

Programming with R 2020 University of Potsdam

Detlef Groth
Summer Semester 2020

KISS:
“Keep it simple, stupid!”?
:)
Paul D. Stroop 1960-12-04

Which Language?

- Perl 4/5/6 (1987, 1994, 2015) Guru: Larry Wall
 - Python 2/3 (1994, 2008) - Guru: Guido van Rossum
 - S/S3, (1976/1988), R (2000) - Ross Ihaka und Robert Gentleman)
 - Tcl/Tk 1988/1991, Tcl 8.0 1997 - John Ousterhout
- ⇒ in this course we use R
- Here: R as a general purpose programming language!!
 - Normal: R is used in the domain of statistics!!
 - “Programming with R” replaces the old course “Statistics with R”

R in the Master Bioinformatics

- Statistical Bioinformatics, M1
- (Mathematical and Algorithmical Bioinformatics, M1)
- (Databases and Practical Programming, B1)
- Analysis of Cellular Networks, E2
- Machine Learning in Bioinformatics, E2
- Structural Bioinformatics, E2
- Quantitative Genetics, E3
- (Adv. meth. for Analysis of Biochemical networks, E3)
- Project work, Master thesis

⇒ Python used in 2-3 modules

⇒ Matlab used in 2-3 modules

⇒ C/C++ used in Programming Expertise (B2)

Hello World!

```
groth:~$ R --slave -e 'print("Hello World!")'  
[1] "Hello World!"
```

```
groth:~$ Rscript -e 'print("Hello World!")'  
[1] "Hello World!"
```

Outline

Day 1 (basics)

- setup, install editor, simple programs
- variables, operators
- data structures, control flow

Day 2 (basics)

- functions
- file input/output
- terminal interaction
- command line arguments

Day 3 (advanced)

- object oriented programming
- code documentation

Day 4 (advanced)

- using packages
- writing packages
- package documentation

Day 5 (advanced)

- graphical user interfaces
- tcltk (shiny)

Examination

- two short written questions (10 minutes)
- practical programming task for about 90 minutes (3/4 CP)
- for 6 credit points MS-BAM students have to complete a more extensive homework within around a month
- dates:
 - 1.) Thu, October 1st, Review session 1 13:00
 - 2.) Tue, October 6th, Exam 1 13:00
 - 3.) Mon, October 26th, Review session 2 13:00
 - 4.) Wed, October 28th, Exam 2 13:00

Course Procedure

- video materials for each lecture around 90min
- 10:00-11:30 self study of video materials
- 12:00-13:00 seminar about the lecture (Zoom)
- 13:30-16:00 practical programming using Padup-Chat and Zoom
- if you know one programming language learning the others will be much easier ...

Lecture Materials / Online Resources

- Videos:
 - Derek Banas YT - R Tutorial
- Links, Books
 - <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>
 - <https://www.tutorialspoint.com/r/>
- More links to tutorials, reference cards on Moodle

Moodle Course

- Moodle: `https://moodle2.uni-potsdam.de/course/view.php?id=22964`
- Moodle-Login: E-mail account credentials
- Course key: Golm2020

Outline

Table of Contents	11
1 Intro	14
1.1 R Basics	17
1.2 Programming Intro	40
1.3 Variables and Data Containers	41
1.4 System Variables	54
1.5 Help	55
1.6 Eval	59
1.7 Operators	61
1.8 Control Flow: Conditionals	63
1.9 Control Flow: Loops	68

1.10	Functions	73
1.11	Colored Terminal	78
1.12	Exercise Introduction	84

Programming with R

SS 2020

Introduction

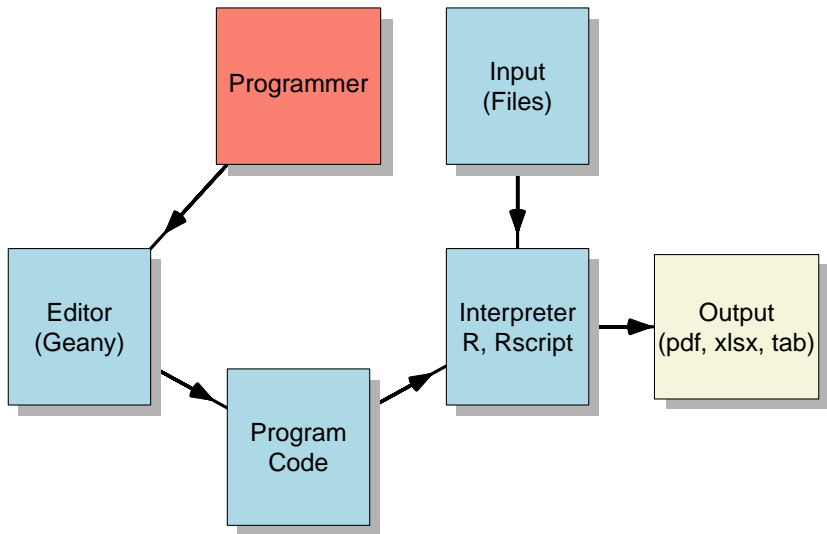
Detlef Groth
2020

1 Intro

Outline

1.1	R Basics	17
1.2	Programming Intro	40
1.3	Variables and Data Containers	41
1.4	System Variables	54
1.5	Help	55
1.6	Eval	59
1.7	Operators	61
1.8	Control Flow: Conditionals	63
1.9	Control Flow: Loops	68
1.10	Functions	73
1.11	Colored Terminal	78
1.12	Exercise Introduction	84

Our Workflow



Software

- OS: recommened Linux (Windows, OSX)
- R programming software environment
`http://www.r-project.org`
- Text editor Geany `https://www.geany.org/`

1.1 R Basics



- R is an implementation of the S-Language.
- S was originally developed at Bell Laboratories in the 1980s, 1990s.
- Implementations (GUI and command line):
- S-Plus on Win32, Unix (commercial Insightful Co), 1995 V 3.3, 1997 first version for Windows.
- R (open source, Win32, Unix, Mac-OSX), 1997 first alpha, 2000 V 1.0, currently 3.6.3.
- We will use the R variant which is free.
- Knowledge validity? 2000 ... 2020 ... 2030 ...

R-features

- Available for Win32, Linux, Mac-OSX and others OS.
- Mainly console driven, but GUIs are available
- Shell with history, “TAB”-completion of variables, functions, objects as in Unix :)
- Extensible programming language
- Large amount of packages (CRAN)
- Bioconductor project for packages useful to analyse genomics data
- <http://www.bioconductor.org/>
- Sometimes too many packages although ...

Working with R

- interactive in the R terminal, good for statistics and data exploration
- using script files from the terminal
- in this course we prefer script files as this requires correct code in your files

We learn to use R without doing Statistics!!

Hello World!

```
#!/usr/bin/Rscript  
print("Hello R-World! 2020!") ;  
⇒ [1] "Hello R-World! 2020!"
```



⇒ `#!/usr/bin/Rscript` ⇒ shebang (Unix)

Version

```
#!/usr/bin/Rscript  
print(R.version.string);  
⇒ [1] "R version 3.6.3 (2020-02-29) "
```

 version.R

Language vs Interpreter

Language	Interpreter Compiler	Interactive	Tiobe- Index
C/C++	gcc, g++	gdb	1 and 4
Python	python3	python3, idle3	3 (3, 4)
R	R, Rscript	R, RStudio	9 (15, 14)
Perl	perl	perl -de1, pirl	13 (19, 16)
Matlab	matlab	matlab	16 (20, 11)
Julia	julia	julia	28 (42, 37)

Modus Operandi

- No compilation to machine code necessary with R, Perl, Python.
- How we can work with scripting languages?
 - interactive (R!!, Python3).
 - one-liners in the terminal (Perl!!)
 - **run code from saved source code script files** (R, Perl, Python)
 - for the latter you need a good text editor!

Text Editor

- edit in plain text mode
- syntax highlighting (R, Python, C, C++, Octave/Matlab, LaTeX, Markdown, ...)
- list of functions, classes overview
- code snippets, templates
- fast and lightweight
- project management (folder based)
- external tools for code snippets (no support by me):
 - Windows: autohotkey
 - Linux (X11): autokey
 - OSX: Hammerspoon

Why not Rstudio?

- Rstudio great for interactive sessions and learning
- students however are not encouraged to write running applications
- often code mess of different trials instead of structured programs
- does not encourages structured work within different projects
- very memory intensive, often crashing
- may be too many features
- quite R specific, although support for Python and others are improving
- outline of functions not very sophisticated

*Rstudio encourages an unstructured working style
(DG)*

Why Geany?

- <https://www.geany.org>
- general purpose text editor
- supports many more languages than Rstudi (Python, R, LaTeX, Perl, Matlab/Octave, C, C++, ...)
- use one editor for all of your programming tasks
- encourages good code and project organization
- teaches you to understand what is going on in the background



Other editors

- hardcore programmers: Emacs, Vi, Vim
- easier variants: Geany, Kate, Gedit, Atom, Visual Studio Code
- I use MicroEmacs (last update in 2010) ...
- RStudio is good for learning but not for coding! (personal opinion)
- RStudio license is dubious
- we will use **Geany** in this course
<https://www.geany.org/> like in the statistics course
- Geany can be used with a lot of Programming languages and is available for all the major operating systems

Installs

- **Linux-Fedora:**

```
sudo dnf install geany R
```

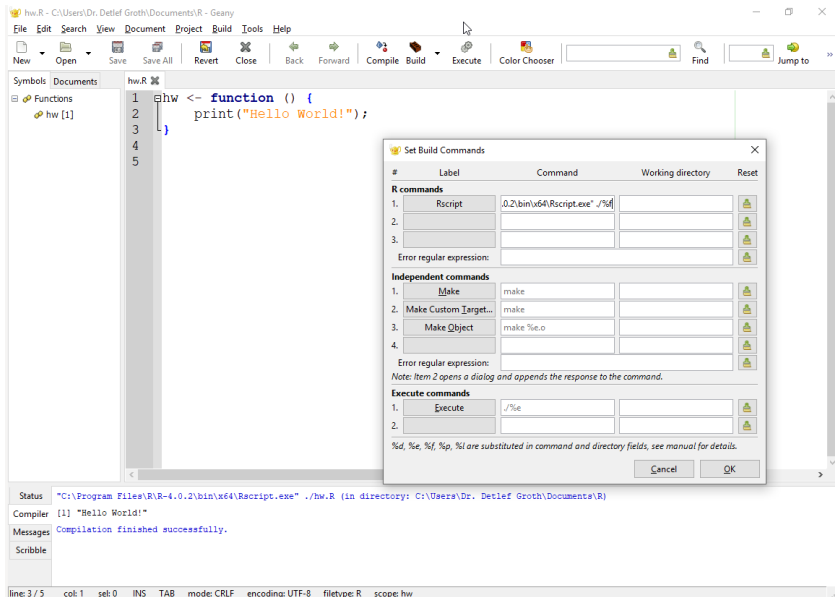
- **Linux-Ubuntu:**

```
sudo apt-get install geany R
```

- **Windows/MacOSX**

- **Geany:** <https://www.geany.org/download/releases/>
- **R:** <https://lib.ugent.be/CRAN/>

Setup R for Geany on Windows



Setup R for Geany on Linux

The screenshot shows the Geany IDE interface. The main editor displays an R script with the following code:

```
1 hw <- function () {  
2   print("Hello World!")  
3 }  
4  
5 hw()  
6
```

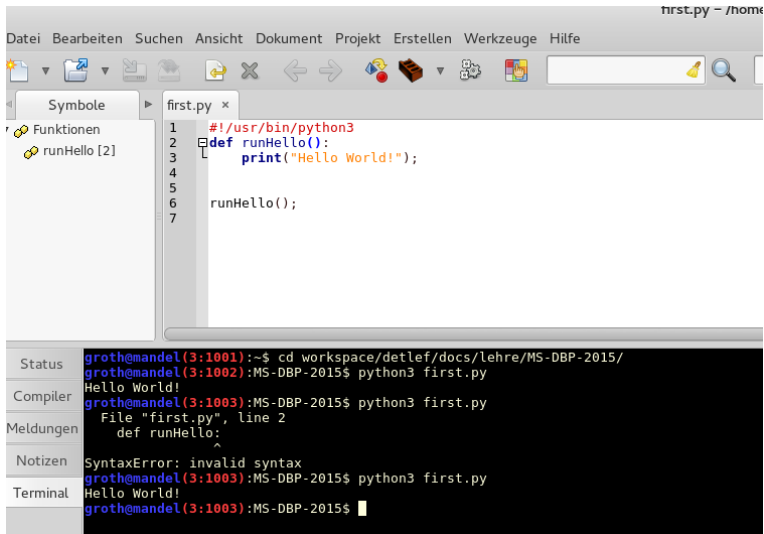
The 'Symbols' pane on the left shows a function 'hw [1]'. The 'Set Build Commands' dialog is open, showing the following configuration:

#	Label	Command	Working directory	Reset
R commands				
1.	Rscript	Rscript %f		
2.				
3.				
		Error regular expression:		
Independent commands				
1.	Make	make		
2.	Make Custom Target...	make		
3.	Make Object	make %e.o		
4.				
		Error regular expression:		
Note: Item 2 opens a dialog and appends the response to the command.				
Execute commands				
1.	Execute	./%e		
2.				

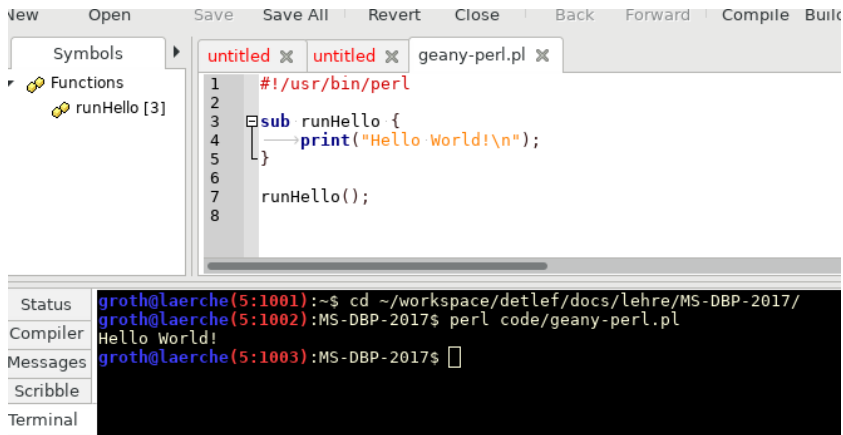
At the bottom, the status bar shows the following messages:

```
Status: Rscript hw.R (in directory: /home/groth/workspace/delfgroth/docs/Lehre/PwR-2020/scripts)  
Compiler: [1] "Hello World!"  
Messages: Compilation finished successfully.
```

Editor Setup: Geany - Python



Editor Setup: Geany - Perl



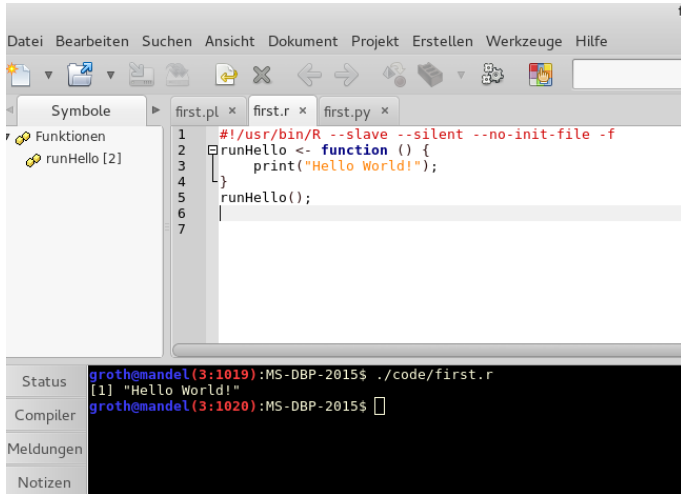
The screenshot displays the Geany IDE interface. The top menu bar includes 'View', 'Open', 'Save', 'Save All', 'Revert', 'Close', 'Back', 'Forward', 'Compile', and 'Build'. The left sidebar shows a 'Symbols' pane with a 'Functions' section containing 'runHello [3]'. The main editor window has three tabs: 'untitled', 'untitled', and 'geany-perl.pl'. The 'geany-perl.pl' tab is active, showing a Perl script with the following code:

```
1  #!/usr/bin/perl
2
3  sub runHello {
4      print("Hello World!\n");
5  }
6
7  runHello();
8
```

Below the editor is a terminal window with a status bar on the left. The status bar shows 'Status', 'Compiler', 'Messages', 'Scribble', and 'Terminal'. The terminal output is as follows:

```
groth@laerche(5:1001):~$ cd ~/workspace/detlef/docs/lehre/MS-DBP-2017/
groth@laerche(5:1002):MS-DBP-2017$ perl code/geany-perl.pl
Hello World!
groth@laerche(5:1003):MS-DBP-2017$
```


Editor Setup: Geany - R



Snippets

Geany Wiki: “Snippets are small strings or code constructs which can be replaced or completed to a more complex string. So you can save a lot of time when typing common strings and letting Geany do the work for you.”

- Tools->Configuration Files->snippets.conf
- `https://www.geany.org/manual/current/index.html#user-definable-snippets`
- `https://wiki.geany.org/snippets/start`

Standalone Scripts: Shebang, Linux, OSX

- shebang: first line of a file starts with `#!` and the path to a script interpreter
- rest of the file is run with this interpreter
- use
`chmod 755 filename`
to make a file executable

```
#!/bin/sh
# next line is executed by /bin/sh
echo 'Hello World!'
```

Standalone Scripts: chmod 755 / Linux

```
groth@mandel:~$ cat ./code/first.py
```

```
#!/usr/bin/python3
```

```
def runHello():
```

```
    print('Hello World!');
```

```
runHello();
```

```
groth@mandel:~$ ./code/first.py
```

```
bash: ./code/first.py: Keine Berechtigung
```

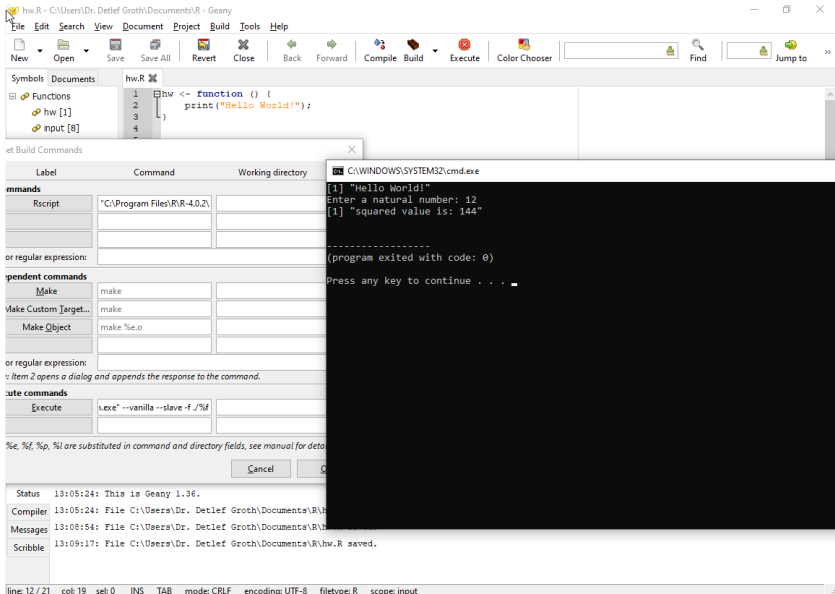
```
groth@mandel:~$ chmod 755 ./code/first.py
```

```
groth@mandel:~$ ./code/first.py
```

```
Hello World!
```

```
groth@mandel:~$ cat ./code/first.r
#!/usr/bin/R --slave --no-init-file -f
runHello <- function () {
    print('Hello World!');
}
runHello();
groth@mandel:~$ chmod 755 ./code/first.r
groth@mandel:~$ ./code/first.r
[1] 'Hello World!'
```

Geany Windows with execute



Windows Paths with Geany

Build->Set- Build Comands -> Execute

```
"C:\Program Files\R\R-4.0.2\bin\x64\Rterm.exe"  
--vanilla --slave -f %f
```

1.2 Programming Intro

Concepts:

- variables
- data types
- data containers
- control flow
- functions - day 2
- objects, classes, day 3
- libraries/packages, day 4
- user! interfaces, day 5 (writing programs for others)

1.3 Variables and Data Containers

- variable: one or more values
- container: structure to store those values
- reserve some space in memory for the values
- different data types (different space requirements):
 - numbers (integer, long, float, double)
 - strings
 - booleans
 - factors (classified strings)
- some basic data structures:
 - skalar: single values
 - vector, array (matrix): many values, same type
 - list, data frame: many values, diff. types possible
 - others: tibble, table, graph, sql-table

Variables:

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.

[Tutorialspoint — Python 3 – Variables, 2019]

- variables are an alias for data or other objects (functions, database connection, ...)
- we have always **global and local scope** of variables
- local scope means restricted access, for instance inside functions or objects
- global scope means visible everywhere in your program
- hint: it is advisable to avoid global scope
- we have some basic data types for (data) variables
- R data types:
 - numeric
 - string
 - boolean
 - factor
 - ...

Variables: R

- local scope assignment operator:
 - `var = value`
 - `var <- value`
 - `value -> var`
- all the same: `x = 3; x <- 3; 3 -> x`
- if used at the global namespace they are global
- global scope assignment operator `<<-` inside a function:
`global.x <<- 3`
- hidden variables: start with a dot `.hidden=1`
- command `ls()` lists all variables but not hidden ones

Variables: R-Session

```
> global.x=3
> .hidden.var = 1
> test.func = function () { y = 3 ; z <<- 4 ;
+     global.x <<- global.x +1
+ }
> ls()
[1] "global.x"    "test.func"
> test.func()
> ls()
[1] "global.x"    "test.func"   "z"
> global.x
[1] 4
> test.func()
> global.x
[1] 5
```

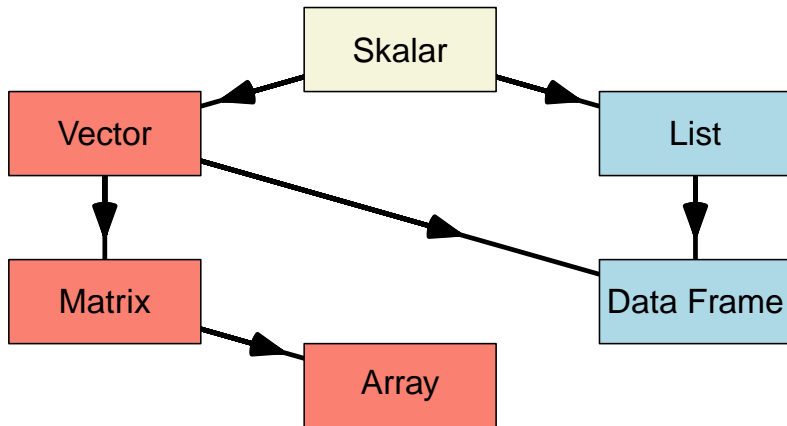
Summary variables

Language	local	global
R	#inside function x <- 1 x = 1	#inside function globX <-1 ;
Python	x = 1; #inside def	global globX; globX = 1; # inside def
Perl	my \$x = 1; # inside sub	my \$globX = 1; our \$GLOBX = 1; # out of sub ;

⇒ if possible avoid global variables!!!

Data containers (data structures)

⇒ objects in which we store our variables



Variable examples

```
#!/usr/bin/Rscript
skal_var = 3;
vect_var = c(1,2,3,4);
list_var = list('me'='Instructor', 'you'='Student!');
matr_var = matrix(1:4,nrow=2)
print(skal_var);
print(vect_var);
print(list_var[1]);
print(list_var[['me']]);
print(matr_var)
print(matr_var[1,1:2])
⇒ [1] 3
⇒ [1] 1 2 3 4
⇒ $me
⇒ [1] "Instructor"
⇒
⇒ [1] "Instructor"
```


⇒ [, 1] [, 2]
⇒ [1,] 1 3
⇒ [2,] 2 4
⇒ [1] 1 3

 vars.r

⇒ R arrays/vectors start with 1 for first element

Data Containers

Lang	Skalar	Vector/Array	List/Hash/Dict
R	<code>x=1</code>	<code>a=c(1,2,3)</code>	<code>l=list(a=1,b=2)</code>
Perl	<code>\$x=1 ;</code>	<code>@a=(1,2,3);</code>	<code>%l=('a'=>1, 'b'=>2);</code>
Python	<code>x=1</code>	<code>a=[1,2,3]</code>	<code>l={'a':1, 'b':2 }</code>

Giving out a variable

```
#!/usr/bin/Rscript
x=3;
print(x) # with automatic return and line number
cat(x,'\n') # no automatic line break and no numbering
print(paste('x is',x));
# using sprintf with formatting codes
print(sprintf('float: %.3f integer: %i ',x,x))
⇒ [1] 3
⇒ 3
⇒ [1] "x is 3"
⇒ [1] "float: 3.000 integer: 3 "
```

 varsout.r

Getting input from the user

⇒ users should not need to write values in the source code!!

- interactive (readline)
- command line arguments (commandArgs - day 2)
- (Python:

```
> > > x = input('give a number:  '))
```

- R:

```
> x = readline(prompt='give a  
number: ')
```

- but with R readline works only in interactive mode, not in scripts :(

Our R readline extension

```
input = function (prompt="Enter: ") {  
  if (interactive() ){  
    return(readline(prompt))  
  } else {  
    cat(prompt);  
    return(readLines("stdin",n=1))  
  }  
}  
  
x=input("Enter a numerical value: ")  
x=as.integer(x)  
print(x*12)
```

```
[groth@mandel]$ Rscript ../scripts/input.r  
Enter a numerical value: 12  
[1] 144
```

1.4 System Variables

```
#!/usr/bin/R --slave --no-init-file -f  
print(Sys.getenv("HOME"))  
print(Sys.getenv("USER"))
```

```
⇒ [1] "/home/groth"
```

```
⇒ [1] "groth"
```



On Windows:

```
> Sys.getenv("HOME")  
[1] "C:\\Users\\Dr. Detlef Groth\\Documents"  
> Sys.getenv("USER")  
[1] ""  
> Sys.getenv("USERNAME")  
[1] "Dr. Detlef Groth"
```

1.5 Help

Help for R

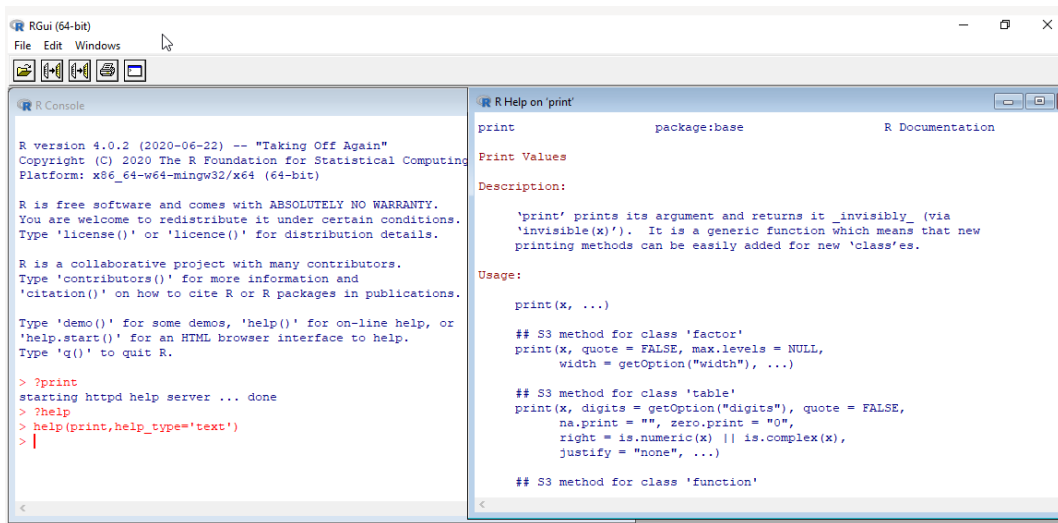
Interactive:

```
> help(print)
> ?print
> ??clust
```

Bash:

```
$ R -e 'help(print) '
$ function Rdoc { R --slave -e "help($1) " ; }
$ # this can go into .bashrc
$ Rdoc print
```

Help on Windows



The screenshot shows the RGui (64-bit) window. The title bar reads "RGui (64-bit)". The menu bar includes "File", "Edit", and "Windows". Below the menu bar is a toolbar with icons for file operations. The window is split into two panes. The left pane, titled "R Console", displays the R version 4.0.2 (2020-06-22) -- "Taking Off Again" and copyright information. It also contains instructions on how to use the help system, including the `?print` command. The right pane, titled "R Help on 'print'", shows the documentation for the `print` function from the `package:base`. It includes a description of the function, its usage, and examples of how to use it with different data types like `factor`, `table`, and `function`.

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

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.

> ?print
starting httpd help server ... done
> ?help
> help(print,help_type='text')
> |
```

```
print                                package:base                                R Documentation

Print Values

Description:

'print' prints its argument and returns it _invisibly_ (via
'invisible(x)'). It is a generic function which means that new
printing methods can be easily added for new 'class'es.

Usage:

print(x, ...)

## S3 method for class 'factor'
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table'
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0",
      right = is.numeric(x) || is.complex(x),
      justify = "none", ...)

## S3 method for class 'function'
```


General Help in the Terminal

- `man command` \Rightarrow top down reading
- `info command` \Rightarrow hyper text browser
- `info` \Rightarrow all info pages
- `command --help` \Rightarrow close to man pages
- `command -h`

```
$ man less
```

```
$ info less
```

```
$ man cd
```

```
$ info bash
```

```
$ less --help | less
```

Comments: R

```
# a hash single line comment
# use it for commenting
print('Hello World!')
if (FALSE) {
  trick
  this gets never executed
  # used for larger blocks
  res=dg.longrunning.func()
}
print('Bye') # executed again
```

1.6 Eval

```
#!/usr/bin/R --slave --no-init-file -f  
x=3;  
eval(parse(text=paste('print(3*',x,')')));  
⇒ [1] 9
```

 eval.r

But “eval” with care, not on server for clients!!

Summary I

- Background:
 - installs and editor
 - interactive, terminal, script file
 - editing and executingscript files
 - standalone executable files, chmod 755
- Programming:
 - variables: skalar, vector, array, list
 - user input
 - comments
 - help systems
 - system variables

1.7 Operators

Logical
&, |, !, ...

Arithmetic
+, −, *, /...

Membership
%in%, (%ni%)

Assignment
=, <−, <<−, ...

Relational
==, <=, >...

Miscellaneous
?, %*%, :

```
> '%ni%' = function (x,y) { !(x %in% y) }
```

Help about Operators

- `help(TOPIC)`
- `help(Arithmetic)`
- `help(Logic)`
- `??operators`

```
> x <- 5
```

```
> x
```

```
[1] 5
```

```
> x = 3
```

```
> x
```

```
[1] 3
```

```
> x == 3
```

```
[1] TRUE
```

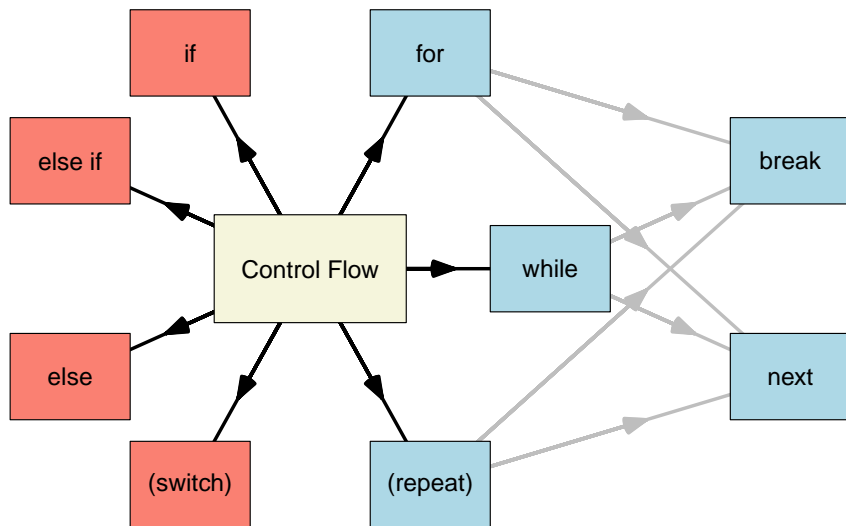
```
> !(x == 3)
```

```
[1] FALSE
```

```
> c(1:6) %in% x
```

```
[1] FALSE FALSE TRUE FALSE FALSE FALSE
```

1.8 Control Flow: Conditionals



Decision Making:

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

[Tutorialspoint — Python 3 – Decision Making, 2019]

- programming logic:
- `if` something is true \Rightarrow do something
- `else if` something else is true \Rightarrow do something
- `else` in any other case \Rightarrow do something
- `?Control`
- Perl and R use curly braces to structure blocks
- Python is special, it uses colon and indentation!
- van Rossum thought that programmers must be forced to properly format their code
- I personally think that this is the task of the editor, not the programmer

Conditionals

```
#!/usr/bin/Rscript
```

```
Sys.time()
```

```
substr(Sys.time(),12,13) # get the hour part
```

```
x=as.numeric(substr(Sys.time(),12,13))
```

```
if (x > 4 && x < 11) {
```

```
  print('Good morning')
```

```
} else if (x < 14) {
```

```
  print('Good day!')
```

```
} else if (x < 18) {
```

```
  print('Good afternoon!')
```

```
} else if (x < 22) {
```

```
  print('Good evening!')
```

```
} else {
```

```
  print('Good night!')
```

```
}
```

```
⇒ [1] "2020-09-12 18:18:06 CEST"
```

```
⇒ [1] "18"
```

```
⇒ [1] "Good evening!"
```



Placement of Curly Braces

```
if (cond) {          if (cond) {expr}          if (cond)
    expr              {                          {
}                                                            expr
                                                            }
```

- whitespace does not matter
- formatting just for better reading and understanding
- just a matter of personal style
- my style:
 - opening brace at the end of a line of a control flow construct (no line lost for a single brace!)
 - closing brace at the beginning of a line, or just whitespaces before
 - else if and else directly after closing brace
- your style: develop one and stick with it

1.9 Control Flow: Loops

Loops:

In general, statements are executed sequentially – The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times.

[Tutorialspoint — Python 3 – Loops, 2019]

Loop aim

⇒ Do operations as long as a certain condition is true (*while*) or for a certain condition or a certain number of times (*for*)!

- *for* (R, Perl, Python)
- *foreach* (Perl)
- *while* (R, Perl, Python)
- *repeat* (R)
- exit from loop:
 - *last* (Perl)
 - *break* (Python, R)
- skip rest of current and go to next iteration:
 - *next* (Perl, R)
 - *continue* (Python)


Loops - for: R

```
#!/usr/bin/R --slave --silent --no-init-file -f
lst=list(a=1,b=2,k=3)
for (nm in names(lst)) {
  cat(paste(nm, '=', lst[[nm]], ';'))
}
cat('\n')
for (i in 1:4) {
  cat(paste(i, ' '))
}
cat('\n')
⇒ a = 1 ; b = 2 ; k = 3 ;
⇒ 1 2 3 4
```

 loopfor.R

Loops - while: R

```
#!/usr/bin/R --slave --silent --no-init-file -f
x=1
while (x < 3) {
  x=x+1
  cat(paste('x =', x, ' ; '))
}
cat('\n')
while (TRUE) {
  if (x > 10) { break ; }
  cat(paste(x, ' '))
  x=x+1
}
cat('\n')
⇒ x = 2 ; x = 3 ;
⇒ 3 4 5 6 7 8 9 10
```

 loopwhile.R

Summary Control Flow

- curly braces, indentation
- if, else if, else (switch)
- for, while (repeat)
- repeat == while(TRUE)
- break, next
- editor snippets

1.10 Functions

- **function** keyword R (def - Python, sub - Perl)
- functions are used to structure code
- organize your code in a central place
- functions can get arguments, sometimes also called *parameters*
- arguments are processed inside the function
- arguments can be optional or mandatory
- functions can return processed value(s)

Function:

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

[Tutorialspoint — Python 3 – Functions, 2019]

Preview Functions

- function keyword (Python def, Perl sub, Tcl proc, ...)
- function name
- argument definitions - input of the caller of this function
- function body - the implementation
- eventually a return statement
- more tomorrow

Functions: R

- Intro-R: chapter 10!!
- <http://www.tutorialspoint.com/r/index.htm>

```
name <- function (arguments) {  
  function body  
  return(value)  
}
```

```
#!/usr/bin/R --slave --silent --no-init-file -f  
func.name = function (mand.arg, optional.arg=2) {  
  return(mand.arg^optional.arg);  
}  
print(func.name(2))  
print(func.name(2, 4))  
print(func.name(optional.arg=4, mand.arg=3))
```

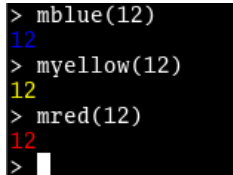
⇒ [1] 4
⇒ [1] 16
⇒ [1] 81



1.11 Colored Terminal

⇒ https://en.wikipedia.org/wiki/ANSI_escape_code

```
> mblue = function (msg) {  
+   cat(paste('\033[94m', msg, '\033[0m\n', sep=' '))  
+ }  
> mred = function (msg) {  
+   cat(paste('\033[91m', msg, '\033[0m\n', sep=' '))  
+ }  
> myellow = function (msg) {  
+   cat(paste('\033[93m', msg, '\033[0m\n', sep=' '))  
+ }
```



```
> mblue(12)  
12  
> myellow(12)  
12  
> mred(12)  
12  
> 
```

	90
31	91
32	92
33	93
34	94
35	95
36	96
37	97

⇒ see also R-library crayon

Colored Terminal with global variables

```
GREY=GRAY='\033[90m'  
RED='\033[91m'  
GREEN='\033[92m'  
YELLOW='\033[93m'  
BLUE='\033[94m'  
MAGENTA='\033[95m'  
CYAN='\033[96m'  
WHITE='\033[97m'  
RESET='\033[0m'  
cat(paste(RED, 'Hello Red World!\n', RESET))  
cat(paste(GREEN, 'Hello Green World!\n', RESET))  
cat('Hello Normal World!\n')
```

```
> GREY=GRAY='\033[90m'  
> RED='\033[91m'  
> GREEN='\033[92m'  
> YELLOW='\033[93m'  
> BLUE='\033[94m'  
> MAGENTA='\033[95m'  
> CYAN='\033[96m'  
> WHITE=BLUE='\033[97m'  
> RESET=BLUE='\033[0m'  
> cat(paste(RED, 'Hello Red World!\n', RESET))  
Hello Red World!  
> cat(paste(GREEN, 'Hello Green World!\n', RESET))  
Hello Green World!  
> cat('Hello Normal World!\n')  
Hello Normal World!
```



```
#!/usr/bin/Rscript
showAnsi = function() {
  for (i in 0:10) {
    for (j in 0:9) {
      n = 10*i + j;
      if (n > 109) break;
      cat(sprintf("\033[%dm %3d\033[m", n, n))
    }
    cat("\n");
  }
  return (0);
}
showAnsi()
```

```
[groth@bariuke build]$ ./ansi.R
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109
[1] 0
```

Hints

- avoid global variables if possible
- personal coding styles for indentation
- DG:
 - indent with four spaces
 - opening brace not on own line
 - only final closing brace on own line
- configure Geany for a template (snippet)!!
- (for applications always have a function *main* which should start your program, place it at the end)
- (before *main* other function definitions)
- (check if the script is interpreted directly, if yes execute *main*)

Summary II

- variables: local vs global
- data types integer, float, string, boolean, ...
- data structures scalar, vector, array, list
- operators: math, logical
- conditionals: if, else if, else
- loops: for, while and break and next
- ANSI codes for colored terminal
- tomorrow:
 - functions
 - command line arguments
 - file input and output

1.12 Exercise Introduction

Aim:

- install the required tools, editor, R
- prepare a directory for todays session
- make you comfortable with the editor
- write simple hello world scripts
- control flow exercises
- syntax highlighting
- code snippets

Task 1 - Installation Geany

- install the Geany text editor
- Linux, Unix use the package manager
- Fedora: `sudo dnf install geany`
- Debian/Ubuntu: `sudo apt-get install geany`
- Windows/MacOSX - manual download and install from:
<https://www.geany.org/download/releases/>
- after install start the Geany text editor (geany)

Task 2 - Install R

- install the R programming tools
- Linux and other Unixes (MacOSX(?), BSD, ...) use the package manager
- Fedora: `sudo dnf install R`
- Debian/Ubuntu: `sudo apt-get install R`
- Windows/MacOSX - manual download and install from:
<https://lib.ugent.be/CRAN/>
- after install start R (R)
- do a simple calculation `> 1+1` and then `> 1+2==2` (two equal signs!)
- check out what is your home directory by writing
`Sys.getenv("HOME")` (UNIX) or
`Sys.getenv("USERPROFILE")` (Windows)
in the R terminal after

Task 3 - Creating a “Hello World” script

- get the geany editor running: `$ geany &`
- open the terminal inside geany (Linux)
- create a directory inside your home called `PwR-labs`
- create the a R hello world scripts from the lecture at slide 20 (`hw.R`)
- store the script files inside the new `PwR-labs` directory
- UNIX (Linux, OSX):
 - use `chmod 755` to make them running standalone
 - run them in your terminal, check output
- Configure the geany text editor, that it can run R scripts directly by using the Geany menu point 'build->set build command'
- below at “execute commands” write the command line to execute R scripts

- Linux: `R --vanilla --slave %f`
- Windows: `"C:\Program Files\R\R-4.0.2\bin\x64\Rterm.exe" --vanilla --slave "%f"`
- MacOSX: `R --vanilla --slave "%f"`

Task 4 - Extending “Hello World!”

- extend the hello world script, so that it prints as well the home directory name `studNNNN`
- extend the script that hello world is printed in green color on the terminal by using a variable `green` as can be seen in the lecture
- extend the script, that it displays as well the user name in the terminal in red color `Sys.getenv('USERNAME')` (Windows), `Sys.getenv('USER')` (Unix)
- save those scripts with a new name like `hw-user.pl`
- make them running standalone using `chmod 755 filename`
- replace `filename` with the name of the script

Task 5 - Dealing with input

- create multiplier scripts for R: `mymulti.R`
- you should place the input function (see below) at the beginning of your script
- the user should be asked for two numbers and then the scripts multiplies both numbers and prints the result
- please note, that you have to convert the input to a number by using the R function `as.numeric`
- create an `exercise.R` script which creates two random numbers using the R `sample` function
- `sample(1:20,2)` samples two numbers in the range from 1 to 20
- write a multiplication questions with those two random numbers to the user
- compare the given user answer with the real solution
- write wrong or false in red and green to the terminal depending on the correctness of the answer
- use a for loop to ask 5 questions one after the other

```
input = function (prompt="Enter: ") {  
  if (interactive() ){  
    return(readline(prompt))  
  } else {  
    cat(prompt);  
    return(readLines("stdin",n=1))  
  }  
}  
# comment for usage example  
# x=input("Enter a numerical value: ")  
# x=as.integer(x)
```

⇒ finish the task until tomorrow

⇒ have a look at: [https://www.tutorialspoint.com/r/](https://www.tutorialspoint.com/r/chapters)
chapters until decision making and loops

References

- Tutorialspoint — Python 3 – Variables. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/python3/python_variable_types.htm. [Online; accessed 21-October-2019].
- Tutorialspoint — Python 3 – Decision Making. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/python3/python_decision_making.htm. [Online; accessed 21-October-2019].
- Tutorialspoint — Python 3 – Loops. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/python3/python_loops.htm. [Online; accessed 21-October-2019].

Tutorialspoint — Python 3 – Functions. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/python3/python_functions.htm. [Online; accessed 20-October-2019].