Week 1 – Introduction

This week we will focus on the following:

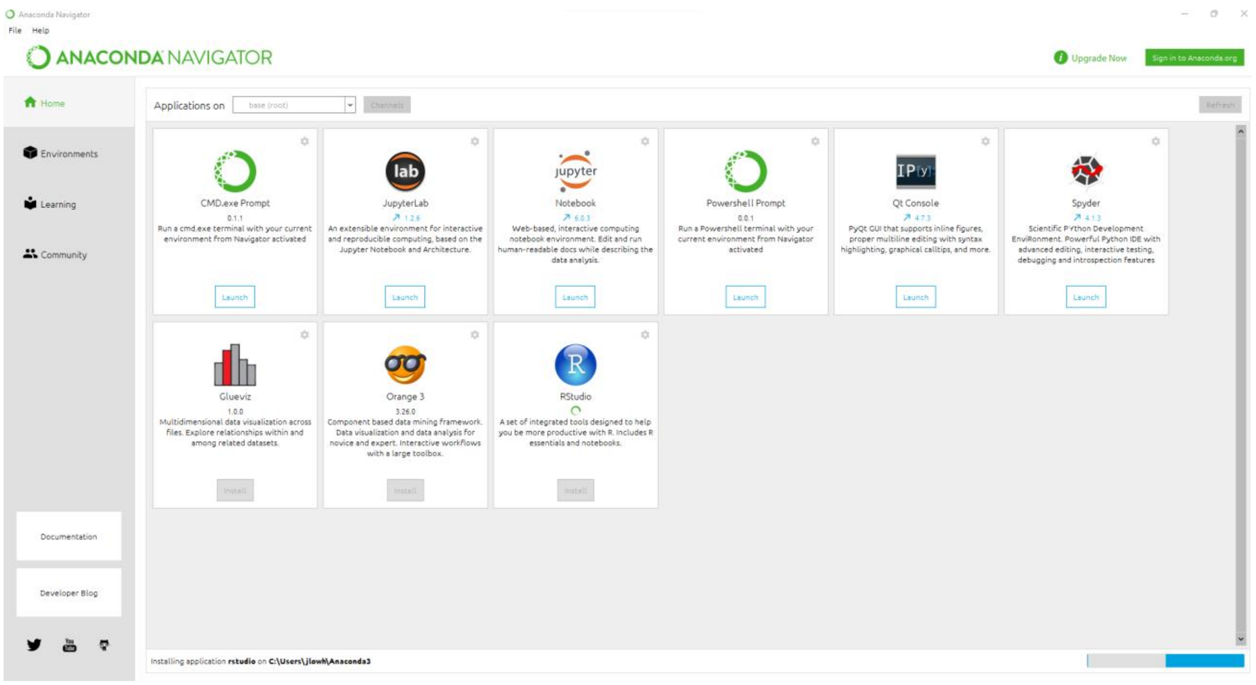
1. Installing R & RStudio
2. Version Control & GitHub
3. Working Directories
4. R Markdown
5. R Environments & scoping

Installing R & RStudio

In the first four weeks of class we will be learning the R programming language. We will first need to download the language itself and then download the most widely used IDE for R development, RStudio.

There are several ways to do this, but the most efficient for this class will be to install Anaconda which ships with everything we will need for both R and Python. Anaconda can be installed from here: Anaconda. They provide installers for Windows, MacOS, and Linux based systems so this should work for everyone.

Once you have Anaconda installed you can simply install RStudio which will ship with R (the language itself)



CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated

JupyterLab
1.2.6
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook
6.0.3
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Powershell Prompt
0.0.1
Run a Powershell terminal with your current environment from Navigator activated

Qt Console
4.7.3
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Spyder
4.1.3
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Glueviz
1.0.0
Multidimensional data visualization across files. Explore relationships within and among related datasets.

Orange 3
3.26.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

RStudio
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Installing application **rstudio** on **C:\Users\jlowh\Anaconda3**

If you wish to do your own installation of R and RStudio you will first need to install R from CRAN. CRAN

And then install RStudio Desktop from: RStudio

Mac Specific:

I have seen folks have issues with Mac installations due to not having xcode-select.

To install this prerequisite, you will need to open your Mac terminal and run: xcode-select –install

If you don't know how to open your terminal, you can do either of the following:

- Click the Launchpad icon in the Dock, type Terminal in the search field, then click Terminal.
- In the Finder, open the /Applications/Utilities folder, then double-click Terminal.

Version Control/GitHub

We will be using GitHub in this course to organize and store all your work. Most professional organizations use GitHub enterprise to version control their software and store their code off-prem. For this reason, it's a good idea to begin learning the Git paradigm and structing your work in organized directory structures.

All the course materials for this class will be stored on the GitHub. You will need to fork the repository and clone it to your local machine. Once you have it locally you can begin to complete the assignments and push it back to your GitHub. I expect you to turn everything in through GitHub.

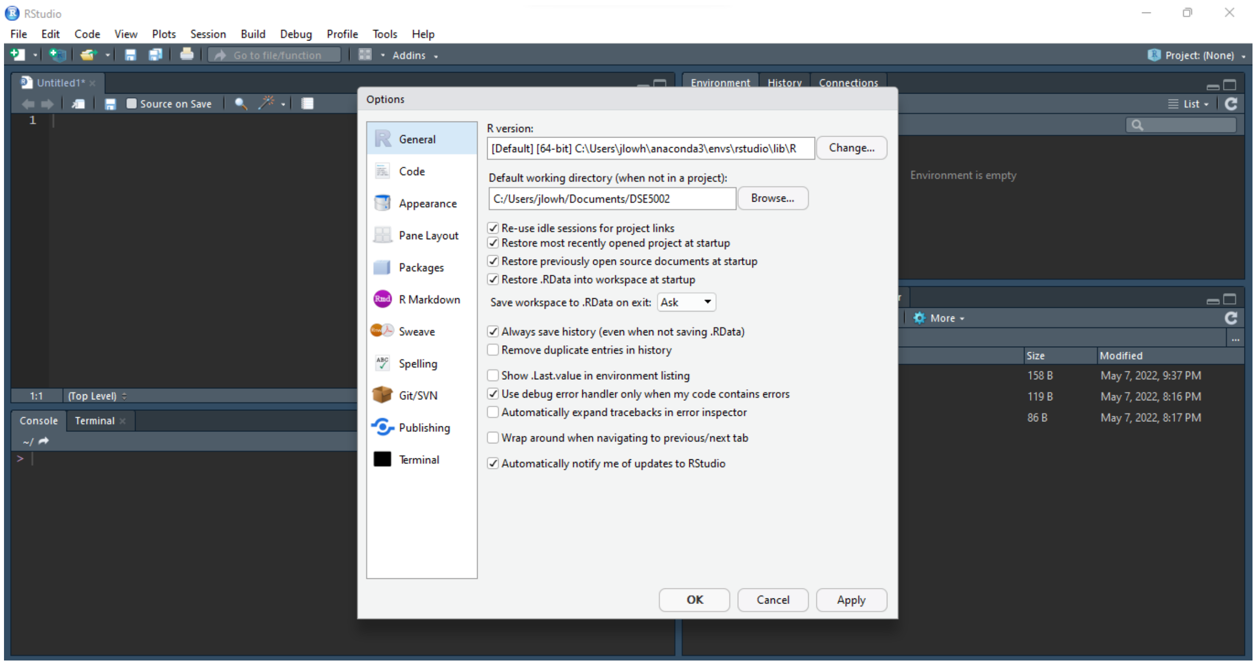
Steps for getting setup:

1. To begin create an account at: github
2. For my repository for DSE 5002
3. Once you have a GitHub account you will need to install GitHub Desktop from here: GitHub Desktop
4. Install GitHub Desktop and link it to your account
5. Clone DSE5002 to your local drive
6. Finish your assignments and push the completed work back to GitHub

Working Directories

A working directory is the ‘folder’ in which R sets as the ‘root’ directory for your project. In this class we will work out of a structured directory structure that I have already setup for you. There is a folder for each week of class containing everything you need to complete the assignments, project, and follow along with the class materials.

Once you have cloned DSE5002 to your local computer you will need to modify the default working directory in R to point to your DSE5002 folder. After you have applied those changes, you will need to restart RStudio for them to take effect.



R version:
[Default] [64-bit] C:\Users\jlowh\anaconda3\envs\rstudio\lib\R

Default working directory (when not in a project):
C:/Users/jlowh/Documents/DSE5002

- [✓] Re-use idle sessions for project links
- [✓] Restore most recently opened project at startup
- [✓] Restore previously open source documents at startup
- [✓] Restore .RData into workspace at startup

Save workspace to .RData on exit: Ask

- [✓] Always save history (even when not saving .RData)
- [ ] Remove duplicate entries in history
- [ ] Show .Last.value in environment listing
- [✓] Use debug error handler only when my code contains errors
- [ ] Automatically expand tracebacks in error inspector
- [ ] Wrap around when navigating to previous/next tab
- [✓] Automatically notify me of updates to RStudio

You should be able to verify the directory has changed by running the following command in the console of RStudio:

```
> getwd()
[1] "C:/Users/jlowh/Documents/DSE5002"
```

R Markdown

In this course and subsequent courses, you will be required to provide markdown documents knitted to PDF for your assignments. R Markdown is a file format for making dynamic documents with R. An R Markdown document is written in markdown (an easy-to-write plain text format) and contains chunks of embedded R code. To convert R Markdown documents (Knit) to PDF you will need a software packaged called MiKTeX. Install the full version at your own peril. Luckily, we can download a much smaller version specifically for R by running the following command in the R console:

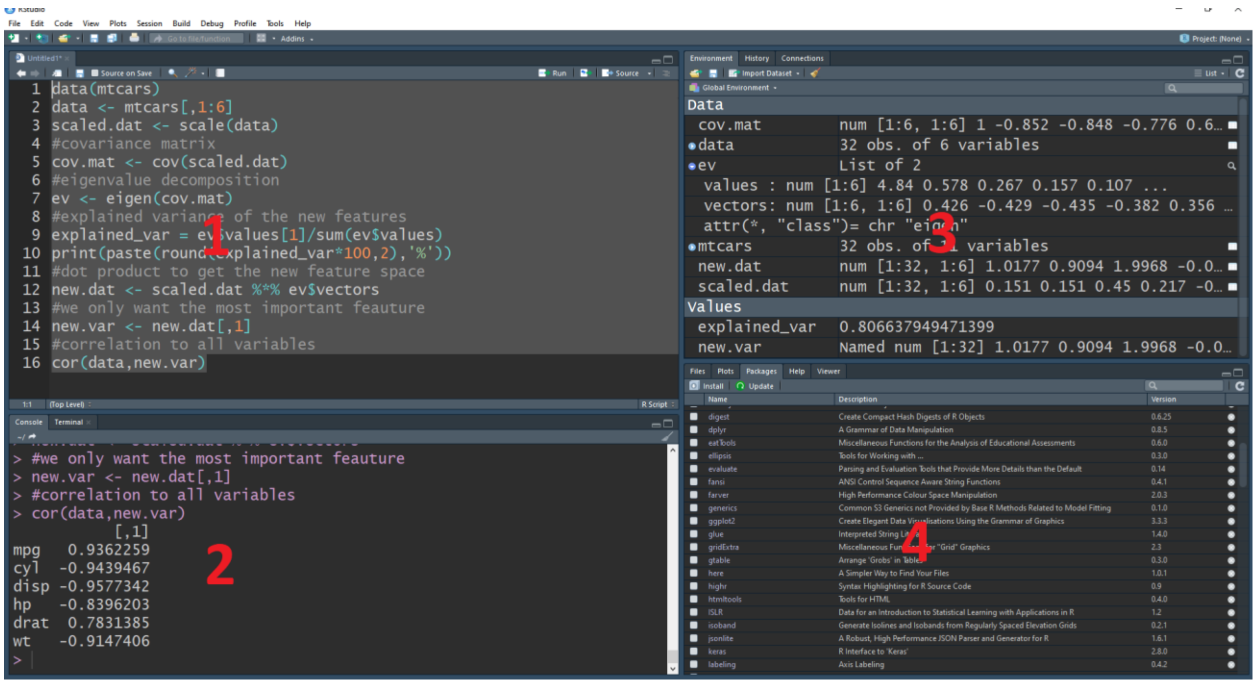
tinytex::install_tinytex()

You are expected to turn in organized markdown documents for this course. That means you need to prepare your code in a neat, organized manner with descriptions on what you are doing in the code. I would suggest when you are solving a particular coding problem to first wrangle the problem in an R script and move it to your markdown document AFTER you have the code in a manageable format. Think of your markdown document as your final draft for your code, not a place to experiment. Its also important to note that anything you do in the other script will need to be replicated in your markdown document. So, if you create a variable CREATE IT IN THE MARKDOWN DOCUMENT. This is often overlooked and causes a great deal of frustration for students. The reason this happens is due to scoping which we will get into next!

Please watch the video for a walkthrough of R Markdown.

R Environments & Scoping

To begin working in R you will need a basic understanding of the IDE, RStudio and how it is broken out into the four components shown in the below picture.


File Edit Code View Plots Session Build Debug Profile Tools Help
Untitled1*
Source on Save
Run
Source
Environment History Connections
Import Dataset
Global Environment
Console Terminal
Files Plots Packages Help Viewer
Install Update
1
2
3
4

```
data(mtcars)
data <- mtcars[,1:6]
scaled.dat <- scale(data)
#covariance matrix
cov.mat <- cov(scaled.dat)
#eigenvalue decomposition
ev <- eigen(cov.mat)
#explained variance of the new features
explained_var = ev$values[1]/sum(ev$values)
print(paste(round(explained_var*100,2),'%'))
#dot product to get the new feature space
new.dat <- scaled.dat %*% ev$vectors
#we only want the most important feauture
new.var <- new.dat[,1]
#correlation to all variables
cor(data,new.var)
```

```
> #we only want the most important feauture
> new.var <- new.dat[,1]
> #correlation to all variables
> cor(data,new.var)
           [,1]
mpg   0.9362259
cyl  -0.9439467
disp -0.9577342
hp   -0.8396203
drat  0.7831385
wt   -0.9147406
>
```

```
Data
  cov.mat        num [1:6, 1:6] 1 -0.852 -0.848 -0.776 0.6...
 data            32 obs. of 6 variables
 ev              List of 2
   values : num [1:6] 4.84 0.578 0.267 0.157 0.107 ...
   vectors: num [1:6, 1:6] 0.426 -0.429 -0.435 -0.382 0.356 ...
   attr(*, "class")= chr "eigen"
 mtcars          32 obs. of 11 variables
  new.dat        num [1:32, 1:6] 1.0177 0.9094 1.9968 -0.0...
  scaled.dat     num [1:32, 1:6] 0.151 0.151 0.45 0.217 -0...
Values
  explained_var  0.806637949471399
  new.var        Named num [1:32] 1.0177 0.9094 1.9968 -0.0...
```

Name	Description	Version
digest	Create Compact Hash Digests of R Objects	0.6.25
dplyr	A Grammar of Data Manipulation	0.8.5
eatTools	Miscellaneous Functions for the Analysis of Educational Assessments	0.6.0
ellipsis	Tools for Working with ...	0.3.0
evaluate	Parsing and Evaluation Tools that Provide More Details than the Default	0.14
fansi	ANSI Control Sequence Aware String Functions	0.4.1
farver	High Performance Colour Space Manipulation	2.0.3
generics	Common S3 Generics not Provided by Base R Methods Related to Model Fitting	0.1.0
ggplot2	Create Elegant Data Visualisations Using the Grammar of Graphics	3.3.3
glue	Interpreted String Literals	1.4.0
gridExtra	Miscellaneous Functions for "Grid" Graphics	2.3
gtable	Arrange 'Grobs' in Tables	0.3.0
here	A Simpler Way to Find Your Files	1.0.1
highr	Syntax Highlighting for R Source Code	0.9
htmltools	Tools for HTML	0.4.0
ISLR	Data for an Introduction to Statistical Learning with Applications in R	1.2
isoband	Generate Isolines and Isobands from Regularly Spaced Elevation Grids	0.2.1
jsonlite	A Robust, High Performance JSON Parser and Generator for R	1.6.1
keras	R Interface to 'Keras'	2.8.0
labeling	Axis Labeling	0.4.2

What are these components? Let's look at what each is, and how you can use it.

1) When you have a script, Markdown file, or text file, or any other file open in R it is displayed in this panel. Typically, this is where you write your code/scripts. In the screen shot above you see some code I have written to complete a principal components analysis of the mtcars data set. When ran, this code is executed in the second panel.
2) This is the console, or where your code executes. As mentioned previously, when I run the code from panel 1 it will execute the code and print any results that I have asked it to return. Here we see the correlation of the first principal component of our analysis to each of the individual variables in mtcars. There is also a terminal to the local filesystem, although the usage of that is outside of the scope of this class. The variables or objects that we defined in our script can be viewed in our global environment (because we haven't used any special scoping) in panel 3.
3) Panel 3 includes any objects you have in memory (from executed code) and displays their type, size, and in some cases other special metadata about the objects and classes. The broom looking button with remove ALL objects from your global environment.
4) Panel 4 contains several different options for you to view Files, Plots, Packages, get Help, or use the Viewer. If you are looking to install a package, this is the easiest place to do it. The file browser will give you an easy way to browse your filesystem and the Help tab will allow you to search for functions and determine their usage patterns.

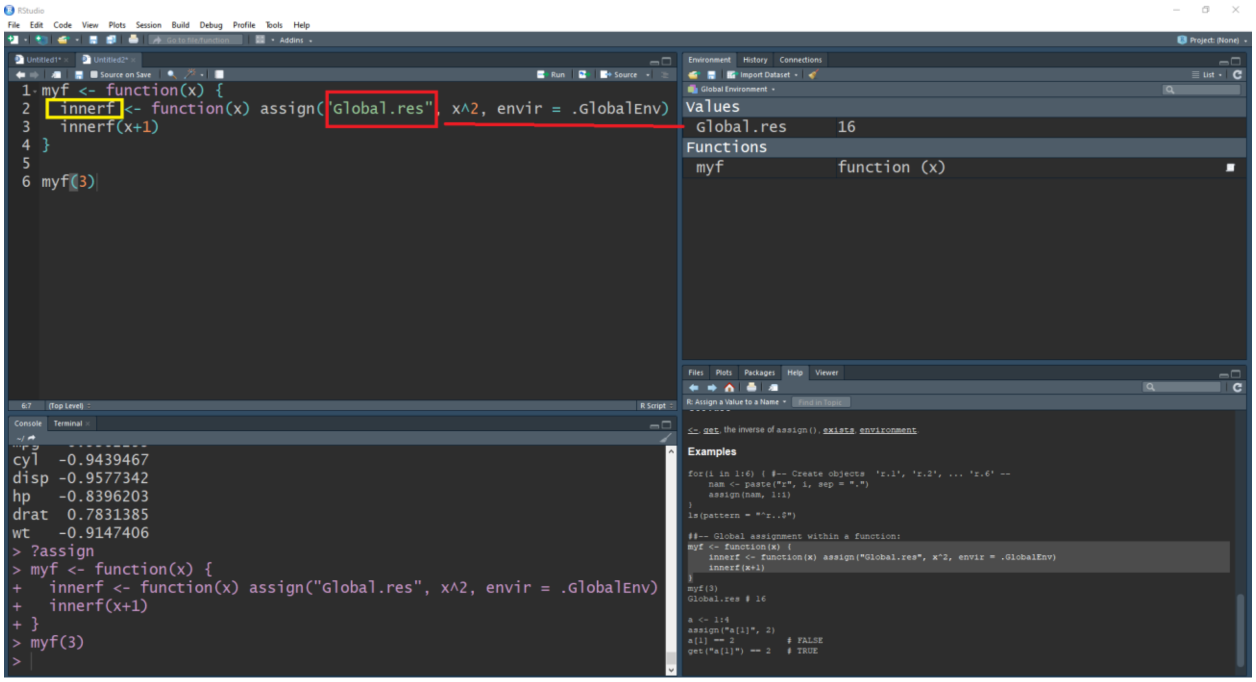
So, what's an environment and what's scoping?

An environment **is a collection of objects (functions, variables, etc.)**. An environment is created when we first start the R interpreter (open RStudio and have a running console).

The top-level environment is called the global environment and that is what is shown in panel 3. There can be local environments that are scoped within functions and classes and whose objects are not available in the global environment. Typically, when you write a function the object that is returned is contained in a return statement. All other variables will not be available unless you **assign()** them.

Example 1:

Let’s look at the following example:



```
myf <- function(x) {
  innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
  innerf(x+1)
}

myf(3)
```

```
Values
  Global.res    16
Functions
  myf           function (x)
```

```
cyl  -0.9439467
disp -0.9577342
hp   -0.8396203
drat  0.7831385
wt   -0.9147406
> ?assign
> myf <- function(x) {
+   innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
+   innerf(x+1)
+ }
> myf(3)
>
```

R: Assign a Value to a Name

<-, get, the inverse of assign(), exists, environment.

Examples

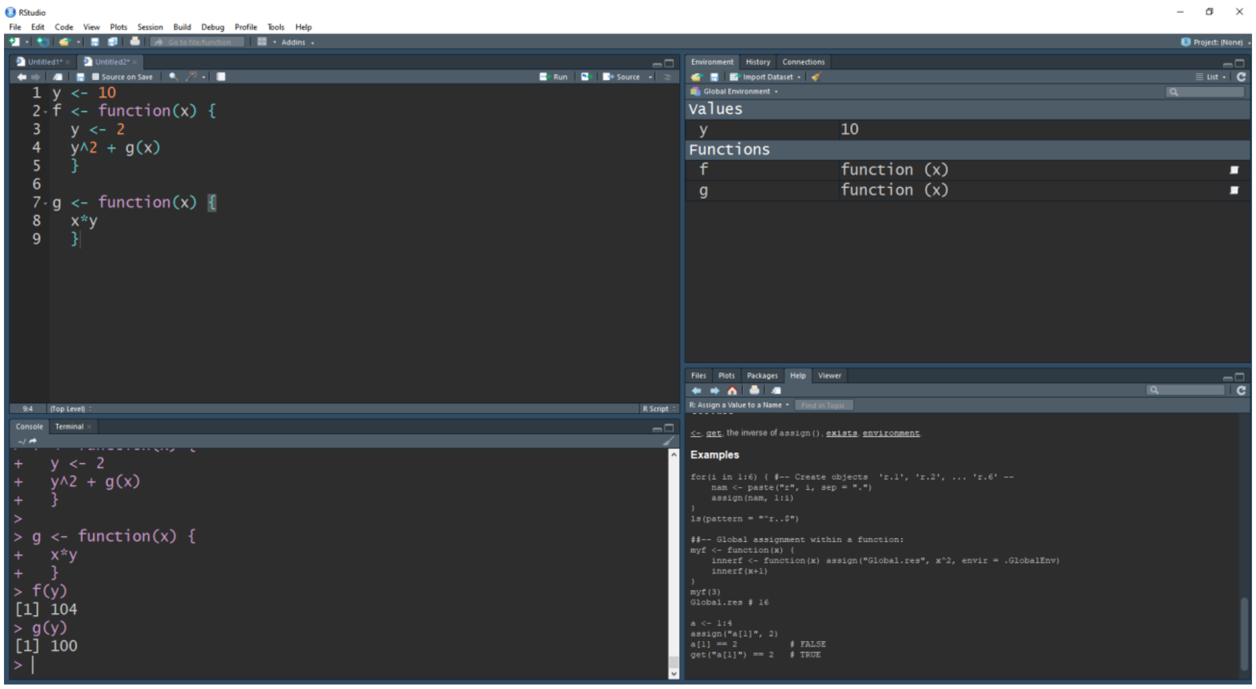
```
for(i in 1:6) { #-- Create objects  'r.1', 'r.2', ... 'r.6' --
    nam <- paste("r", i, sep = ".")
    assign(nam, 1:i)
}
ls(pattern = "^r..$")

##-- Global assignment within a function:
myf <- function(x) {
    innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
    innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2          # FALSE
get("a[1]") == 2   # TRUE
```

In the above picture we have declared a function named myf that takes an input, x. We can see on the 3rd panel that this function is available in our global environment, but we also have another value that we defined in our function. The inner function takes whatever input we give to myf, squares it, and assigns it to the global environment naming it Global.res. Notice that innerf is not available in the global environment, but the value that is created with the **assign()** function is. This is a special case as most programming languages will not allow you define functions inside functions.

Example 2:

What is lexical scoping? Lexical scoping is special to R and python and allows you to inherit global variables into your functions. This means you can define a variable outside of a function and use it within the function. In the below example we define the value y and two functions that take x as an input. In the function g(x) we inherit the value y from the global environment and multiply it by x. If you look at panel 2 we set x to be the value of y so 10 * 10 = 100.

The second example f(x) sets the value of y within the function. The output from g(x) is 100 and since y = 2 we get the result 104. Notice how the value of y did not change in the global environment, it is still 10, even though we set it to 2 in f(x).



```
y <- 10
f <- function(x) {
  y <- 2
  y^2 + g(x)
  }

g <- function(x) {
  x*y
  }
```

```
y        10
f        function (x)
g        function (x)
```

```
+   y <- 2
+   y^2 + g(x)
+   }
>
> g <- function(x) {
+   x*y
+   }
> f(y)
[1] 104
> g(y)
[1] 100
> |
```

<-, get, the inverse of assign(), exists, environment.

Examples

```
for(i in 1:6) { #-- Create objects  'r.1', 'r.2', ... 'r.6' --
    nam <- paste("r", i, sep = ".")
    assign(nam, 1:i)
}
ls(pattern = "^r..$")

##-- Global assignment within a function:
myf <- function(x) {
    innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
    innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2          # FALSE
get("a[1]") == 2   # TRUE
```

Naming Conventions

In most programming courses naming conventions are not taught because they are considered optional and up to the programmer to decide. For data scientists this can lead to a long pattern of bad coding principals, so it is important that you establish a consistent, verbose, method for your variable names. We will be using snake case for this course. The rules for snake case are simple, all words must be lower case, and each word needs to be separated by an underscore.

Your variables are expected to be verbose meaning that you need to make the variable name descriptive of the object. For example, if you have a data frame that contains housing prices you could name it housing_prices_df. Notice that I included the object type at the tail of the variable name. When you have a project that contains many variables, including the object type can help you better distinguish between objects.

Good variable names:

boston_housing_df

housing_prices_vector

boston_housing_linear_model

Bad variable names:

data, data1, data2, df1, df2 etc

prices (what is it?)

mod1 (what kind of model, what's it doing?)

Assignment & Operators

Operator	Description	Usage
+	Addition of two operands	a + b
–	Subtraction of second operand from first	a – b
*	Multiplication of two operands	a * b
/	Division of first operand with second	a / b
%%	Remainder from division of first operand with second	a %% b
%/%	Quotient from division of first operand with second	a %/% b
^	First operand raised to the power of second operand	a^b
<	Is first operand less than second operand	a < b
>	Is first operand greater than second operand	a > b

Operator	Description	Usage
==	Is first operand equal to second operand	a == b
<=	Is first operand less than or equal to second operand	a <= b
>=	Is first operand greater than or equal to second operand	a > = b
!=	Is first operand not equal to second operand	a!=b
&	Element wise logical AND operation.	a & b
|	Element wise logical OR operation.	a | b
!	Element wise logical NOT operation.	!a
&&	Operand wise logical AND operation.	a && b
||	Operand wise logical OR operation.	a || b
=	Assigns right side value to left side operand	a = 3
<-	Assigns right side value to left side operand	a <- 5
->	Assigns left side value to right side operand	4 -> a
<<-	Assigns right side value to left side operand	a <<- 3.4

Operator	Description	Usage
->>	Assigns left side value to right side operand	c(1,2) ->> a
:	Creates series of numbers from left operand to right operand	a:b
%in%	Identifies if an element(a) belongs to a vector(b)	a %in% b
%*%	Performs multiplication of a vector with its transpose	A %*% t(A)
[ ]	Slices an object (for lists we use double brackets [[ ]])	x[x==2]

Data Types

Data Type	Example	Verify

Logical
 	
Aka boolean
	TRUE, FALSE	`v <- TRUE`
		`print(class(v))`
		it produces the following result −
		`[1] "logical"`

Numeric	12.3, 5, 999	`v <- 23.5`
		`print(class(v))`
		it produces the following result −
		`[1] "numeric"`

Integer	2L, 34L, 0L	`v <- 2L`
		`print(class(v))`
		it produces the following result −
		`[1] "integer"`

Complex	3 + 2i	`v <- 2+5i`
		`print(class(v))`
		it produces the following result −
		`[1] "complex"`

Character
 	
Aka string
	'a' , '"good", "TRUE", '23.4'	`v <- "TRUE"`
		`print(class(v))`
		it produces the following result −
		`[1] "character"`

Raw	"Hello" is stored as 48 65 6c 6c 6f	`v <- charToRaw("Hello")`
		`print(class(v))`
		it produces the following result −
		`[1] "raw"`