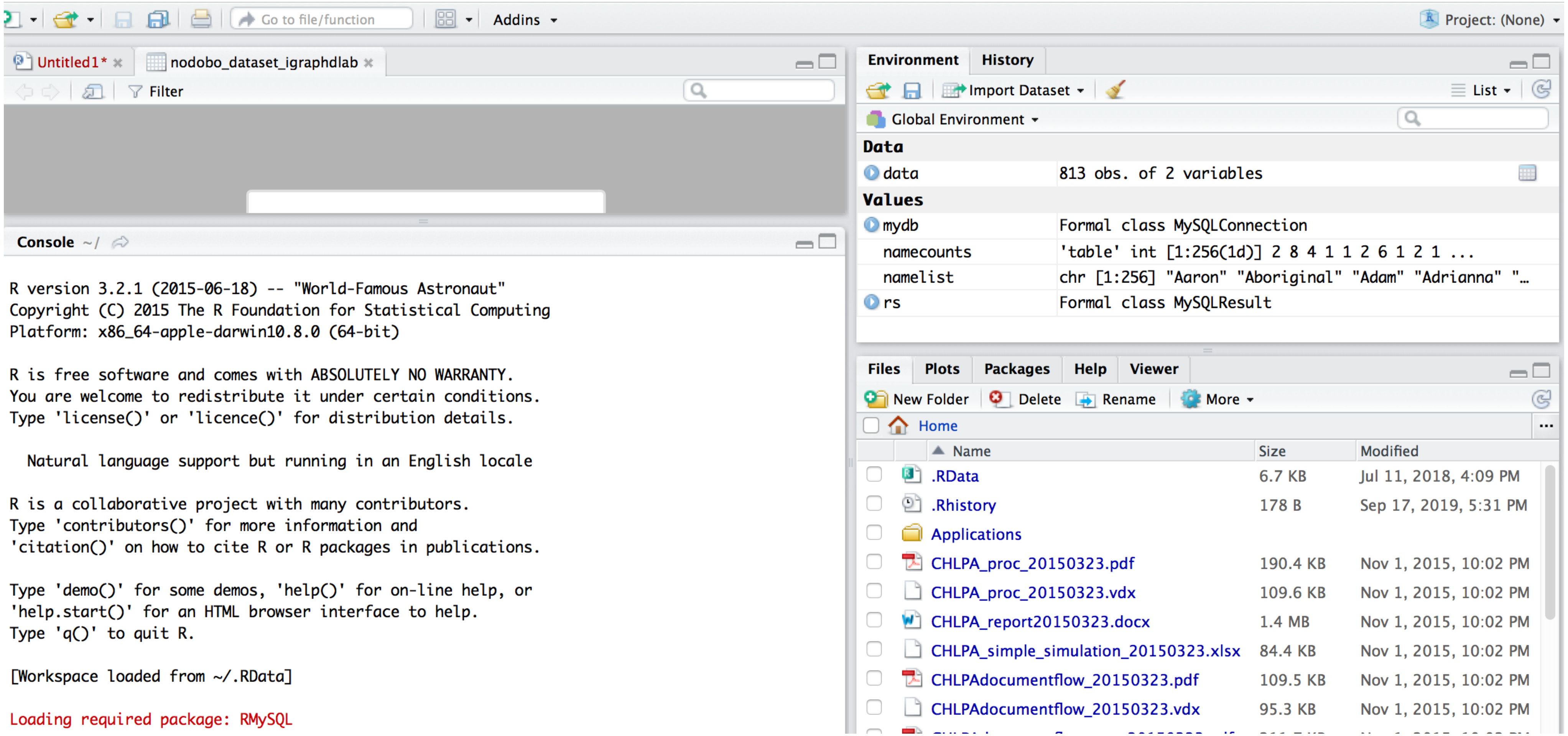
To successfully code, you must be **embedded in a community of coders**. Luckily, the internet has made this much easier – most questions about coding have already been answered somewhere on the internet.

In short – **STACK EXCHANGE** (and other similar sites)

SORTING ALGORITHM: A SKETCH IN R

Your challenge: Using the provided R notebook, write (in R) and run a program that sorts numbers.

R STUDIO



Go to file/function
Addins
Project: (None)
Untitled1*
nodobo_dataset_igraphdlab
Filter
Console ~/
R version 3.2.1 (2015-06-18) -- "World-Famous Astronaut"
Copyright (C) 2015 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.
[Workspace loaded from ~/.RData]
Loading required package: RMySQL
Environment
History
Import Dataset
List
Global Environment
Data
data 813 obs. of 2 variables
Values
mydb Formal class MySQLConnection
namecounts 'table' int [1:256(1d)] 2 8 4 1 1 2 6 1 2 1 ...
namelist chr [1:256] "Aaron" "Aboriginal" "Adam" "Adrianna" "…
rs Formal class MySQLResult
Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
Home
Name
Size
Modified
.RData 6.7 KB Jul 11, 2018, 4:09 PM
.Rhistory 178 B Sep 17, 2019, 5:31 PM
Applications
CHLPA_proc_20150323.pdf 190.4 KB Nov 1, 2015, 10:02 PM
CHLPA_proc_20150323.vdx 109.6 KB Nov 1, 2015, 10:02 PM
CHLPA_report20150323.docx 1.4 MB Nov 1, 2015, 10:02 PM
CHLPA_simple_simulation_20150323.xlsx 84.4 KB Nov 1, 2015, 10:02 PM
CHLPAdocumentflow_20150323.pdf 109.5 KB Nov 1, 2015, 10:02 PM
CHLPAdocumentflow_20150323.vdx 95.3 KB Nov 1, 2015, 10:02 PM

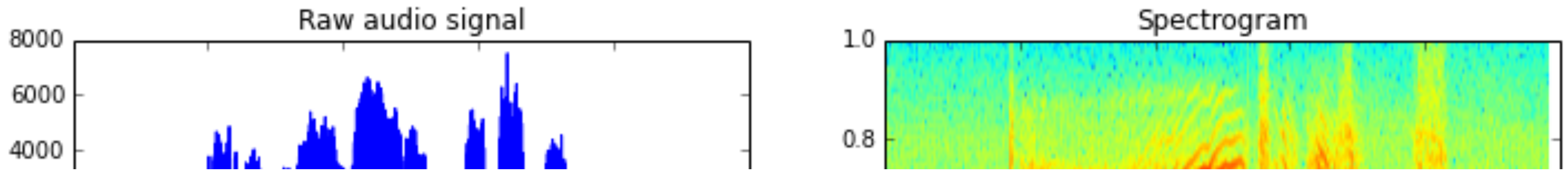
$$X_k = \sum_{n=0} x_n e^{-\frac{2\pi i}{N}kn} \qquad k = 0, \dots, N-1$$

We begin by loading a datafile using SciPy's audio file support:

In [1]:
```python
from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

In [2]:
```python
%matplotlib inline
from matplotlib import pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```



JUPYTER NOTEBOOK ANATOMY

COMPONENTS OF R COMPUTER CODE

- Variables
- Data Structures
- Operators
- Statements and Expressions
- Blocks (and Scope)
- Functions
- Logical (Control) Flow
- Libraries/Packages/Modules
- Inputs/Outputs
- Interpreters/Compilers

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07
Granted to the public domain. See www.Rpad.org for the source and latest version. Includes material from *R for Beginners* by Emmanuel Paradis (with permission).

Getting help

Most R functions have online documentation.

`help(topic)` documentation on `topic`
`?topic` id.
`help.search("topic")` search the help system
`apropos("topic")` the names of all objects in the search list matching the regular expression "topic"
`help.start()` start the HTML version of help
`str(a)` display the internal *str*ucture of an R object
`summary(a)` gives a "summary" of `a`, usually a statistical summary but it is *generic* meaning it has different operations for different classes of `a`
`ls()` show objects in the search path; specify `pat="pat"` to search on a pattern
`ls.str()` str() for each variable in the search path
`dir()` show files in the current directory
`methods(a)` shows S3 methods of `a`
`methods(class=class(a))` lists all the methods to handle objects of class `a`

R: SOME KEY INFO (I)

To create a **variable** in R, simply come up with a name and use the assignment operator to assign a value to the variable

The value might be of variable types: number, character, string, vector, list, matrix, data frame or some other object

R uses the data frame object a lot!

```
> my_number <- 5
> my_string <- "Jen"
> my_vector <- c(1,2,3,4)
> my_list <- list(1,2,3,4)
> my_data_frame <-
data.frame(c("Jen","Pat"),c(4,2),c(6,10))
> colnames(my_data_frame) <-
c("Name","Shoe_Size","Score")
> my_data_frame
  Name Shoe_Size Score
1  Jen         4     6
2  Pat         2    10
```

OBJECT ORIENTED VS PROCEDURAL LANGUAGES

R and Python are objected oriented languages, as opposed to procedural languages.

What does this mean?

To understand the answer we must first understand:

- Data Types
- Data structures
- Functions

DATA TYPES

Languages have a set of built in basic variable types – e.g.:

- Integer: 5
- Character: ‘m’
- List: (5, 3, 9)

Other variables types can be built up out of these basic types – e.g.:

- String = list of characters: (‘t’, ‘a’, ‘b’, ‘l’, ‘e’)

DATA STRUCTURES AND OBJECTS

A user might want to define their own set of related variables – a data structure:

- struct myNames = {string firstName, string middleName, string lastName}
- jenNames might be a variable of type myNames, with firstName = Jen, middleName = Adele, lastName = Schellinck

In addition a programmer might want to always be able to carry out a set of predefined instructions, or functions, on that data structure:

- jenNames.print_middle_name

An object is **loosely** defined as a user defined data structure plus a set of functions that goes along with that data structure.

R DATA FRAMES

The data frame **object** in R is structured similar to a spreadsheet in Excel:

- It has rows and columns, with associated row and column names
- You can carry out predefined operations on specific values, on selected rows or selected columns

People familiar working with databases, AND people used to working with more vector-focused languages (e.g. Java) might find the data frame implementation in R frustrating!

SORTING ALGORITHM: A SKETCH IN R

Your challenge: using the provided R notebook, write (in R) and run a program that sorts numbers.

SORTING ALGORITHM: A SKETCH IN PYTHON

Your challenge: using the provided Python notebook, write (in Python) a program that sorts numbers.

SOME USEFUL ADDITIONAL DETAILS

COMPILED VS INTERPRETED LANGUAGES

Compiled Language: Program is written as a whole, compiler checks the code **as a whole** and turns it into a low level language

Interpreted Language: Interpreter reads, turns into low-level code, and carries out **one statement at a time**.

Using an interpreter lets you program in a more free-form, improvisational way – like playing jazz instead of classical music.

This can be useful if you are doing exploratory work, but you can run into trouble if you use this strategy to generate larger or more substantial programs.

DEBUGGING

Debugging is mostly about revealing what is in memory at different points in the control flow of the code – is the code doing what you think it is?

Debugging is a bit of an art form

Debugging requires you to be a detective

Debugging teaches you perseverance

There are debugging tools that can help you all of this

SOME RELEVANT COMPUTER SCIENCE FRAMEWORKS

Languages (Computer, Mark Up)

Libraries/APIs

Software (Applications, Utilities, Systems)

Code (Open-Source, Uncompiled)

Protocol/Standard

Programming Models/Styles

OPTIONAL EXERCISES AND READINGS

BASICS OF PROGRAMMING

EXERCISES: LABS COMING UP!

In the next session, you will have the opportunity learn more about R and Python and to run some **pre-made** notebooks.

Starting in February, you will be having labs, where **the programming responsibility will fall to you**.

Prior to these labs, for out-of-class work we encourage you to:

- Review the 'Introduction to R' notebook
- Review the introductory Python notebooks
- Try writing R and Python statements and 'code snippets' in the Notebook environment, in order to become comfortable with R and Python

OPTIONAL EXERCISES: TRY IT OUT

Starting from an empty (new) notebook:

- Load some data from a file into a data frame
- Create a plot or graph using some of the loaded data

Note – using code from other notebooks or from on-line resources is **not** cheating!

REFERENCES

BASICS OF PROGRAMMING

REFERENCES

Sample of C language specifications: https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/cminusminusspec.html

C code snippet: https://www.programiz.com/c-programming/examples/swapping

Cheat Sheet for Jupyter Notebooks: https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Jupyter_Notebook_Cheat_Sheet.pdf

Cheat sheet for R: https://cran.r-project.org/doc/contrib/Short-refcard.pdf

Cheat sheet for Markdown: https://scottboms.com/downloads/documentation/markdown_cheatsheet.pdf

IMAGES

Signpost: https://upload.wikimedia.org/wikipedia/commons/2/29/Worden_park_signpost.jpg

Muffins: https://commons.wikimedia.org/wiki/Category:Muffins#/media/File:Sweet_potato_pecan.jpg

Assembly Language: https://en.wikipedia.org/wiki/Assembly_language#/media/File:Motorola_6800_Assembly_Language.png

Internet Map: https://upload.wikimedia.org/wikipedia/commons/d/d2/Internet_map_1024.jpg