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

$\Rightarrow$ in this course we use R

- Here: R as a general purpose programing 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

$\Rightarrow$ Python used in 2-3 modules
$\Rightarrow$ Matlab used in 2-3 modules
$\Rightarrow$ 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

Programming with R
SS 2020
Introduction

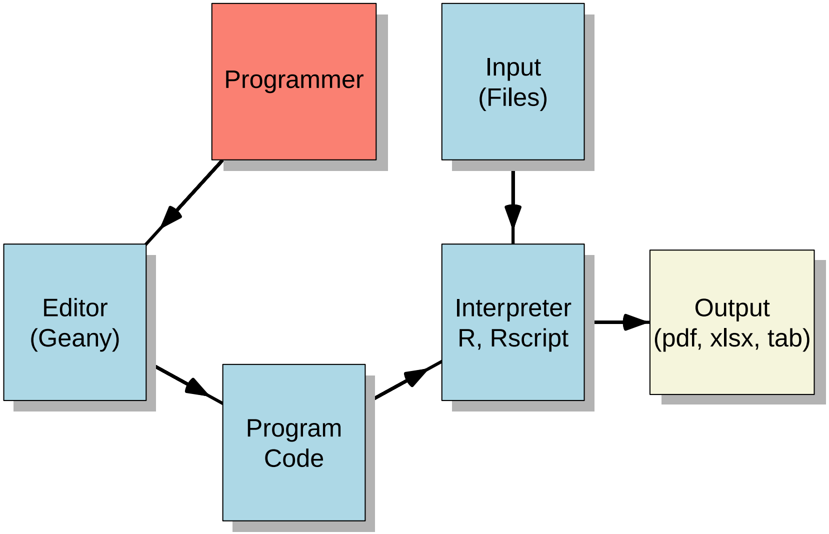
Detlef Groth

2020

1 Intro

Outline

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 prefere 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!"
```

tobi.R

⇒ #!/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

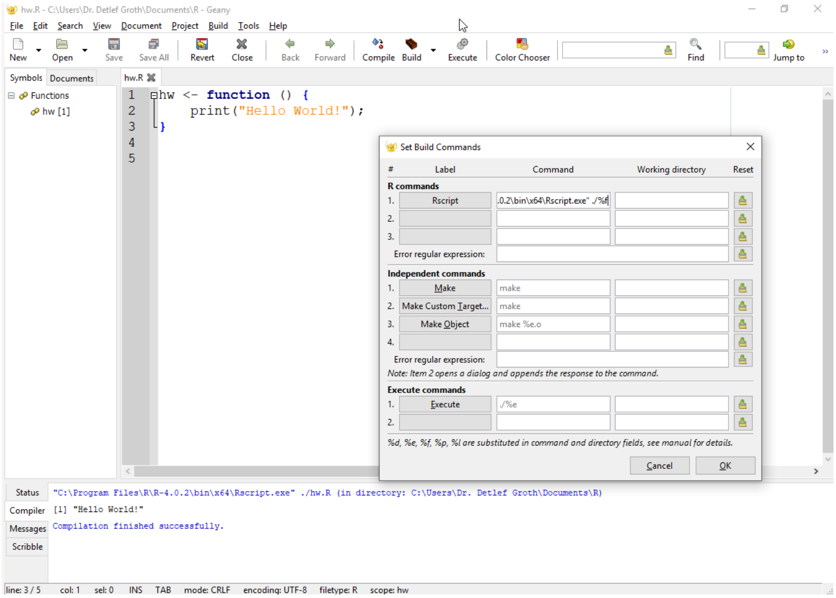
Other editors

- 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


hw.R - C:\Users\Dr. Detlef Groth\Documents\R - Geany
File Edit Search View Document Project Build Tools Help
New Open Save Save All Revert Close Back Forward Compile Build Execute Color Chooser Find Jump to
Symbols Documents
Functions
hw [1]
hw.R

```
hw <- function () {
    print("Hello World!");
}
```

Set Build Commands

#	Label	Command	Working directory	Reset
R commands				
1.	Rscript	.0.2\bin\x64\Rscript.exe" ./%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.

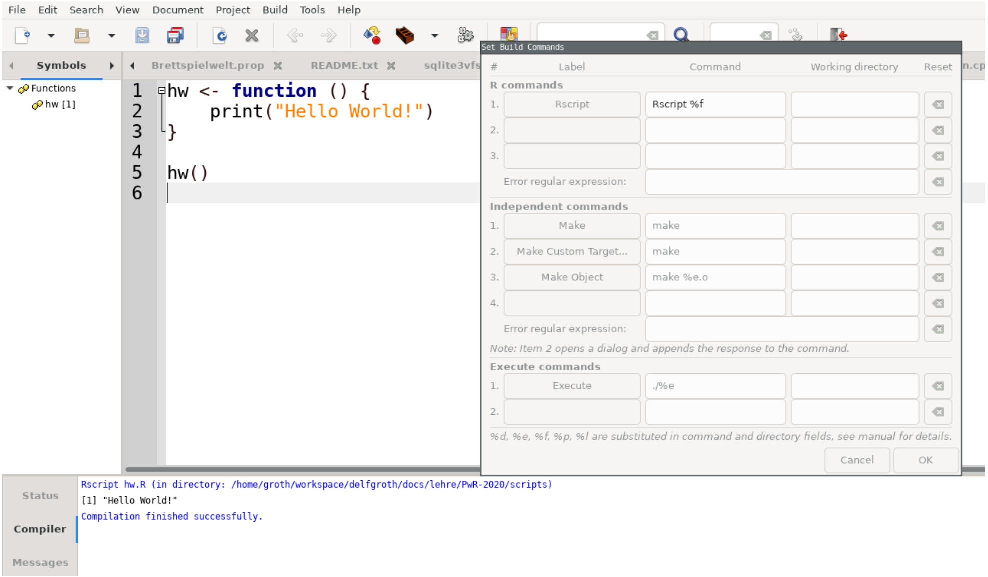
#	Label	Command	Working directory	Reset
Execute commands				
1.	Execute	./%e		
2.				

%d, %e, %f, %p, %l are substituted in command and directory fields, see manual for details.

```
"C:\Program Files\R\R-4.0.2\bin\x64\Rscript.exe" ./hw.R (in directory: C:\Users\Dr. Detlef Groth\Documents\R)
[1] "Hello World!"
Compilation finished successfully.
```

line: 3 / 5 col: 1 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: R scope: hw

Setup R for Geany on Linux



```
hw <- function () {
    print("Hello World!")
}

hw()
```

Set Build Commands

#	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.

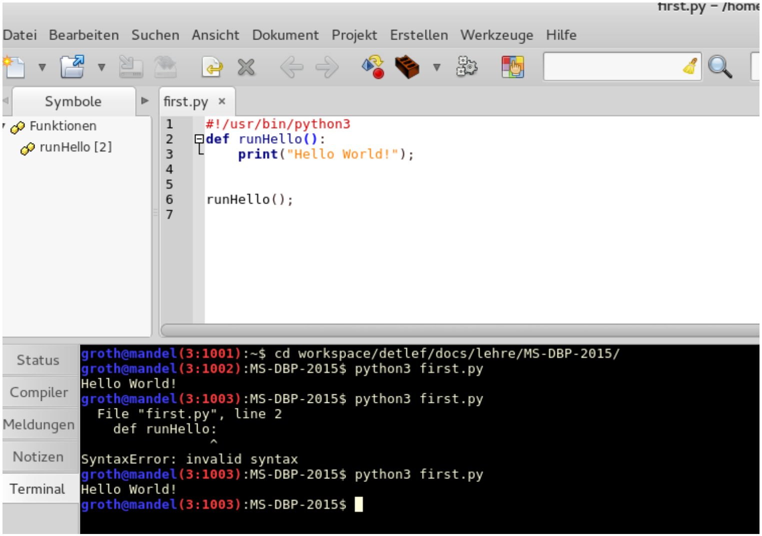
#	Label	Command	Working directory	Reset
Execute commands				
1.	Execute	./%e		
2.				

%d, %e, %f, %p, %l are substituted in command and directory fields, see manual for details.

Cancel OK

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

Editor Setup: Geany - Python

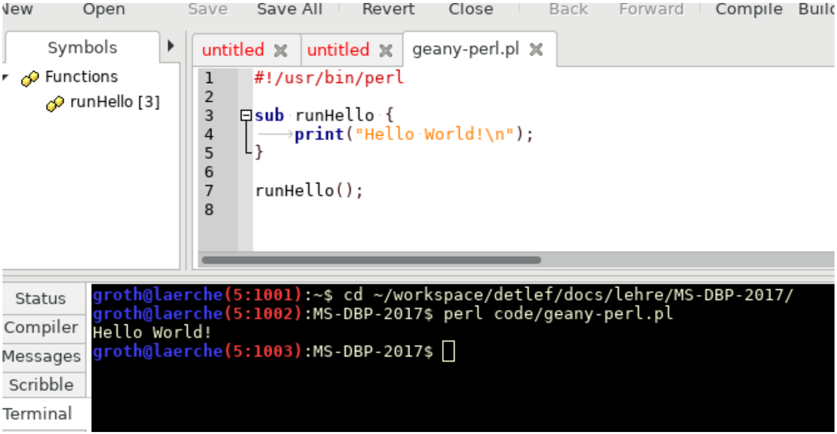


```python
#!/usr/bin/python3
def runHello():
    print("Hello World!");


runHello();
```

```
groth@mandel(3:1001):~$ cd workspace/detlef/docs/lehre/MS-DBP-2015/
groth@mandel(3:1002):MS-DBP-2015$ python3 first.py
Hello World!
groth@mandel(3:1003):MS-DBP-2015$ python3 first.py
  File "first.py", line 2
    def runHello:
               ^
SyntaxError: invalid syntax
groth@mandel(3:1003):MS-DBP-2015$ python3 first.py
Hello World!
groth@mandel(3:1003):MS-DBP-2015$
```

Editor Setup: Geany - Perl



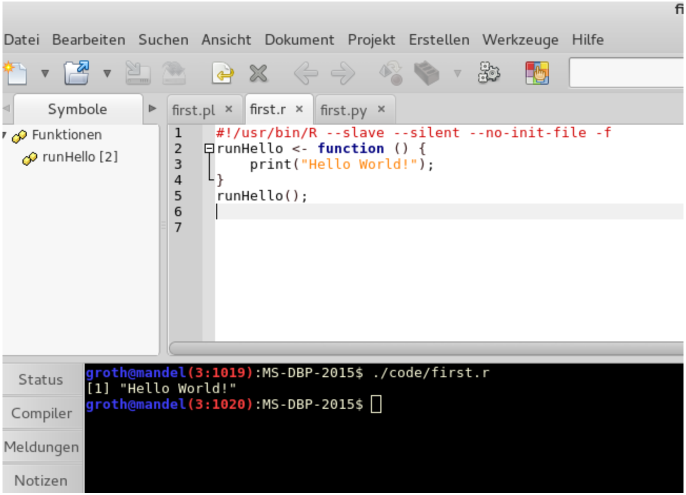
```perl
#!/usr/bin/perl

sub runHello {
	print("Hello World!\n");
}

runHello();
```

```
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



```
#!/usr/bin/R --slave --silent --no-init-file -f
runHello <- function () {
    print("Hello World!");
}
runHello();
```

```
groth@mandel(3:1019):MS-DBP-2015$ ./code/first.r
[1] "Hello World!"
groth@mandel(3:1020):MS-DBP-2015$
```

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

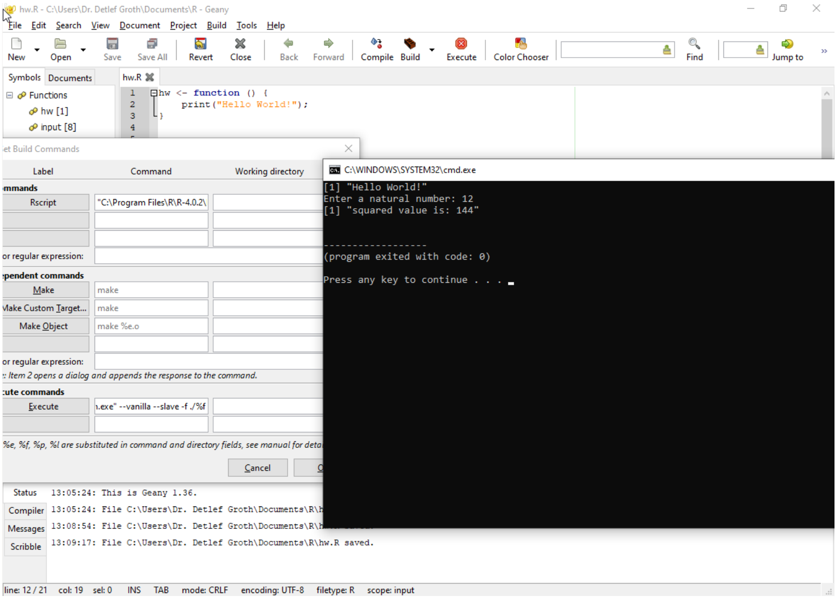
```
#!/bin/sh
# next line is executed by /bin/sh
echo 'Hello World!'
```

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


hw.R - C:\Users\Dr. Detlef Groth\Documents\R - Geany
File Edit Search View Document Project Build Tools Help
New Open Save Save All Revert Close Back Forward Compile Build Execute Color Chooser Find Jump to
Symbols Documents
Functions
hw [1]
input [8]
Status
Compiler
Messages
Scribble

```
hw <- function () {
    print("Hello World!");
}
```

et Build Commands

Label	Command	Working directory
mmands		
Rscript	"C:\Program Files\R\R-4.0.2\	
or regular expression:		
pendent commands		
Make	make	
Make Custom Target...	make	
Make Object	make %e.o	
or regular expression:		
Execute	ı.exe" --vanilla --slave -f ./%f	

: Item 2 opens a dialog and appends the response to the command.

%e, %f, %p, %l are substituted in command and directory fields, see manual for deta

Cancel

C:\WINDOWS\SYSTEM32\cmd.exe

```
[1] "Hello World!"
Enter a natural number: 12
[1] "squared value is: 144"


------------------
(program exited with code: 0)

Press any key to continue . . .
```

```
13:05:24: This is Geany 1.36.
13:05:24: File C:\Users\Dr. Detlef Groth\Documents\R\h
13:08:54: File C:\Users\Dr. Detlef Groth\Documents\R\h
13:09:17: File C:\Users\Dr. Detlef Groth\Documents\R\hw.R saved.
```

line: 12 / 21 col: 19 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: R scope: input

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 ;

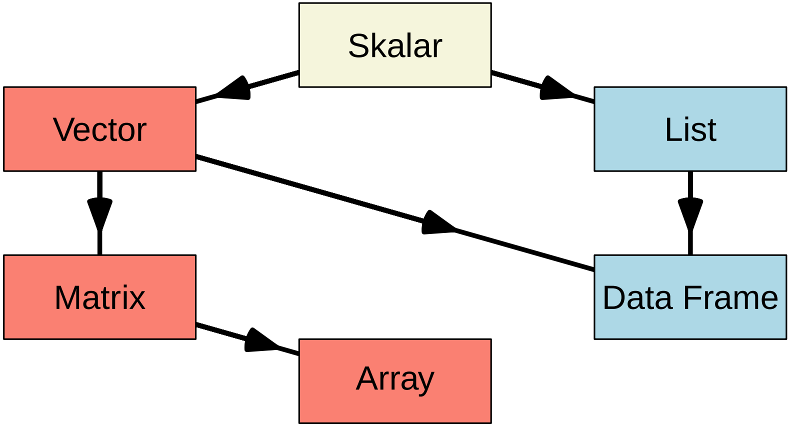
Python	x = 1;
	#inside def
		global globX;
		globX = 1;
		# inside def

Perl	my $x = 1;
	# inside sub
		my $globX = 1;
		our $GLOBX = 1;
		# out of sub ;

$\Rightarrow$ 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	x=1	a=c(1,2,3)	l=list(a=1,b=2)
Perl	$x=1 ;	@a=(1,2,3);	%l=('a'=>1, 'b'=>2);
Python	x=1	a=[1,2,3]	l={'a':1, 'b':2 }

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)
```

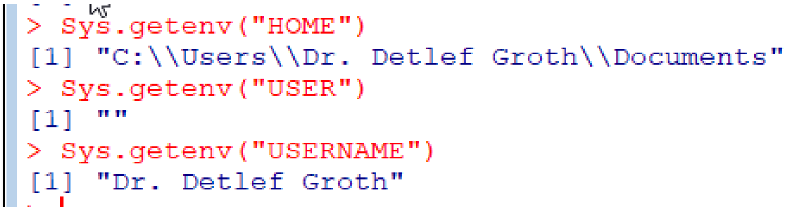
```
[groth@mandel]$ Rscript ../scripts/input.r
Enter a numerical value: 12
[1] 144
```

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"
```

sys.r

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

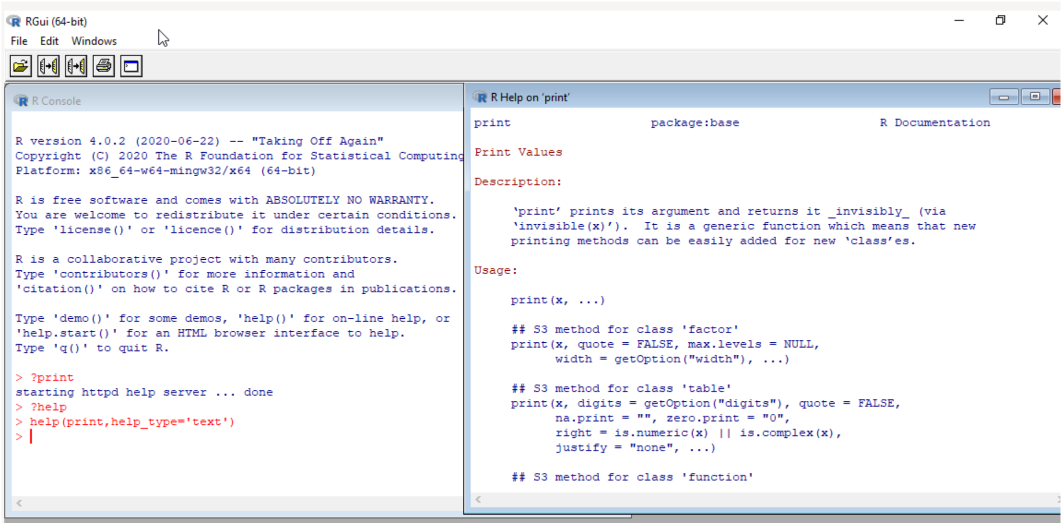
Interactive:

```
> 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



```
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 ?, %*%, :

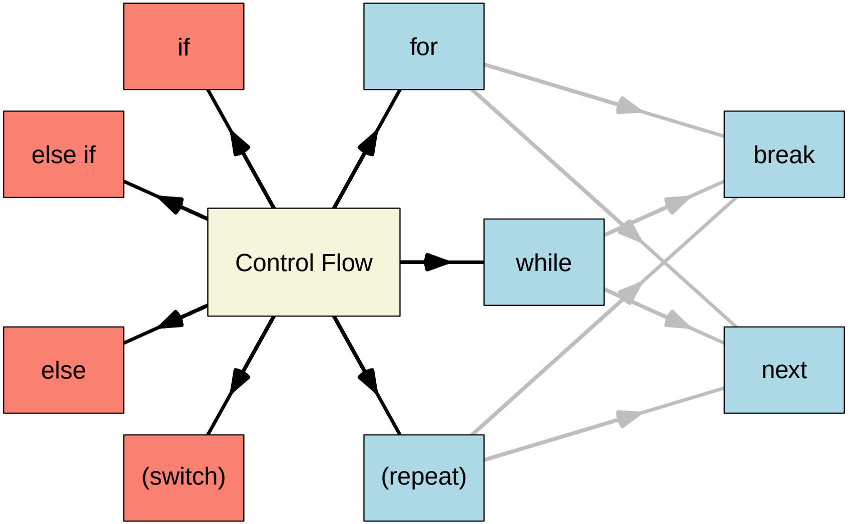
```
> '%ni%' = function (x,y) { !(x %in% y) }
```

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 progammers 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] "2020-09-16 16:32:27 CEST"
⇒ [1] "16"
⇒ [1] "Good afternoon!"
```

condif.R

Placement of Curly Braces

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

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

```
if (cond)
{
   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

$\Rightarrow$ 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))
```

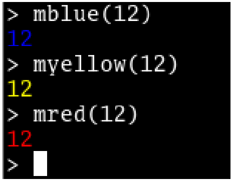
```
⇒ [1] 4
⇒ [1] 16
⇒ [1] 81
```

func.R

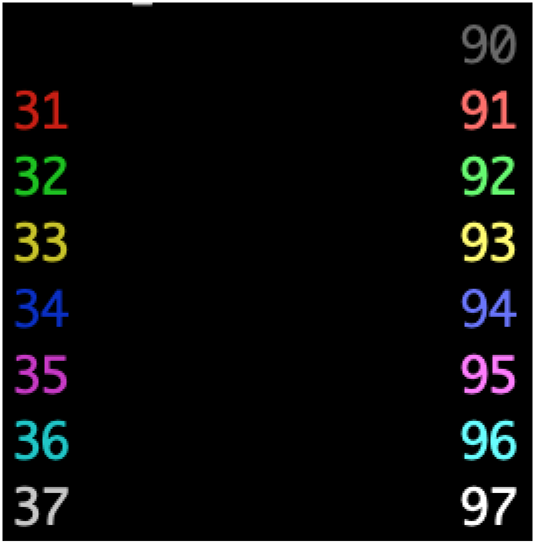
1.11 Colored Terminal

⇒ 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
> 
```

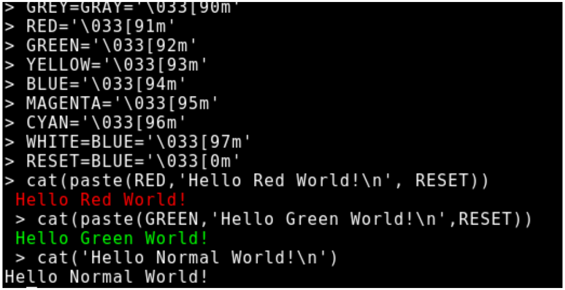


```
           90
31         91
32         92
33         93
34         94
35         95
36         96
37         97
```

⇒ see also R-library crayon

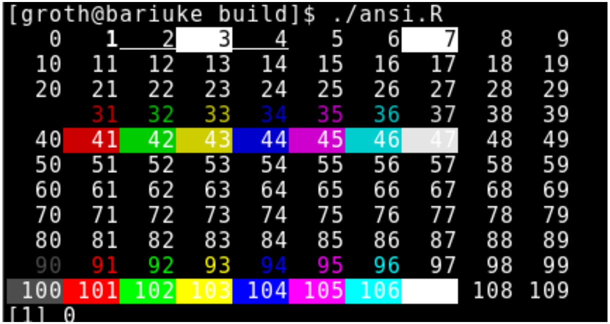
Colored Terminal with global variables

```
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       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 a 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 begining 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

```r
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/`
chapters until decision making and loops

Programming with R
Day 2 - Functions
University of Potsdam

Detlef Groth

2020

Last Lecture

- install R
- install Geany
- variables
- global, local scope (functions)
- data types
- data structures
- help
- ANSI colors

2 Functions

Outline

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
- R base graphics system

Day 4 (advanced)

- using packages
- writing packages
- package documentation

Day 5 (advanced)

- graphical user interfaces
- tcltk (shiny)

2.1 Definition and code organization

2.1.1 Definition

- **function** keyword in R
- functions are used to structure code
- organize your code in a central place
- goal: code reuse
- functions can get *arguments*, sometimes also called *parameters*
- arguments are processed inside the function
- arguments can be mandatory or optional
- R supports named arguments, flexible order possible
- 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]

Functions

- keyword (R function, Python def, Perl sub, ...)
- function name
- argument definition(s) - input by caller
- function body - the implementation
- eventually a return statement

Functions: R

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

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

```r
#!/usr/bin/env Rscript
func.name = function (mand.arg,optional.arg=2) {
    return(mand.arg^optional.arg);
}
print(func.name(2))
print(func.name(2,2))
print(func.name(2,4))
print(func.name(optional.arg=4,mand.arg=3))
```

```
⇒ [1] 4
⇒ [1] 4
⇒ [1] 16
⇒ [1] 81
```

func.R

2.1.2 The main function

In applications, not libraries, you should have a function `main` at the end of you code which is the entry point of your programming logic. You can further check if the R-script file was executed directly. So, if called with `R --vanilla -f filename.R` or using Rscript with `Rscript filename.R` `main` will be automatically executed. Languages like C and C++ will automatically call a main function. In R, Python and many other programming languages, this is not required but considered good style and practice.

```
# filename.R
func1 <- function(x,y) {
    # ...
    func2(x+y)
}

func2 <- function(x) { # ... }

main <- function () {
    func1(1,2)
    # ...
}
# check if this is the script sourced
# first in the interpreter
if (sys.nframe() == 0L & !interactive()) {
    main()
}
```

Simple example: main.R

```
#!/usr/bin/Rscript
main <- function () {
    cat("Hello Main!\n");
    # ...
}
# check if this is the main script
if (sys.nframe() == 0L & !interactive()) {
    main()
}
```

```
[groth@bariuke build]$ chmod 755 ../main.R
[groth@bariuke build]$ ../main.R
Hello Main!
```

```
[groth@bariuke build]$ R --vanilla

R version 3.6.3 (2020-02-29) -- "Holding the
Copyright (C) 2020 The R Foundation for Stat
Platform: x86_64-redhat-linux-gnu (64-bit)


...
Type 'q()' to quit R.

> source("../main.R") # no output below
>
```

⇒ The file main.R can be sourced within an interactive session without calling the main function.

2.1.3 Function arguments

- mandatory (required, no default)
- optional (with defaults)
- three dots (takes and delegates any additional argument)

Mandatory arguments

- no default value given in argument definition
- can't be omitted if calling the function

```
#!/usr/bin/env Rscript
incr <- function (x,y) {
    print(paste('x is:',x))
    x=x+y
    return(x)
}
print(incr(2,5))
print(incr(y=5,x=2)) # named: order switch possible
⇒ [1] "x is: 2"
⇒ [1] 7
⇒ [1] "x is: 2"
⇒ [1] 7
```

margs.R

Optional arguments

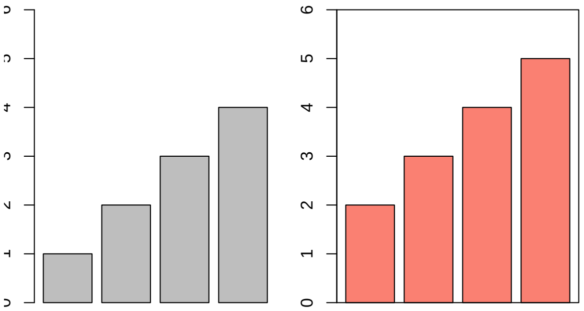
- optional arguments have a useful default
- can be omitted, then default is used

```
#!/usr/bin/env Rscript
mround <- function (x,digit=3) {
    return(round(x,digit))
}
print(mround(2.12345))
print(mround(2.12345,2))
print(mround(2.12345,digit=2)) # prefered way
print(mround(digit=2,x=2.12345))
⇒ [1] 2.123
⇒ [1] 2.12
⇒ [1] 2.12
⇒ [1] 2.12
```

margs2.R

The delegation argument: ...

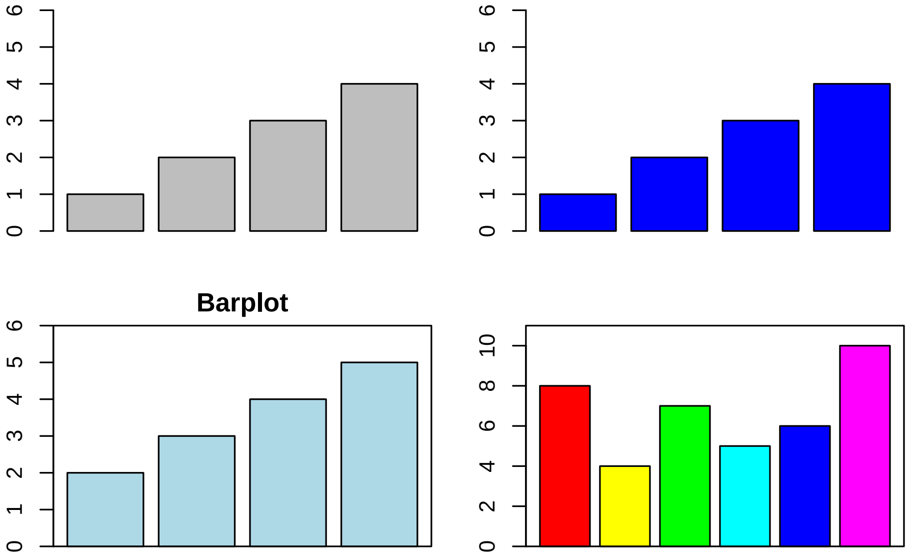
```
> par(mfrow=c(1,2),mai=c(0.3,0.3,0.3,0.3))
> mbarplot = function (data,col='salmon',...) {
+     barplot(data,col=col,...)
+     box()
+ }
> barplot(1:4,
+     ylim=c(0,6))
> mbarplot(2:5,
+     ylim=c(0,6))
```



Same as PDF

```
#!/usr/bin/env Rscript
pdf('barplot.pdf',width=6,height=4)
par(mfrow=c(2,2),mai=c(0.3,0.3,0.3,0.3))
mbarplot = function (data,col='salmon',...) {
    barplot(data,col=col,...)
    box()
}
barplot(1:4,ylim=c(0,6))
barplot(1:4,ylim=c(0,6),col='blue')
mbarplot(2:5,ylim=c(0,6),col='light blue',
  main='Barplot')
mbarplot(sample(1:10,6), col=rainbow(6),ylim=c(0,11))
dev.off()
⇒null device
⇒          1
```

barplot.R


Barplot
0
1
2
3
4
5
6
0
2
4
6
8
10

Summary demonstration

```
#!/usr/bin/env Rscript
f1 <- function (z) {
    print('inside f1')
    print(z^2)
}
main <- function (x, y=2, ...) {
    print('inside main')
    print(x^2+y)
    if (!missing(...)) {
        f1(...)
    }
}
if (sys.nframe() == 0L & !interactive()) {
    main(2)
    main(2,10)
    main(2,10,4)
    main(z=4,y=10,x=2)
}
```

```
⇒ [1] "inside main"
⇒ [1] 6
⇒ [1] "inside main"
⇒ [1] 14
⇒ [1] "inside main"
⇒ [1] 14
⇒ [1] "inside f1"
⇒ [1] 16
⇒ [1] "inside main"
⇒ [1] 14
⇒ [1] "inside f1"
⇒ [1] 16
```

args.R

⇒ prepare your Geany snippets!

2.2 File IO

- input
  - files
  - stdin
- output:
  - files
  - stdout
  - stderr

read.table

⇒ Standard reading of csv files.

- read.table
- read.csv
- read.csv2
- read.<TAB> ...

GFF files

```
> sdata=read.table(
+   "../../data/uniprot-proteome-UP000000354.gff",
+     sep="\t")
> dim(sdata)
[1] 292  10
> sdata[1:4,1:5]
      V1        V2                 V3 V4 V5
1 P59637 UniProtKB              Chain  1 76
2 P59637 UniProtKB Topological domain  1 16
3 P59637 UniProtKB      Transmembrane 17 37
4 P59637 UniProtKB Topological domain 38 76
> table(sdata[,'V1'],sdata[,'V3'])[1:4,1:3]
         Beta strand Chain Coiled coil
  P0C6X7           0    15           0
  P59594          74     0           2
  P59595          12     1           0
  P59596           0     1           0
```

openxlsx::read.xlsx

- Can be used to read and write Excel xlsx files.
- Xlsx files are standardized spreadsheet files.
- There are many packages which can read Excel files.
- openxlsx is my favourite as it is very fast, nicely documented and can be used event to embed plots into output Excel files.

$\Rightarrow$ read.xlsx and read.table are great for tabular data which fit into memory, but what about files which are huge and/or which are not in tabular format?

Non-tabular data: How to copy a file?

⇒ let's ignore that we have `file.copy` in R

```
#!/usr/bin/env Rscript
fin  = file('../main.R', "r")
fout = file('main-copy.R',"w")
while (length((line = readLines(fin,n=1L)))>0) {
    cat(paste(line,"\n",sep=""),file=fout)
}
close(fin)
close(fout)
print('Copy of file made!')
⇒ [1] "Copy of file made!"
```

fcopy.R

```
#!/bin/sh
ls -l main-copy.R ../main.R
⇒-rwxr-xr-x. 1 groth groth 174 Sep 16 16:32 main-copy.R
⇒-rwxr-xr-x. 1 groth groth 174 Sep 11 13:56 ../main.R
```

diff.sh

Geany snippets

```
fin=fin = file('%cursor%','r')\nwhile(length((
    line=readLines(fin,n=1L)))>0) {\n   print(line)\n
    }\nclose(fin)
fout=fout=file('%cursor%','w')\ncat('Hello\n',
   file=fout)\nclose(fout)\n
```

⇒ remove returns on first definition line

Example input

```
#!/bin/sh
head -n 5 ../../data/uniprot-proteome-UP000000354.* | cut
⇒ ==> ../../data/uniprot-proteome-UP000000354.fasta
⇒ >sp|P59637|VEMP_CVHSA Envelope small membrane prot
⇒ MYSFVSEETGTLIVNSVLLFLAFVVFLLVTLAILTALRLCAYCCNIVNVS
⇒ RVKNLNSSEGVPDLLV
⇒ >sp|P59596|VME1_CVHSA Membrane protein OS=Human SA
⇒ MADNGTITVEELKQLLEQWNLVIGFLFLAWIMLLQFAYSNRNRFLYIIKL
⇒
⇒ ==> ../../data/uniprot-proteome-UP000000354.gff <=
⇒ ##gff-version 3
⇒ ##sequence-region P59637 1 76
⇒ P59637        UniProtKB       Chain           1       76
⇒ P59637        UniProtKB       Topological domain
⇒ P59637        UniProtKB       Transmembrane           17
```

fasta.sh

R: file.show

```
#!/usr/bin/env Rscript
file.show('../../data/uniprot-proteome-UP000000354.fasta'
    ,pager='head')
⇒ >sp|P59637|VEMP_CVHSA Envelope small membrane protein O
⇒ MYSFVSEETGTLIVNSVLLFLAFVVFLLVTLAILTALRLCAYCCNIVNVSLVKPT
⇒ RVKNLNSSEGVPDLLV
⇒ >sp|P59596|VME1_CVHSA Membrane protein OS=Human SARS co
⇒ MADNGTITVEELKQLLEQWNLVIGFLFLAWIMLLQFAYSNRNRFLYIIKLVFLWL
⇒ LACFVLAAVYRINWVTGGIAIAMACIVGLMWLSYFVASFRLFARTRSMWSFNPET
⇒ VPLRGTIVTRPLMESELVIGAVIIRGHLRMAGHSLGRCDIKDLPKEITVATSRTL
⇒ GASQRVGTDSGFAAYNRYRIGNYKLNTDHAGSNDNIALLVQ
⇒ >sp|Q7TLC7|Y14_CVHSA Uncharacterized protein 14 OS=Huma
⇒ MLPPCYNFLKEQHCQKASTQREAEAAVKPLLAPHHVVAVIQEIQLLAAVGEILLL
```

fasta.R

But this does not work on windows!

file and dir commands

```
> file.<TAB>
file.access  file.copy   file.exists  file.mode
file.remove  file.size   file.append  file.create
file.info    file.mtime  file.rename  file.symlink
file.choose  file.edit   file.link    file.path
file.show
> dir.<TAB>
dir.create  dir.exists
```

⇒ Exercise: Write a `file.head` function which displays the first 6 lines of a file regardless of the pager! So, it should work as well on Windows.

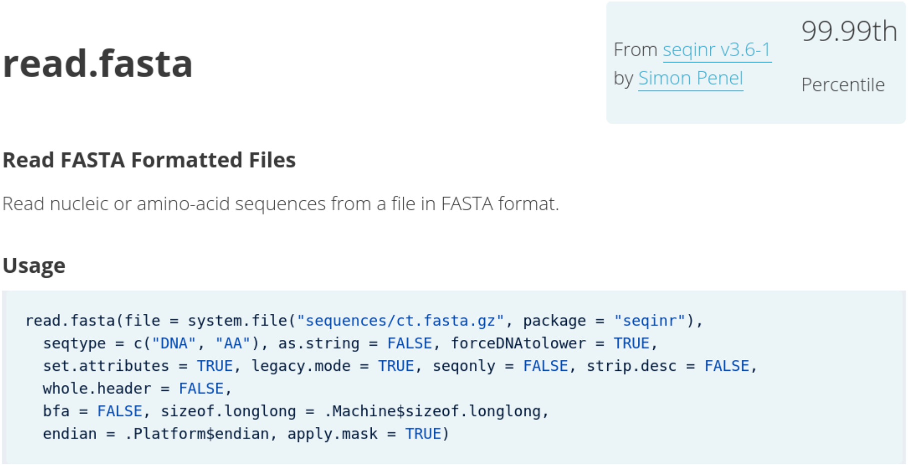
file.head

```
#!/usr/bin/env Rscript
file.head = function (filename,n=6) {
  if (!file.exists(filename)) {
    stop(paste('Error! File',filename,'does not exist!'))
  }
  fin=file(filename,'r')
  i=0
  while(length((line=readLines(fin,n=1L)))>0) {
     i = i +1
     cat(line,'\n')
     if (i == n) {
        break
     }
  }
  close(fin)
}
file.head('../../data/uniprot-proteome-UP000000354.fasta'
          ,n=3)
```

```
⇒ >sp|P59637|VEMP_CVHSA Envelope small membrane protein O
⇒ MYSFVSEETGTLIVNSVLLFLAFVVFLLVTLAILTALRLCAYCCNIVNVSLVKPT
⇒ RVKNLNSSEGVPDLLV
```

fhead.R

seqinr::read.fasta



read.fasta

From seqinr v3.6-1
by Simon Penel

99.99th
Percentile

Read FASTA Formatted Files

Read nucleic or amino-acid sequences from a file in FASTA format.

Usage

```
read.fasta(file = system.file("sequences/ct.fasta.gz", package = "seqinr"),
  seqtype = c("DNA", "AA"), as.string = FALSE, forceDNAtolower = TRUE,
  set.attributes = TRUE, legacy.mode = TRUE, seqonly = FALSE, strip.desc = FALSE,
  whole.header = FALSE,
  bfa = FALSE, sizeof.longlong = .Machine$sizeof.longlong,
  endian = .Platform$endian, apply.mask = TRUE)
```

$\Rightarrow$ In the course we will write our own FASTA package

2.3 Command line arguments

- As you submit arguments to functions you can submit arguments to your application on the terminal.
- Example: `Rscript app.R arg1 arg2 arg3`
- Goal: Set variables in your application at runtime, not by writing the values into the source code.
- Example:
  `firefox https://www.uni-potsdam.de`
- Firefox has not hardcoded the URL into its source code, but you can still load the Uni web page

C and command line arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.
The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly.

`https://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm`

R: commandArgs - function

```
> ?commandArgs
commandArgs          package:base         R Documentation

Extract Command Line Arguments

Description:
 Provides access to a copy of the command line
 arguments supplied when this R session was invoked.

Usage:
     commandArgs(trailingOnly = FALSE)

Arguments:
  trailingOnly: logical. Should only arguments after
  ?--args? be returned?
```

commandArgs example

```
#!/usr/bin/env Rscript
options(width=55)
print('All:')
print(commandArgs())
print('Trailing Only:')
print(commandArgs(trailingOnly=TRUE))
⇒ [1] "All:"
⇒ [1] "/usr/lib64/R/bin/exec/R" "--slave"
⇒ [3] "--no-restore"            "--file=./commandArgs.R"
⇒ [1] "Trailing Only:"
⇒ character(0)
```

commandArgs.R

```
#!/usr/bin/sh
./commandArgs.R some arg --arg1 value1--args2 val2
⇒ [1] "All:"
⇒  [1] "/usr/lib64/R/bin/exec/R"
⇒  [2] "--slave"
⇒  [3] "--no-restore"
⇒  [4] "--file=./commandArgs.R"
⇒  [5] "--args"
⇒  [6] "some"
⇒  [7] "arg"
⇒  [8] "--arg1"
⇒  [9] "value1--args2"
⇒ [10] "val2"
⇒ [1] "Trailing Only:"
⇒ [1] "some"            "arg"             "--arg1"
⇒ [4] "value1--args2" "val2"
```

commandArgs.sh

⇒ Access to arguments by index!

More complex example

```
#!/usr/bin/env Rscript
options(width=55)
getArgs = function () {
    l=list()
    args=commandArgs(trailingOnly=TRUE)
    i = 0
    for (arg in args) {
        i = i +1
        if (grepl('^--',arg)) {
            l[[arg]]=args[i+1]
            next
        }
    }
    return(l)
}
args=getArgs()
print(args)
print(args[['--infile']])
```

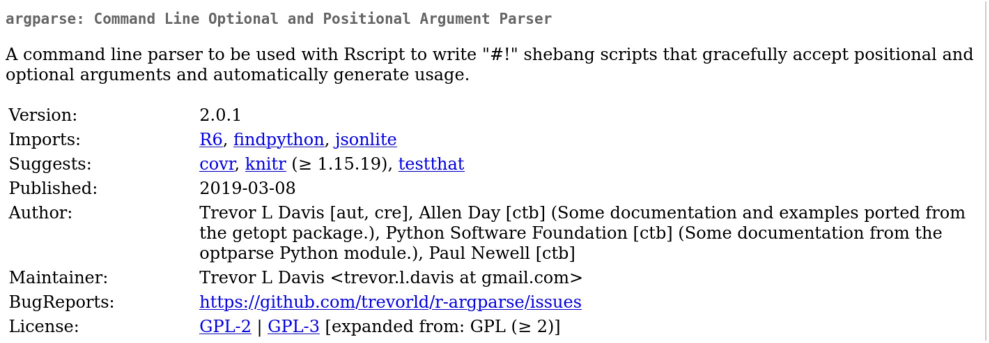
```
⇒ list()
⇒ NULL
```

commandArgs2.R

```
#!/usr/bin/sh
./commandArgs2.R --infile test.fasta --outfile out.txt \
    --arg1 value1
⇒ $`--infile`
⇒ [1] "test.fasta"
⇒
⇒ $`--outfile`
⇒ [1] "out.txt"
⇒
⇒ $`--arg1`
⇒ [1] "value1"
⇒
⇒ [1] "test.fasta"
```

commandArgs2.sh

R library argparse (needs Python!!)



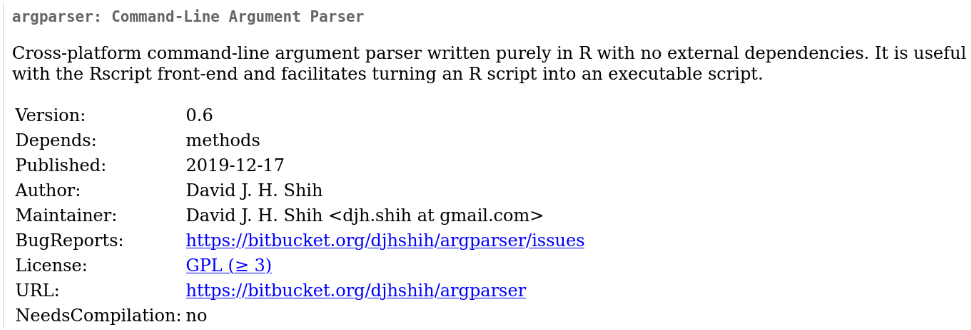
argparse: Command Line Optional and Positional Argument Parser

A command line parser to be used with Rscript to write "#!" shebang scripts that gracefully accept positional and optional arguments and automatically generate usage.

Version:	2.0.1
Imports:	R6, findpython, jsonlite
Suggests:	covr, knitr (≥ 1.15.19), testthat
Published:	2019-03-08
Author:	Trevor L Davis [aut, cre], Allen Day [ctb] (Some documentation and examples ported from the getopt package.), Python Software Foundation [ctb] (Some documentation from the optparse Python module.), Paul Newell [ctb]
Maintainer:	Trevor L Davis <trevor.l.davis at gmail.com>
BugReports:	https://github.com/trevorld/r-argparse/issues
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]

⇒ this R-package has dubious dependencies!
⇒ integrating several programming languages is fun first but ends with pain later ...

R library argparser (no dependencies!!)



argparser: Command-Line Argument Parser

Cross-platform command-line argument parser written purely in R with no external dependencies. It is useful with the Rscript front-end and facilitates turning an R script into an executable script.

Version:	0.6
Depends:	methods
Published:	2019-12-17
Author:	David J. H. Shih
Maintainer:	David J. H. Shih <djh.shih at gmail.com>
BugReports:	https://bitbucket.org/djhshih/argparser/issues
License:	GPL (≥ 3)
URL:	https://bitbucket.org/djhshih/argparser
NeedsCompilation:	no

⇒ KISS: avoid packages with many dependencies if you can!
⇒ No installation waiting on users computer if you install your script

Argparser example:

```
#!/usr/bin/env Rscript
# install.packages('argparser')
library(argparser)
p <- arg_parser("A text file modifying program")
# Add a positional argument
p <- add_argument(p, "input", help="input file")
# Add an optional argument
p <- add_argument(p, "--output", help="output file", defa
# Add a flag
p <- add_argument(p, "--append", help="append to file", f
# Add multiple arguments together
p <- add_argument(p,
    c("ref", "--date", "--sort"),
    help = c("reference file", "date stamp to use", "sort
    flag = c(FALSE, FALSE, TRUE))
print(p) # goes to stderr :(
print('done')
```

```
⇒ [1] "done"
```

argparser.R

```
[groth@bariuke build]$ ./argsparser.R
usage: ./argsparser.R [--] [--help] [--append]
       [--sort] [--opts OPTS] [--output OUTPUT]
       [--date DATE] input ref

A text file modifying program

positional arguments:
  input                 input file
  ref                   reference file

flags:
  -h, --help            show this help message and exit
  -a, --append          append to file
  -s, --sort            sort lines
```

```
optional arguments:
  -x, --opts OPTS      RDS file containing argument values
  -o, --output OUTPUT output file [default: output.txt]
  -d, --date DATE      date stamp to use
```

⇒ The argparser should support all things you over need.
⇒ Only GPL might be an issue if you work in a company.
⇒ I have my own small object oriented implementation, MIT licensed (argvparse.R).

Summary

Standard R:

- commandArgs
- parse by position/index
- trailingOnly option

Library argparser:

- if you need more
- double dash and short option support
- `arg_parser`
- `add_argument`
- prints on stderr (I send an issue request for this)

⇒ We focus here on `commandArgs` and eventually my getArgs function shown above.

2.4 Terminal applications

User interfaces

Command line interfaces (CLI):

- simple to implement but powerful
- sometimes tricky for ordinary users to use
- easy pipelining of applications
- good for automation

Graphical user interface (GUI):

- more difficult to implement
- easier to use for ordinary users
- difficult to automate and to connect to other tools

Guidelines

- provide a help message if not the right arguments are given
- help message should give:
  - application name
  - short description what the program does
  - available options with understandable messages
  - (version number)
  - (author name and company)
  - (license, PD, GPL or MIT)

⇒ Even if you are the only user of the program (first three points)!
⇒ You should not need to read the source code to know what the program is doing

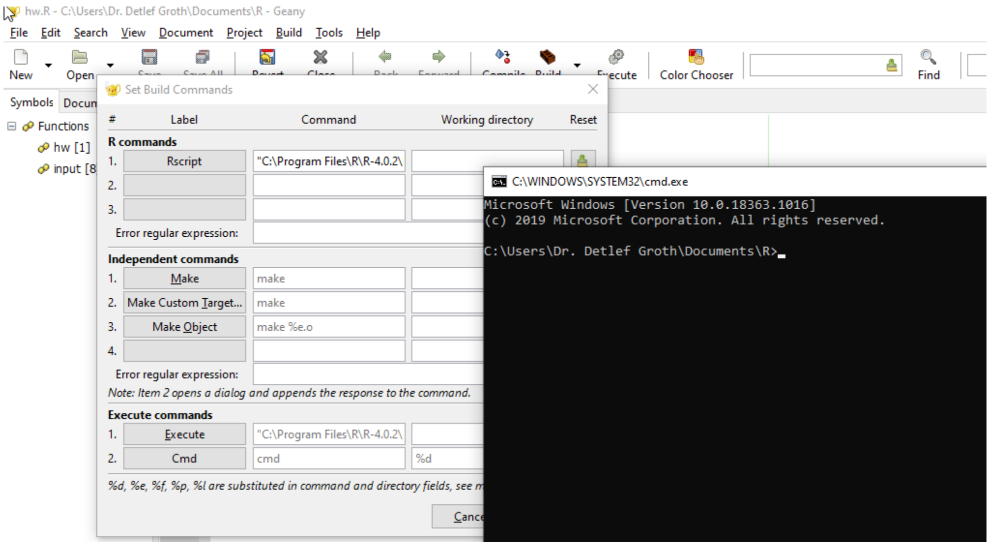
General application outline

Homework: Create a snippet app which outlines a standard application with the following code:

```
#!/usr/bin/env Rscript
usage <- function () {
    cat("Application name by Author, Uni Potsdam, 20
    cat("Usage: appname --arg value\n")
}
main <- function (args) {

}
if (sys.nframe() == 0L & !interactive()) {
    main(commandArgs(trailingOnly=TRUE)
}
```

Geany and cmd - Terminal



hw.R - C:\Users\Dr. Detlef Groth\Documents\R - Geany
File Edit Search View Document Project Build Tools Help
New Open Save Save All Revert Close Back Forward Compile Build Execute Color Chooser Find
Symbols Docum
Functions
hw [1]
input [8
Set Build Commands

#	Label	Command	Working directory	Reset
R commands				
1.	Rscript	"C:\Program Files\R\R-4.0.2\		
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.

#	Label	Command	Working directory	Reset
Execute commands				
1.	Execute	"C:\Program Files\R\R-4.0.2\		
2.	Cmd	cmd	%d	

%d, %e, %f, %p, %l are substituted in command and directory fields, see m

C:\WINDOWS\SYSTEM32\cmd.exe

```
Microsoft Windows [Version 10.0.18363.1016]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Dr. Detlef Groth\Documents\R>_
```

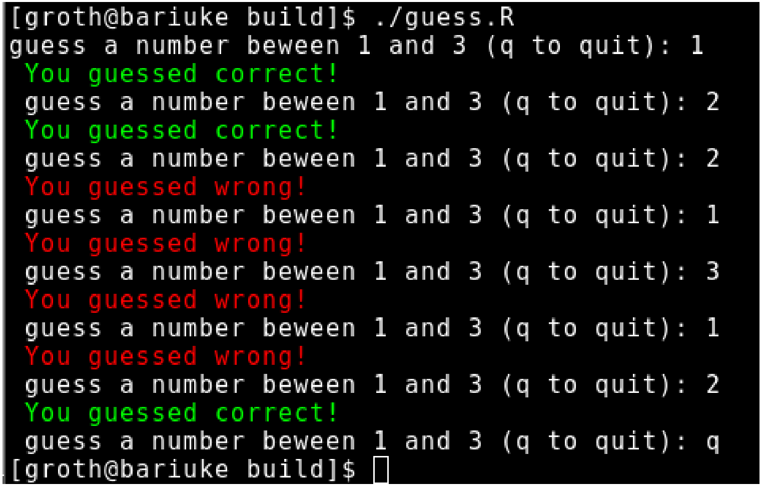
Terminal Menu

⇒ How to write a menu for the terminal: `while (TRUE)`

```
#!/usr/bin/env Rscript
source('../scripts/terminal.R')
menu = function () {
  while (TRUE) {
    number=input('guess a number beween 1 and 3
      (q to quit): ')
    if (number == 'q') {
       break
    }
    if (as.integer(number) == sample(1:3,1)) {
      cat(GREEN,'You guessed correct!\n',RESET);
    } else {
      cat(RED,'You guessed wrong!\n',RESET);
    }
  }
}
```

```
#menu()
print('done');
⇒ [1] "done"
```

guess.R



```
[groth@bariuke build]$ ./guess.R
guess a number beween 1 and 3 (q to quit): 1
 You guessed correct!
 guess a number beween 1 and 3 (q to quit): 2
 You guessed correct!
 guess a number beween 1 and 3 (q to quit): 2
 You guessed wrong!
 guess a number beween 1 and 3 (q to quit): 1
 You guessed wrong!
 guess a number beween 1 and 3 (q to quit): 3
 You guessed wrong!
 guess a number beween 1 and 3 (q to quit): 1
 You guessed wrong!
 guess a number beween 1 and 3 (q to quit): 2
 You guessed correct!
 guess a number beween 1 and 3 (q to quit): q
[groth@bariuke build]$ 
```

terminal.R

```
input = function (prompt="Enter: ") {
    if (interactive() ){
        return(readline(prompt))
    } else {
        cat(prompt);
        return(readLines("stdin",n=1))
    }
}
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'
```

Splitting code in files

How to organize applications where the code is splitted into different files?
We would like to reuse functions and may be even use global variables over several files.
Solutions:

- `library`: assemble several files into a library (Day 4)
- `source`: load individual R source code files
- Problem with source is: Where is the file to find in dependence from the working directory?
- I would like to be able to source files in a directive relative to the current file.
- There is no inbuild solution, but a few workarounds.

The problem

```
#!/usr/bin/env Rscript
message("Hello")
source("other.R")
```

⇒ This does only work if we are in the directory where ⇒ other.R is.

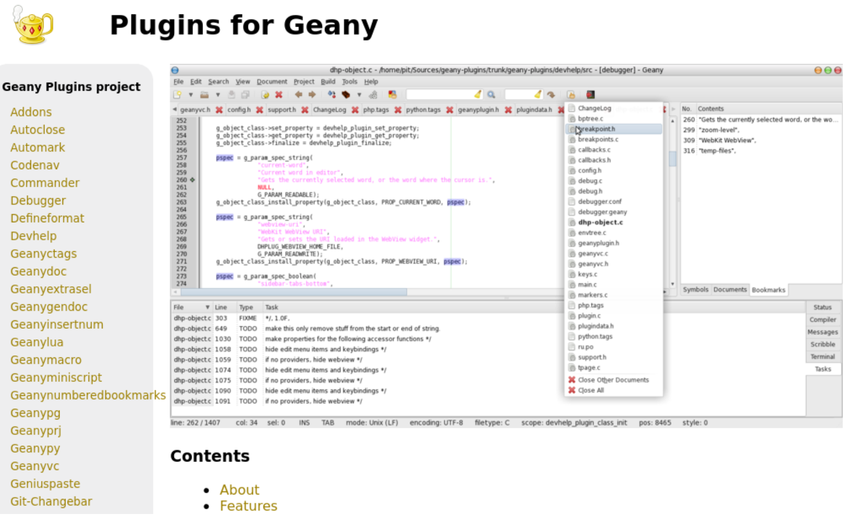
Workaround where is the currently sourced file?

```
thisFile <- function() {
    cmdArgs <- commandArgs(trailingOnly = FALSE)
    needle <- "--file="
    match <- grep(needle, cmdArgs)
    if (length(match) > 0) {
        # Rscript
        return(normalizePath(sub(needle, "",
            cmdArgs[match])))
    } else {
        # 'source'd via R console
        return(normalizePath(sys.frames()[[1]]$ofile
    }
}
```

Alternative

- Create one single big R-file
- concat all your R files into a single R file to make a running application
- ...

Geany Plugins



Plugins for Geany

Geany Plugins project

- Addons
- Autoclose
- Automark
- Codenav
- Commander
- Debugger
- Defineformat
- Devhelp
- Geanyctags
- Geanydoc
- Geanyextrasel
- Geanygendoc
- Geanyinsertnum
- Geanylua
- Geanymacro
- Geanyminiscript
- Geanynumberedbookmarks
- Geanypg
- Geanyprj
- Geanypy
- Geanyvc
- Geniuspaste
- Git-Changebar

dhp-object.c - /home/pit/Sources/geany-plugins/trunk/geany-plugins/devhelp/src - [debugger] - Geany

```
    g_object_class->set_property = devhelp_plugin_set_property;
    g_object_class->get_property = devhelp_plugin_get_property;
    g_object_class->finalize = devhelp_plugin_finalize;

    pspec = g_param_spec_string(
                "current-word",
                "Current word in editor",
                "Gets the currently selected word, or the word where the cursor is.",
                NULL,
                G_PARAM_READABLE);
    g_object_class_install_property(g_object_class, PROP_CURRENT_WORD, pspec);

    pspec = g_param_spec_string(
                "webview-uri",
                "WebKit WebView URI",
                "Gets or sets the URI loaded in the WebView widget.",
                DHPLUG_WEBVIEW_HOME_FILE,
                G_PARAM_READWRITE);
    g_object_class_install_property(g_object_class, PROP_WEBVIEW_URI, pspec);

    pspec = g_param_spec_boolean(
                "sidebar-tabs-bottom",
```

No.	Contents
260	"Gets the currently selected word, or the wo...
299	"zoom-level",
309	"WebKit WebView",
316	"temp-files",

File	Line	Type	Task
dhp-object.c	303	FIXME	*/, 1.0F,
dhp-object.c	649	TODO	make this only remove stuff from the start or end of string.
dhp-object.c	1030	TODO	make properties for the following accessor functions */
dhp-object.c	1058	TODO	hide edit menu items and keybindings */
dhp-object.c	1059	TODO	if no providers, hide webview */
dhp-object.c	1074	TODO	hide edit menu items and keybindings */
dhp-object.c	1075	TODO	if no providers, hide webview */
dhp-object.c	1090	TODO	hide edit menu items and keybindings */
dhp-object.c	1091	TODO	if no providers, hide webview */

```
line: 262 / 1407    col: 34    sel: 0    INS    TAB    mode: Unix (LF)    encoding: UTF-8    filetype: C    scope: devhelp_plugin_class_init    pos: 8465    style: 0
```

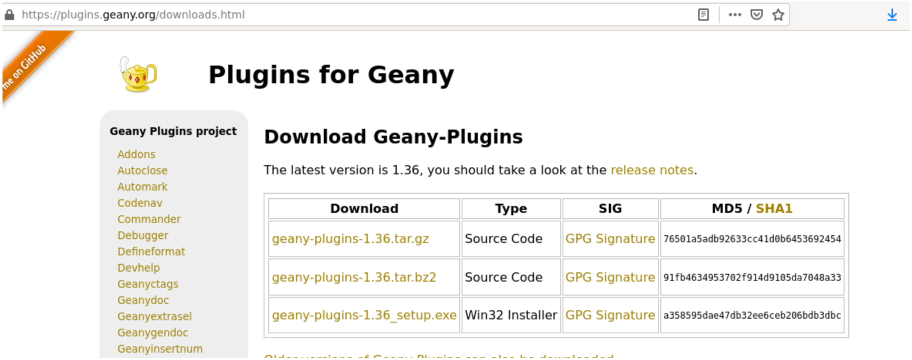
Contents

- About
- Features

Plugins Install

- Fedora: `sudo dnf install geany-plugins-*`
- CentOS: `yum install geany-plugins-*`
- Ubuntu: `sudo apt-get install geany-plugins`
- Windows: no package manager, use the download page
- Mac OSX I don't know, build them yourself?

Geany Plugins Download



Plugins for Geany

Geany Plugins project

- Addons
- Autoclose
- Automark
- Codenav
- Commander
- Debugger
- Defineformat
- Devhelp
- Geanyctags
- Geanydoc
- Geanyextrasel
- Geanygendoc
- Geanyinsertnum

Download Geany-Plugins

The latest version is 1.36, you should take a look at the release notes.

Download	Type	SIG	MD5 / SHA1
geany-plugins-1.36.tar.gz	Source Code	GPG Signature	76501a5adb92633cc41d0b6453692454
geany-plugins-1.36.tar.bz2	Source Code	GPG Signature	91fb4634953702f914d9105da7048a33
geany-plugins-1.36_setup.exe	Win32 Installer	GPG Signature	a358595dae47db32ee6ceb206bdb3dbc

2.5 Exercise Functions

Task 1 - Workspace preparation

The goal of todays exercise today is to write a R-script which creates a tabular output file with the two columns sequence-id and sequence-length from a given FASTA file.

Steps:

- start the Geany text editor
- the files from last days exercise should be visbible again
- activate the split window plugin (Tools-Plugin Manager) and assign the keys horizontal split (Ctrl-F2), vertical split (Ctrl-F3) and unsplit (Ctrl-F1)
- Windows: prepare the Cmd tool as described in the lecture for opening a terminal directly from geany in your current folder
- Windows: create a *Rscript.bat* file in this folder to execute directly your commands in this folder:

`"C:\Program Files\R\R-4.0.2\bin\x64\Rscript.exe" %1 %2 %3 %4 %5 %6 %7` The percent signs are place holders for possible command line arguments

- download the Fasta files from Moodle and save them in a directory `data` below of your R script
- look into the FASTA files using geany and rename them to sars-cov1.fasta and sars-cov2.fasta depending on the content
- so your directory stucture should look like this:

```
data
data/sars-cov1.fasta
data/sars-cov2.fasta
hw.R
Rscript.bat (if on Windows)
```

Windows: Try to execute "Rscript.bat hw." in the Cmd-Terminal

Task 2 - Outline application Count number of sequences in FASTA file

Hint: if not done yet create the snippets `fin` and fout for your text editor as shown in the lecture.
Functions to use:

- commandArgs
- file.exists (check if a file exists)

Workflow:

- create a new empty file and save it as fastainfo.R
- add the shebang line '#!/usr/bin/env Rscript' at top of your file even if you are on Windows
- your script might then Run as well on Windows
- add a main and a help function (later to a Geany snippet if not done yet)
- use the commandArgs function with trailingOnly=TRUE to parse command line arguments

- if there is no command line argument (or there is no file with the given name) call the help function

Outline:

```
shebang

help-function

sys.nframe check
  main-function with
     * commandArgs
     * file.exists
```

Task 3 - Implementation: Count sequence lengths in FASTA file

Functions to use:

- file (open a file in read or write mode)
- grepl (check if a text pattern exists in a string, l stands for logical)
- gsub (replacement of text)
- nchar (number of characters)
- to your application add countFasta function
- arguments to this function should be a filename
- again do a check for the file existence (Why again - who knows?)
- open and loop over the file
- if a line starts with a greater sign, checked with grepl we have a header
- we then extract the id using gsub

```
> grepl('^>','MACTR')
[1] FALSE
```

```
> grepl('^>','>ID01 some more text')
[1] TRUE
> gsub('^>([^ ]+).+','\\1','>ID01 some more text')
[1] "ID01"
```

Those are regular expressions, for all the details look at `?regex`.
Expressions in braces can be captured in \\1, \\2, etc
Some examples:

- ˆ - beginning of a string
- [A-Z0-9] any capital letter or number
- [ˆA-Z] any character except capital letters (second meaning of the tegmentum)
- . (dot) any character except newline
- [A-Z]{2} two capital letters in sequence
- [A-Z]+ one or any number of capital letters
- [a-z]* zero or any number of lowercase letters
- [ˆ]{2,5} two to five letters which are not a space

Task 4 - Write data into file

- We now like to write the tabular output into an output file.
- To the function getCounts add an optional argument outfile which defaults to an empty string.
- So something like `getCounts(filename,outfile='')`
- If no outfile is give we should still just write the output to the terminal.
- If an outfile is given we should write our data to this file.
- Add the implementation to write to this outfile to your function
- add check for a second argument to your application into your main function, if this argument exists, add it to the function call.

```
# within main function do this
args=commandArgs(trailingOnly=TRUE)
if (length(args) == 2)) {
    getCounts(args[1],outfile=args[2])
} else if (length(args) == 1) {
    getCounts(args[1])
} else {
    help()
}
```

Homework studies:

- `https://www.tutorialspoint.com/r/r_functions.htm`
- `https://www.tutorialspoint.com/r/r_csv_files.htm`
- `https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf` (Chapter 10 - writing functions)

Tomorrow:

$\Rightarrow$ Object oriented programming!!

$\Rightarrow$ R base graphics system - let's paint!!

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].