Programming with R
Day 3
Object Oriented Programming
Graphics
University of Potsdam

Detlef Groth

Last Lecture

- R function:
  - mandatory
  - optional
  - delegation
- File IO
  - read.table
  - file r, w
- command line arguments, commandArgs
- terminal applications, while(TRUE)

2 Object Oriented Programming

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 progr.**
- **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 Object Oriented Programming

Object Oriented Programming:

... is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse ...

[Wikipedia — The free Encyclopedia, 2020]

Goal of OOP

- simulate real world actions
- create objects with methods and properties
- logic is naturally divided
- Example objects in a statistical analysis:
  - DataReader
    - read.data (patient data from database)
    - create new variables (count amino acid two letter codes, ...)
  - Plotter (amino acid sequences, amino acid statistics)
  - FastaUtils (read fasta, read FASTA data NCBI from website, ...)
  - ...

Terminology

- Class: template for creating objects
- (Class variable: Variables shared by all objects of a class)
- (Class method: functions for the class)
- Object: a certain instance of a class
- Instance variable: a variable for each object (property)
- Instance method: functions working for the objects
- Inheritance: methods & properties inherited from other classes
- Overriding: methods can be overwritten by a child class to give them a new functionality

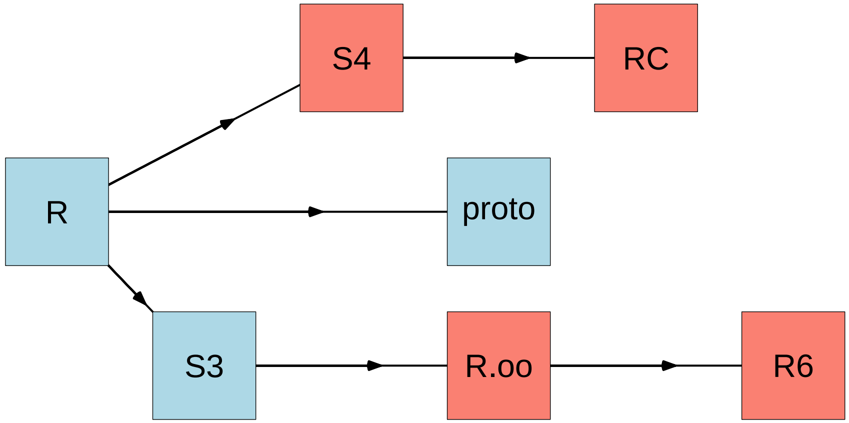
Most R class systems have no support for class variables and class methods. R is more object oriented in this case.

R OOP systems

- **S3** (1986) - core - most often used
- S4 (1992) - core - extension to S3
- RC (2010) - extension to S4 - core
- R6 (2014) - extension to S3 - separate library
- R.oo (2005) - extension to S3 - separate library
- **proto** (2005) - very object oriented - separate library

$\Rightarrow$ we will discuss here **S3** as it is the most widely used OO system in R and **proto**, as this is quite close to the syntax of Python or Perl object oriented systems.

OOP Evolution



2.2 S3

⇒ Syntax: `method(object,args)`

```
> options(continue='  ')
> mike=list(name='Mike',type='animal')
> class(mike)='Animal'
> print(mike)
$name
[1] "Mike"

$type
[1] "animal"

attr(,"class")
[1] "Animal"
```

Create a print.Animal function

```
> print.Animal <- function (self) {
     print(paste('My name is ', self$name,'!',sep=''))
 }
> print(mike)
[1] "My name is Mike!"
```

That is ok! But what about let run Mike:

```
> run.Animal <- function (self) {
    print(paste(self$name,' runs!',sep=''))
 }
> try (run(Mike),silent=FALSE,outFile=stdout())
Error in run(Mike) : could not find function "run"
> methods(class='Animal')
[1] print
see '?methods' for accessing help and source code
```

Need a generic function - run.default

```
> run = function (x,...) UseMethod("run")
> run.default = function (x) {
     print('generic run')
 }
> run('Hi')
[1] "generic run"
> run(521)
[1] "generic run"
> run(list(a=1,b=1:4))
[1] "generic run"
> run(mike)
[1] "Mike runs!"
> methods(class='Animal')
[1] print run
see '?methods' for accessing help and source code
```

Inheritance with S3

```
> susi=list(name='Susi',age=5,sex='lady')
> class(susi)=c('Animal','Cat')
> run(susi)
[1] "Susi runs!"
```

That's nice, let's meow her:

```
> meow.Cat = function (self,n=2) {
     return(paste(self$name,' says ', paste(rep('meow',n)
 }
> meow = function (x,...) UseMethod("meow")
> meow.default = function (x,...) {
     print('generic meow')
 }
> meow(susi,n=4)
[1] "Susi says meow, meow, meow, meow!"
```

Disadvantages of S3

```
> class(susi)
[1] "Animal" "Cat"
> methods(class=class(susi))
[1] print run
see '?methods' for accessing help and source code
> for (clss in class(susi)) {
     print(methods(class=clss))
 }
[1] print run
see '?methods' for accessing help and source code
[1] meow
see '?methods' for accessing help and source code
```

- fine for standard methods like print, plot, summary
- for more specific methods (meow) we need actually to declare three methods each time
- not so easy access to methods of the class

Summary S3

- S3 use method dispatching
- installation procedure:

1) `methname = function (x,...)`
   `UseMethod('methname')`
2) create a default method: `methname.default`
3) create `methname.Classname` for your class

⇒ hint use a snippet for this three step procedure

- if you would like to learn only one OOP system, you should probably learn S3

2.3 Proto

⇒ Syntax: `object$method(args)`

⇒ Proto works on objects.

```
> #install.packages('proto')
> library(proto)
> moritz=proto(name='Moritz',type='rat',age=1)
> ls(moritz)
[1] "age"  "name" "type"
> print(moritz$name)
[1] "Moritz"
> moritz$run = function (self) {
     paste(self$name,' runs!',sep='')
 }
> moritz$run()
[1] "Moritz runs!"
```

The self argument

- first argument of proto method is the object itself
- you can name is as you like
- some people use a dot '.', other 'this'
- I prefere 'self' as self is used as well in Python and Perl by convention
- they took this name from the programming language Self (1986) which was one of the first object oriented languages
- Self, as JavaScript or proto are an OOP style based on prototypes
- prototype programming means we create simple objects first and then extend them stepwise as we need more features

Don’t forget the self

- it is often that we define the self in the method declaration
- a typical error with forgotten self looks like this

```
library(proto)
math=proto()
math$add <- function (x,y) { # incorrect,no self
    return(x+y)
}
math$add(1,3)
```

The error you get is:

```
Error in res(x, ...) : unused argument (3)
Calls: <Anonymous>
Execution halted
```

Fix:

```
library(proto)
math=proto()
math$add <- function (self,x,y) { # corrected
    return(x+y)
}
math$add(1,3)
-> 4
```

So calling `math$add(1, 3)` actually is calling `math$add(math,1,3)`.

⇒ Advantage: The object itself is always available in the function body!

Let's create first a class Animal, than an individual rat animal

```
> Animal = proto()
> Animal$new <- function (self,name, # constructor
     type='generic animal',
     age=0) {
     self$name=name
     self$type=type
     self$age=age
     self$.km=0 ; # hidden/private dot variable
     return(self) # return the object
 }
> Animal$run <- function (self,km=1) {
     self$.km=self$.km+km
     return(paste(self$name, ' runs ',km,'!'))
 }
> Animal$getKm <- function (self) { # standard method
     return(self$.km)
 }
```

```
> ls(Animal)
[1] "getKm" "new"   "run"
> moritz=Animal$new(name='Moritz',type='Rat',age=1)
> moritz$run(km=0.1)
[1] "Moritz  runs  0.1 !"
> moritz$run(km=0.5)
[1] "Moritz  runs  0.5 !"
> moritz$getKm()
[1] 0.6
> # introspection
> ls(moritz)
[1] "age"   "getKm" "name"  "new"   "run"   "type"
> str(moritz)
proto object
 $ getKm:function (self)
  ..- attr(*, "srcref")= 'srcref' int [1:8] 366 17 368 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcf
 $ type : chr "Rat"
 $ name : chr "Moritz"
 $ age  : num 1
 $ run  :function (self, km = 1)
  ..- attr(*, "srcref")= 'srcref' int [1:8] 362 15 365 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcf
 $ new  :function (self, name, type = "generic animal", a
  ..- attr(*, "srcref")= 'srcref' int [1:8] 353 15 361 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcf
```

Inheritance

```
> Cat = Animal$proto() # inheritance from Animal
> Cat$meow = function (self,n=2) {
    return(paste(self$name,' says ',
     paste(rep('meow',n),collapse=', '),'!',sep=''))

 }
> ls(Cat)
[1] "meow"
> susi=Cat$new('Susi',type='Cat')
> susi$meow(n=5)
[1] "Susi says meow, meow, meow, meow, meow!"
> ls(susi)
[1] "age"   "meow" "name" "type"
> susi$run(km=1.1)
[1] "Susi  runs  1.1 !"
```

```
> susi$getKm()
[1] 1.1
> class(susi)
[1] "proto"       "environment"
> susi$ls()
[1] "age"  "meow" "name" "type"
> str(Cat)
proto object
 $ meow:function (self, n = 2)
  ..- attr(*, "srcref")= 'srcref' int [1:8] 383 12 387 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcf
 $ type: chr "Cat"
 $ name: chr "Susi"
 $ age : num 0
 parent: proto object
> ls(susi$.super)
[1] "age"   "getKm" "name"  "new"   "run"   "type"
```

Code conventions

- Classes: uppercase letters `FastaUtil = proto()`
- Objects: lowercase letters `fu = FastaUtil$proto()`
- public methods: lowercase letters `FastaUtil$read.fasta`
- private methods (used only within the class): starting with dot, _ or uc letters: `FastaUtil$.seqcheck`
- Files: each class in its own file with the name of the class as basename `FastaUtil.R`
- separate folder for applications based on your classes
- `lib` for classes, `app` or `bin` for applications
- conventions are optional but very helpful for organizing your work und mandatory for larger projects especially if you cooperate with other persons on the same project

Summary proto

- prototype based programming
- very flexible, easy dynamic extension of objects instead of using mostly static class templates like in R6 (R), Java or C++
- classes in proto are just objects as well
- the first automatic self argument allows access to the object within the function
- don't forget `self`
- Geany shows you as well the functions in the document outline (not the case with S4, R6)
- you inherit from exisiting objects using the `proto` method of them

2.4 RDS files saveRDS and readRDS

- Advantage of proto - you can save easily your objects with all properties and methods to the harddisk!
- Command to save:
  `saveRDS(object,file=filename)`
- Command to read back:
  `object=readRDS(filename)`

```
> susi$run(1.2)
[1] "Susi  runs  1.2 !"
> susi$run(0.5)
[1] "Susi  runs  0.5 !"
> susi$getKm()
[1] 2.8
> saveRDS(susi,file='susi.RDS')
```

Object reloading - How cool is that!!

```
#!/usr/bin/env Rscript
library(proto)
susi2=readRDS('susi.RDS')
ls(susi2)
ls(susi2$.super)
susi2$getKm()
susi2$run(0.1)
susi2$getKm()
susi2$meow(n=1)
⇒ [1] "age"   "meow" "name"  "type"
⇒ [1] "age"    "getKm" "name"   "new"    "run"    "type"
⇒ [1] 2.8
⇒ [1] "Susi  runs  0.1 !"
⇒ [1] 2.9
⇒ [1] "Susi says meow!"
```

readrds.R

RDS files

- writeRDS/readRDS: Functions to write a single R object to a file, and to restore it
- multiple object can be combined into a list then, the list will be stored
- complete objuects with data and analysis functions can be stored to the harddisk and used by anyone at a later time
- imagine, you could send your data and your analysis function even as E-Mail attachment to other researchers
- `proto` supports this type of analysis, the other OO systems not (maybe R6)

2.5 Code documention

- general comments (after #):
  - for the developer to explain in the code why certain things are done that way
  - TODO's what is still needs to be done, etc
- **documenting functions and showing use cases (#')**
  - to use your own functions in other use cases, you don't like to look always in your souce code
  - have your own set of help pages which extract the important things for your own written code

$\Rightarrow$ Code documentation - major key for a good programmer!
$\Rightarrow$ General comments are much less important!

RD documentation

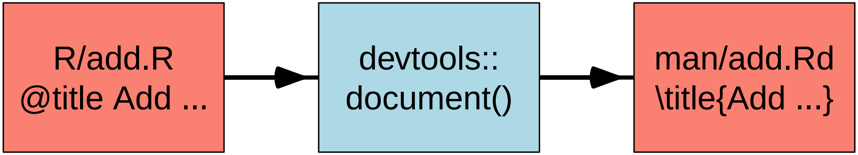
⇒ RD file format recently the standard
⇒ latex like syntax written in separate files,
`R/load.R` ⇒ `man/load.Rd`

Example:

```
% File src/library/base/man/load.Rd
\name{load}
\alias{load}
\title{Reload Saved Datasets}
\description{
  Reload the datasets written to a file with the
      function
  \code{save}.
}
...
```

Roxygen2 documentation

- newer approach
- documentation is directly added to source code
- lines with code documentation are starting with `#'`
- separate R commands are used to create then the Rd files out of the R files automatically
- `library(devtools)` function `devtools::document()` above the package directory



Roxygen2 example

```
#' @title Illustration of crayon colors
#'
#' @description Creates a plot of the crayon colors in
#'       \code{\link{brocolors}}
#'
#' @param method2order method to order colors
#'     (\code{"hsv"} or \code{"cluster"})
#' @param cex character expansion for the text
#' @param mar margin parameters; vector of length 4
#'     (see \code{\link[graphics]{par}})
#'
#' @return None
#'
#' @examples
#' plot_crayons()
#'
#' @export
```

```
plot_crayons <-
function(method2order=c("hsv", "cluster"), cex=0.6, mar=r
{ ... }
```

Practical illustration

```
#!/usr/bin/env Rscript
if (!require('devtools')) {
    install.packages('devtools')
}
usage = function () {
    cat('roxy.R converting R files comments to Rd markup
    cat('Author: D. Groth, University of Potsdam, 2020\n'
    cat('Usage: roxy.R directory\n')
}
indir=commandArgs(trailingOnly=TRUE)
if (length(indir)==0) {
    usage()

} else {
    if (!dir.exists(indir[1]))  {
        usage()
    } else {
        devtools::document(indir[1])
    }
}
⇒ roxy.R converting R files comments to Rd markup manual
⇒ Author: D. Groth, University of Potsdam, 2020
⇒ Usage: roxy.R directory
```

roxy.R

Document in an interactive R-session

- use `setwd('parentpath')` to go into the parent directory of your package
- then call `devtools::document('pkgname')`

```
> setwd("rlibs-dev/")
> list.files()
 [1] "asg"      "asgmd"     "asgui"
 [4] "cat"      "dgtools"
> library(devtools)
> devtools::document('cat')
Updating cat documentation
Writing NAMESPACE
Loading cat
Welcome Package1
Writing NAMESPACE
```

Create packages (pwr2020)

```
#!/usr/bin/R --vanilla --slave -f
if (!dir.exists('pwr2020')) {
    add <- function(x, y) {
       x + y
    }
    tdata=data.frame(a=1:10,b=rnorm(10),c=LETTERS[1:10])
    package.skeleton('pwr2020')
}
options(width=50)
list.files('pwr2020',recursive=TRUE)
file.show('pwr2020/R/add.R')
⇒  [1] "data/tdata.rda"
⇒  [2] "DESCRIPTION"
⇒  [3] "man/add.Rd"
⇒  [4] "man/pwr2020-package.Rd"
⇒  [5] "man/tdata.Rd"
⇒  [6] "NAMESPACE"
⇒  [7] "R/add.R"
⇒  [8] "R/add2.R"
⇒  [9] "R/pwr2020-internal.R"
⇒ [10] "Read-and-delete-me"
⇒ add <-
⇒ function (x, y)
⇒ {
⇒     x + y
⇒ }
⇒
```

pkgskel.R

⇒ more about packages on Day 4

Create add2.R with Roxygen2 documentation

```
> if (file.exists('pwr2020/man/add2.Rd')) {
     file.remove('pwr2020/man/add2.Rd')
 }
[1] TRUE
> cat("
 #' Add together two numbers
 #'
 #' @param x A number
 #' @param y A number
 #' @return The sum of \\code{x} and \\code{y}
 #' @examples
 #' add2(1, 1)
 #' add2(10, 1)
 add2 <- function(x, y) {
   x + y
 }
 ",file='pwr2020/R/add2.R')
```

Creating add2.Rd file

```
#!/bin/sh
ls pwr2020/man
./roxy.R pwr2020
ls pwr2020/man
head pwr2020/man/add2.Rd
⇒ add.Rd
⇒ pwr2020-package.Rd
⇒ tdata.Rd
⇒ Writing add2.Rd
⇒ add2.Rd
⇒ add.Rd
⇒ pwr2020-package.Rd
⇒ tdata.Rd
⇒ % Generated by roxygen2: do not edit by hand
⇒ % Please edit documentation in R/add2.R
⇒ \name{add2}
⇒ \alias{add2}
⇒ \title{Add together two numbers}
⇒ \usage{
⇒ add2(x, y)
⇒ }
⇒ \arguments{
⇒ \item{x}{A number}
```

roxy.sh

Code documentation summary

- for creating help pages of your code
- do this even if you are the only user of your code
- embedding documentation into source code is best
- commenting using Roxygen2 tags directly above the code
- if you change arguments to your function, just update the docs before
- function to create documentation: `devtools::document`
- more about packages tomorrow

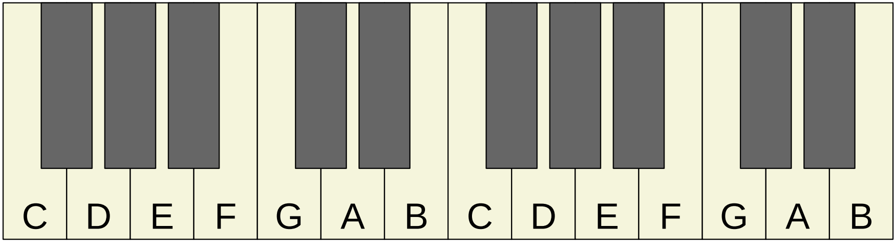
2.6 R graphics system

- graphics - core
- lattice - core
- ggplot2 - additional external popular library

⇒ we will only cover standard graphics here
⇒ it is easy to use for beginners
⇒ it can be used comfortably as well for the experienced programmer even it is sometimes less flexible
⇒ with lattice and ggplot, graphics can be variables
⇒ with base graphics we just paint on the graphics device

Example 1

```
> par(mai=rep(0.1,4))
> # empty painting surface
> plot(1,type='n',xlab='',ylab='',
     xlim=c(1,15),ylim=c(0,3),axes=FALSE)
> for (i in 1:14) {
     rect(i,0.25,i+1,2.75,col='beige')
     text(i+0.5,0.5,
       label=rep(c('C','D','E','F','G','A','B'),2)[i],
       cex=2)
 }
> for (i in c(1,2,3,5,6,8,9,10,12,13)) {
     rect(i+0.6,1,i+1.4,2.75,col='grey40')
 }
```


C
D
E
F
G
A
B
C
D
E
F
G
A
B

Example 2

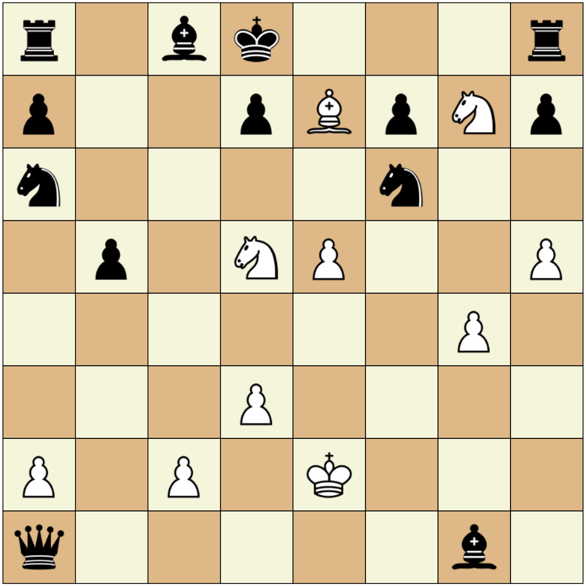
```
> # empty painting surface
> par(mai=rep(0.1,4))
> plot(1,type='n',xlab='',ylab='',
     xlim=c(0,2),ylim=c(0,3),axes=FALSE)
> labels=rev(c('Thu','Fri','Wknd','Mon','Tue','Wed'))
> topics=c('OOP','Functions','Intro','Relaxing ...',
     'GUI','Packages')
> z=1
> for (x in 0:1) {
     for (y in 0:2) {
         rect(x,y,x+1,y+1,lwd=3)
         rect(x,y,x+0.2,y+0.2,lwd=1)
         for (y2 in 1:5) {
             lines(c(x,x+1),c((y+y2*0.2),c(y+y2*0.2)))
         }
         text(x+0.1,y+0.1,label=labels[z])
```

```
        text(x+0.5,y+0.5,label=topics[z],cex=2,
            family="serif")
        z=z+1
    }
}
```

Intro	Packages
Mon	Thu
Functions	GUI
Tue	Fri
OOP	Relaxing ...
Wed	Wknd

Example 3

```
> library(png)
> par(mai=rep(0.1,4))
> plot(1,type='n',xlab='',ylab='',
     xlim=c(0,8),ylim=c(0,8),axes=FALSE)
> imgDir='/home/groth/workspace/delfgroth/mytcl/dchess/'
> fen='r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1'
> for (x in 1:8) {
     fen=gsub(x, paste(rep(' ',x),collapse=''),fen)
 }
> fen=gsub('/','',fen)
> cols=rep(c(rep(c('burlywood','beige'),4),
     rep(c('beige','burlywood'),4)),4)
> k=1
> imgdir='/home/groth/workspace/delfgroth/mytcl/dchess/wi
> for (i in 0:7) {
   for (j in 0:7) {
        rect(i,j,i+1,j+1,col=cols[k])
        k=k+1
   }
 }
> for (i in 0:7) {
   for (j in 0:7) {
    idx=64-8*(i+1)+j+1
    piece=substr(fen,idx,idx)
    if (piece != ' ') {
         if (grepl('[A-Z]',piece)) {
             col='w'  } else { col='b' }
         imgfile=paste(imgdir,col,toupper(piece),
             '.png',sep='')
         pic <- readPNG(imgfile)
         rasterImage(pic, xleft = j+0.1,
             ybottom = i+0.1,
             xright=j+0.9,ytop=i+0.9)
     }
   }
 }
```



R plotting - devices

- screen
  - x11() - Unix
  - windows() - Windows
  - quartz() - Mac OSX
- files
  - pdf (vector)
  - svg (vector)
  - png (bitmap)
  - jpeg (bitmap)
  - ...
- close device - dev.off()

PDF device

- the preferential device for publications
- multipage enabled
- options
  - width (inches)
  - height (inches)

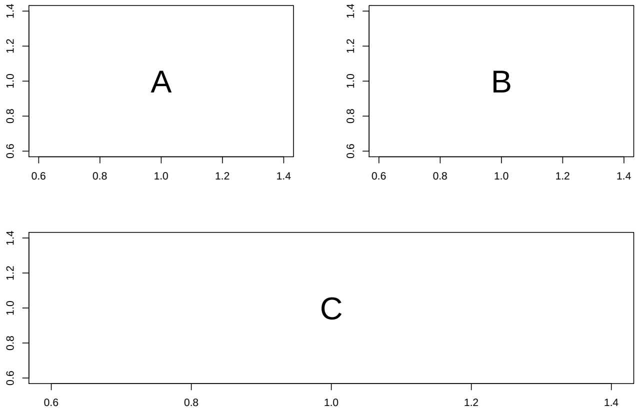
```
> args(pdf)
function (file = if (onefile) "Rplots.pdf" else "Rpl
    width, height, onefile, family, title, fonts, ve
    encoding, bg, fg, pointsize, pagecentre, colormo
    useKerning, fillOddEven, compress)
```

Multifigure plots

- par(mfrow=c(2,3)) - two rows, three columns with rowwise filling
- par(mfcol=c(2,3)) - two rows, three columns with colwwise filling
- layout() - different arrangements possible

```
> pdf('test.pdf',width=9,height=6)
> par(mai=rep(0.5,4))
> mt=matrix(c(1,2,3,3),nrow=2,byrow=TRUE)
> mt
     [,1] [,2]
[1,]    1    2
[2,]    3    3
> layout(mt)
> plot(1,pch='A',cex=3)
> plot(1,pch='B',cex=3)
> plot(1,pch='C',cex=3)
> dev.off()
```

```
null device
          1
```



Summary graphics

- beside standard statistics plots we can build our own graphics
- flexible approaches are starting with en empty surface
- low level functions are used thereafter to plot different graphical elements like text, rectangles, images
- flexible multifigure graphics can be build using par settings or the layout function
- the prefered output for publication is PDF, sometimes with many, many data points bitmap graphics might be a better choice

Lecture summary

OOP:

- S3 (1986) - method(object)
- proto (2005) - object$method()
- proto objects can be saved to the file system
- follow your code conventions

Code documentation:

- document your functions, methods
- Roxygen2 for embedding docu beside the code

Plotting:

- flexible base graphics package
- multifigure plots
- prefered output pdf

2.7 Exercise OOP
Task 0 - prepare workspace and editor

- Start your Geany text editor.
- If not done yet download the FASTA files for the two types of Corona virus from the Moodle course site.
- Place the files under a data sub directory of your R files.
- More features of the Geany text editor:
  - Activate the Plugin File-Browser using the Menu Tools->Plugin manager.
  - On the left you should now have a file browser tab.
  - Templates are starting files for your programming.
  - Create an R template file app.R within the folder ~ `/.config/geany/templates/files/` (Unix) or on something like `C:\Users\UserName\Roaming\geany` (Windows).
  - To find out the correct path look at the beginning of the menu

point output of 'Help->Debug Messages'.

- Please note that to save the new template you might activate the show hidden files option at the bottom of the file open dialog (Show more options).
- For details about templates look later here at the documentation: `https://www.geany.org/manual/current/index.html#templates`.
- The app.R template should have a shebang, the usage function, the main function.
- After saving the file in the template directory reload the configuration (Tools->Reload Configuration) and the new file appears again.

Task 1 - FastaUtil.R - outline and documentation

- Let us write a S3 class FastaUtil.
- At first we create a function read.fasta in the file FastaUtil.R.
- You can copy the read file functionality from yesterdays exercise into this new file.
- The function should have the following arguments:
  - infile: input filename
  - n: number of sequences to read from the file, if n == -1 all sequences should be read
  - id: possible id to be read from the file, then only teh sequence for this id is returned
- The function should return a list object of class FastaUtil.
- Test the code with both corona FASTA files available in Moodle.
- See below for an outline:

```r
#' @title read sequences from a fasta file
#'
#' @description
#'
#' @param infile input filename  of a standard
#'        fasta file
#' ...
#' @export
read.fasta <- function (infile,n=-1,id='') {
    fasta=list()
    # loop over file
    # loop over the file and create list entries for
    # each ID entries consiting of the id as the key
    # and a nested key with description and sequences
    # ex; l[[key]]=list(description="Hello description",
    #    sequence="MATCL...",length=NN)
    class(fasta)="FastaUtil"
    return(fasta)
}
```

Task 2 - FastaUtil.R - summary.FastaUtil

- R summary functions usually display information about the sequences.
- It should display how many sequences are stored in this object and then give a table about the length of each sequence.
- We loop over each key names(fasta) and we extract the relevant information to the terminal.
- We create a data frame within this function and return it.
- See below for an outline

```
#' @title summarize a FastaUtil sequence object
#' ...
summary.FastaUtil <- function (x) {
    names=names(x)
    lengths=c()
    for (nm in names) {
        # calculate lengths as nlength
        lengths=c(lengths,nlength)
    }
    return(data.frame(id=names,length=lengths))
}
```

Homework:

- Have a short look at this R manuall `https://cran.r-project.org/doc/manuals/R-lang.pdf` chapter 4 and 5
- Have a look at the proto Vignette: `https://cran.r-project.org/web/packages/proto/vignettes/proto.pdf`
- Have a look at the biophysical properties of amino acids `https://www.sigmaaldrich.com/life-science/metabolomics/learning-center/amino-acid-reference-chart.html`

References

Wikipedia — The free Encyclopedia. Wikipedia, Object-oriented programming, 2020. URL `https://en.wikipedia.org/wiki/Object-oriented_programming`. [Online; accessed 27-September-2020].