Be in shell

Mostly complete slides

Pavel Fibich

pavel.fibich@prf.jcu.cz
dep. Botany - Na Zlaté stoce 1

02-2016

Přírodovědecká
fakulta
Faculty
of Science

Course motivation

Why to have such course in the bioinformatics study program?

- bioinformaticians are working with big and complex data, e.g. one pair read of tick genome in fasta sanger 1.9 format has 250 Gb
- there is need to easy manipulate, filter, sort, ... process big data

Linux shell is

- powerful, there are many tools for files, text and other manipulations
- reusable and fast, writting script avoids repeating Excel clicking
- widespread as platform, many bioinformatics toos are written for linux
- common in computational centers where you can use parallel processing of data

Be in shell allows you to *program on steroids* (fast writting of fast and powerful tools).

Course introduction

Goals of the course:

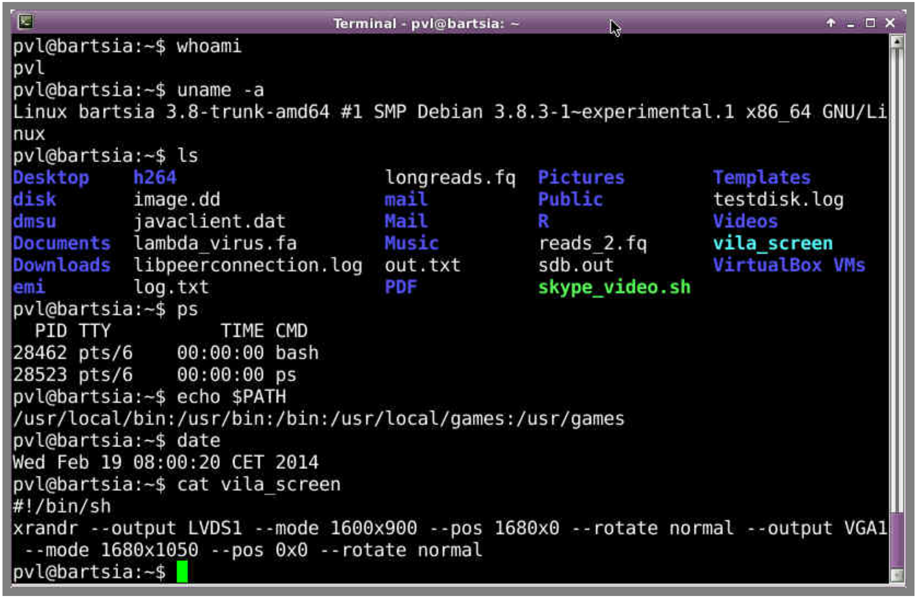
- be able to work under linux shell (i.e. bash) and use its power
- know and do not hesitate to use bash scripting, sed, AWK, GNUplot, R, python or perl
- BUT NOT to make linux administrators from you

How to make it:

- lectures every second week on Wed 13:15 – 15:30 at BB-7 room
- combination of theoretical and practical parts
- few practical homeworks influence final score
- final exam as discussion

No attendence checking!

Be in shell



```
pvl@bartsia:~$ whoami
pvl
pvl@bartsia:~$ uname -a
Linux bartsia 3.8-trunk-amd64 #1 SMP Debian 3.8.3-1~experimental.1 x86_64 GNU/Li
nux
pvl@bartsia:~$ ls
Desktop    h264                     longreads.fq  Pictures        Templates
disk       image.dd                 mail          Public          testdisk.log
dmsu       javaclient.dat           Mail          R               Videos
Documents  lambda_virus.fa          Music         reads_2.fq      vila_screen
Downloads  libpeerconnection.log    out.txt       sdb.out         VirtualBox VMs
emi        log.txt                  PDF           skype_video.sh
pvl@bartsia:~$ ps
  PID TTY          TIME CMD
28462 pts/6    00:00:00 bash
28523 pts/6    00:00:00 ps
pvl@bartsia:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
pvl@bartsia:~$ date
Wed Feb 19 08:00:20 CET 2014
pvl@bartsia:~$ cat vila_screen
#!/bin/sh
xrandr --output LVDS1 --mode 1600x900 --pos 1680x0 --rotate normal --output VGA1
 --mode 1680x1050 --pos 0x0 --rotate normal
pvl@bartsia:~$ 
```

Materials

Shell scripting as ”Be in shell”

- Course materials at ‣ http://botanika.prf.jcu.cz/fibich/teaching.html
- Mostly all books about linux have some chapter about shell scripting
- Many online tutorials, just look for it
- books
  - Blum R. (2008) Linux Command Line and Shell Scripting Bible. Wiley.
  - Burtch K.O. (2004) Linux Shell Scripting with Bash. Sams Publishing.
  - Sobell M.G. (2009) Practical Guide to Linux Commands, Editors, and Shell Programming. Prentice Hall.

Program – working version

- 17.2. Introduction, work space, shells.
  Files, series of commands, output.
- 2.3. Variables, arithmetic expansion.
  Script basics, compound commands, if-else
- 16.3. *Training on exercises*
  Compound commands, loops, signals, **Homework 1**
- 30.3. Text files manipulations, regex, sed
  AWK, **Homework 2**
- 13.4. *Big exercise*
  GNUplot
- 27.4. R, Python, Perl,
  Big exercise 2, **Homework 3**
- 11.5. *Questions*
  Final exam - the first and recommended try.

Keywords

Unix	is operating system, has many variants
POSIX	family of standards
Linux	is Unix-like and POSIX-compliant operating system (Debian, Ubuntu, ...)
shell	is user interface for accessing services of operating system (CLI or GUI)
sh	is Bourne shell default in Unix systems
bash	is Unix shell command line processor, default in Linux and MAC OS X, free replacement of sh
csh	is C shell, syntax closer to C language
linux console	(tty) is single user way how get/send infromations/commands from/to linux kernel
terminal	is program that runs shell
xterm	is terminal emulator for X window system

For the beginning, shell $\sim$ bash $\sim$ terminal assume as the same.

Bash in general

▸ www.gnu.org/software/bash

bash means Bourne-again shell

- command language interpreter written by Brian Fox, first released 1989
- widely distributed, even ported for MS Windows and cygwin
- typically runs in text window
- sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh)
- supports filename wildcarding, piping, command substitution, variables and control structures for condition-testing and iteration
- keywords, syntax and other basic features of the language were all copied from sh
- GNU GPL version 3

Workspace - playground

```
$ pwd
?
```

Ideally, to have own linux machine, or virtual one, with bash.

If you affraid of linux, use application VirtualBox (www.virtualbox.org), create virtual machine and install of Debian or Ubuntu (or use some live linux, eg. Ubuntu). Mostly you need to get iso image with linux installation and attach it to the virtual machine to boot from it.

Where to start?

Log in linux OS and run terminal. You will get to command line, it often ends with $. After $, you can write commands, e.g. `whoami`

```
pvl@bartsia:~$ whoami  # what is my user name
pvl
pvl@bartsia:~$
```

- part before $ is mostly omitted (often USERNAME@COMPUTERNAME:ACTUALFOLDER, where ∼ is alias for HOME directory)
- after $, there are commands written
- line without $ is the output of previous command, e.g. value of variable, content of file, . . .
- after #, there are comments written, they are not interpreted (run) by shell

Few starting commands

```
$ date # actual date
Mon Jan 27 14:17:28 CET 2014
$ ls # list of files in the current folder
myfile
$ cat myfile # print content of file myfile
Nice day!
Nice shell!
$ grep day myfile # print lines having word day in file myfile
Nice day!
$ env # print list of all actual variables
...
$ echo $SHELL # print content of variable SHELL
/bin/bash
$ ps # snapshot of processes of current shell
  PID TTY          TIME CMD
21356 pts/0    00:00:00 bash
21399 pts/0    00:00:00 ps
```

Manuals - where to find help

We should start with `man` command for manual

```
$ man man  # manual pages of command man
$ man grep  # manual pages of command grep
$ apropos grep  # search for keyword grep
$ info grep  # different informations about grep
$ grep --help  # the most of commands have --help option
```

Mans have their options too (e.g. manual sections of man command, see `man man`)

```
$ man passwd  # passwd command changes password
$ man 5 passwd  # man about password file
```

Task

Always look at the man pages of the commands you are using (e.g. `man grep`), until you will get used to use them. `man bash` is quite compressive *long winter nights* article about what we will use.

Linux file system - paths

- Paths of folders and files are in linux separated by / (forward slash).
- Absolute path is full path from root (like in Windows from C:), e.g.
  `/etc/profile`
  `/home/pvl/bash/myFolder`
- Relative path starts from the actual folder (denoted by ., you can get it by `pwd` command), e.g.

```
./myFolder/myfile  # . is actual folder
../pvl/myfile  # .. is folder above the actual folder
```

- If no path with / or . is specified, than actual folder is assumed
- We will often work in some folder in home directory (denoted by ∼) of the user

```
pvl@bartsia:~/Documents/prf.jcu/bash$ pwd  # print work dir
/home/pvl/Documents/prf.jcu/bash
```

Files and Directories

Directories are rooted (start) in the / (slash)

```
$ pwd # print actual directory
/home/pvl/Documents/prf.jcu/bash/c1
$ ls -a # list of all files in the current directory
.  ..  myfile
$ cd .. # go to the parent directory
$ pwd
/home/pvl/Documents/prf.jcu/bash
$ cd c1 # go to c1 directory
$ pwd # print actual directory
/home/pvl/Documents/prf.jcu/bash/c1
$ ls -a / # list of files in the file system root
...
```

If path is not specified, files/directories in the current folder are assumed!

Task

Look what is in the file system root folders (`ls -a /`).

Directories and files - manipulation and access

```
$ mkdir mydir  # creates directory
$ rmdir mydir  # deletes empty directory
$ rm -r mydir  # deletes directory recursively (everything inside)
$ cp file newfile  # copy file in the same dir
$ cp file /home/pvl/somewhere  # copy file into absolute path
$ mv file ./folder/newfile  # rename/move file or directory to new path
$ rm file  # remove file
```

Access rights - filetype, 3x(**r**ead, **w**rite, e**x**ecute) for user,group,others

```
$ ls -la
total 12
drwxr-xr-x 2 pvl pvl 4096 Jan 29 09:32 .
drwxr-xr-x 3 pvl pvl 4096 Jan 29 09:32 ..
-rw-r--r-- 1 pvl pvl   22 Jan 22 21:28 myfile
```

Task

Look at `/dev` for more filetypes.

Directories and files

```
$ file myfile # file type info
myfile: ASCII text, with very long lines
$ chmod a+rx myfile # modifying access rights, all can rx (read execute)
$ chmod go-rwx myfile # group cannot rwx
$ wc -l myfile # print number of lines
104 myfile
$ ln -s myfile linkfile # create pointer to file, link
$ ls -la # print all files with details
drwxr-xr-x 2 pvl pvl 4096 Jan 29 10:45 .
drwxr-xr-x 3 pvl pvl 4096 Jan 29 10:45 ..
lrwxrwxrwx 1 pvl pvl    6 Jan 29 10:45 linkfile -> myfile
-rw-r--r-- 1 pvl pvl 3828 Jan 29 10:01 myfile
$ ls -ltr # list according date, reversally
...
```

Symbolic links are very useful, you can link also folders and do not need extra copies.

When shell starts up

Default shell of the user is mostly defined in the file `/etc/passwd`

```
$ grep pvl /etc/passwd # print lines with word pvl in file passwd
pvl:x:1000:1000:Pavel Fibich,,,:/home/pvl:/bin/bash
```

When shell starts, it runs startup files (distribution specific) to initialize itself.

- `/etc/profile` – system specific settings
- `/etc/bash.bashrc` – shell specific settings
- `~/.profile` – user specific settings
- `~/.bashrc` and `.bash_*` files) – user shell specific settings

If you want to make some user specific changes, do it in `~/.bash_*` or `~/.bashrc` files!

Bash settings * - tunning of shell

Many bash settings is done by `set` or `shopt` commands (running the commands print the list of settings or options). Some useful settings

- do not allow overwriting files

```
$ set -o noclobber # not allow
$ set +o noclobber # allow
```

- `emacs` (default) or `vi` options define differences of command prompt behaviour

```
$ set +o vi
```

- `xtrace verbose` set debugging or verbose mode

```
$ set -o xtrace
$ set -o verbose # print command before execution
```

- `dotglob` filenames starting with . or .. do not match using wildcards

```
$ set -o dotglob
```

Bash shortcuts - working faster

- UP and DOWN keys are used for the listing in the past commands
- two TAB keys find matching file name (e.g. write `whoa` and then press TAB twice)
- CTRL + R searches in the history of command line (e.g. `~/.bash_history`), press it and write `w`, then you can press it again for later appearance, or press enter and UP/DOWN
- !CMD runs last matching command CMD from history (!! runs last command), e.g.

```
$ !w
whoami
pvl
$ history # print history of commands
...
```

- selected text in the shell, can be pasted by middle button of the mouse

Exercise

Exercise

- decide where to work: using linux machine or virtual one
- run shell and try presented commands
- prepare environment where you will work under linux (eg. get used with linux environment - set wifi connection, create directory for the course, make shortcut for running of terminal)
- look briefly at man pages of already mentioned commands
- go through the actual and one future lesson

envir and commands

environment and commands

Users and system info

```
$ id # print user and group ids (see /etc/group)
uid=1000(pvl) gid=1000(pvl) groups=1000(pvl),24(cdrom)
   ,25(floppy),29(audio),30(dip),44(video),46(plugdev)
   ,105(scanner),110(bluetooth),111(netdev),123(scard)
$ w # show who is logged on and what they are doing
 09:46:13 up 3 days, 15:21, 9 users, load average: 0.23,
     0.23, 0.23
USER TTY    FROM LOGIN@ IDLE   JCPU   PCPU WHAT
pvl  pts/0 :0.0 Tue19  12:35m 0.05s  0.05s bash
pvl  pts/1 :0.0 Tue20  12:47m 0.34s  0.02s vim disk26
...
$ uname -a # show machine characteristics
Linux bartsia 3.8-trunk-amd64 SMP Debian 3.8.3-1~
   experimental.1 x86\_64 GNU/Linux
```

Task

Look at `/proc/` files to get more information (e.g. `cat /proc/cpuinfo`)

Printing and editing

```
$ head myfile  # print 10 first lines from file
...
$ tail -n 5 myfile  # print 5 last lines from file
...
$ less myfile  # for reading the file
$ nano myfile  # simple text editor (or use pico)
$ vim myfile  # editor preferred by linux guru :-)
```

Shell often have many editors:

nano ▸ www.nano-editor.org - CTRL+x to exit, CTRL+o to write actual state

Emacs ▸ www.gnu.org/emacs/

vi, vim ▸ www.vim.org/

- two states : press **i** (insert) or ESC (command)
- **:wq** to save and exit
- **:!q** to not save and exit

Task

Get familiar with some text editor!

Text manipulation and printing

Files manipulation and info

```
$ sort myfile # sort file
...
$ uniq myfile # unique lines
...
$ diff myfile newfile # differences of files
...
```

Text printing

```
$ echo "hello" # print text
$ printf "%d\n" 5 # C like print as digit (integer)
5
$ printf "%f\n" 5 # print as float
5.000000
$ printf "There are %d dogs and %d cats.\n" 3 2
There are 3 dogs and 2 cats.
$
```

Exercise

Exercise

- run following command, it will download file `ideff.csv`

```
$ wget http://botanika.prf.jcu.cz/fibich/ideff.csv
```

  or if you do not have wget installed, try `curl`

```
$ curl http://botanika.prf.jcu.cz/fibich/ideff.csv >
    ideff.csv
```

- try to list (print) the file, print only the first 3 lines
- count number of lines of the file
- try to edit the file
- create new directory and copy the file in
- delete the directory
- look briefly at man pages of used commands
- try and read all Tasks

Sequence of commands - background

We can write more commands in one line separated by `;`

```
$ pwd; wc -l myfile; cat myfile
```

We can easily run command in background (append &), or during running of command press CTRL+z to suspend and than write `bg` to move to background or `fg` to move it to foreground

```
$ ls -la & # run ls in background
[1] 18472
...
[1]+  Done                          ls --color=auto
$ sleep 20 & # run sleep for 20 s in background
[1] 18511
$ sleep 30 &
[2] 18512
$ jobs # print jobs in background
[1]-  Running                       sleep 20 \&
[2]+  Running                       sleep 30 \&
$ kill 18512 # force end the job with given PID
```

What to do with the output? - pipe

One of the first options, when we want to combine commands, is the pipe (`|`). It takes output of one command as input for the next command (*redirection between commands*)

```
$ grep cpu /proc/cpuinfo | wc -l
16
$ ls -la | head -n 2  # print first two lines from ls command
total 12
drwxr-xr-x 2 pvl pvl 4096 Jan 29 10:28 .
# who print who is logged, tee send output to std. out and to file
$ who | tee log.txt | grep "2014-02-03"
pvl      pts/4        2014-02-03 20:33 (:0.0)
$ wc -l log.txt  # number of lines in the log.txt
4 log.txt
```

Task

Run commands separately and check their inputs and outputs.

What to do with the output? - redirection

Output redirection to (>) /from (<) file

```
$ ls -la > myfile # redirect std. output into file
$ ls -la >> myfile # append std. output at the end of file
$ cat < myfile # redirects file to the command
```

Commands have also error output

```
$ ls -la /nonsense > myout # non-existing directory, nothing redirected
$ ls -la /nonsense 2> myout # good redirecton of error output
$ ls -l * 1> stdout 2> stderr # separating outputs
```

To combine std. and error output use &>.
Commands also set variable $? (return value of the last command)

```
$ ls -la / &>/dev/null; echo $?
0
$ ls -la /nonsense &>/dev/null; echo $?
2
$
```

Files exercise

Run following commands to generate input for exercise

```
$ mkdir fexer; cd fexer
$ touch {a..e}file; touch {1..26}new
```

Exercise

- count files in the folder
- print reversaly sorted list of files (one per line)
- from the list, print names of the last 4 files and then the first 3 files
- count all files in the folder and files having ”1” in name (e.g. by `grep`)
- delete folder `fexer`
- where it is possible try to use both, redirection and pipe
- go through the actual and future lesson

variables

variables

What to do with the output? - variables

Output of the command can be stored as variable too. They are faster than files, because they are stored in memory.

```
$ Y=5  # set variable Y to 5
$ echo $Y  # print variable Y, must use $
5
$ X="ls -l | wc -l; $Y"; echo $X  # double quotes
ls -l | wc -l; 5
$ X='ls -l | wc -l; $Y'; echo $X  # quotes (apostrophes)
ls -l | wc -l; $Y
$ X=`ls -l | wc -l; $Y`; echo $X  # backquotes (key left from 1)
bash: 5: command not found
4
$ unset Y; echo $Y  # unset variable Y

$ set | less  # list in set of variables
```

Double and single quotes define text (variable substitution is possible only for the first). Back quotes or $() are used for running command inside them.

declare variables

In Bash, we can also use `declare`

```
$ declare COST
$ COST=5
$ declare -x COST=5  # equiv for export COST=5, see slide about visibility
```

Variables are stored as strings, we can specifie the type of variable

```
$ declare -i COST=5  # integer type
$ declare -rx COST=5  # read only exported variable
```

Substring of variable

```
$ MYV=tmp/filesID345561for.csv
$ echo ${MYV:9:8}  # name, offset, length
ID345561
```

Task

Small difference can do a lot of troubles. Check difference `X=‘wc -l file‘` and `X=‘cat file | wc -l‘`.

Variables in practice

Exercise

- create variable MN and use `whoami` to set it
- create variable MR and set as the number of files in /
- create var MRI like MR, but only from files having i in the name
- create var AVAR and set it to ”Hello dear”
- create var BVAR and set it to value of AVAR + MR, and print it
- check differences

```
$ printf "\%s\n" $BVAR; printf "\%s\n" "$BVAR"
$ echo $BVAR; echo "$BVAR"; echo '$BVAR'
```

- if you store command in variable, how to run it? e.g. `cat /proc/cpuinfo`
- for `A=pwd; B='$A'` try to print B, and using B force substitution of its value by `eval` (run command stored in A)

Array variables

Bash supports one dimensional array variable. Subscripts are integers from zero.

```
$ NAMES=(max helen sam zach)
$ echo ${NAMES[0]} # print first element
max
$ echo ${NAMES[*]} # print all elements as one value (for array use @)
max helen sam zach
$ declare -a  A='([0]="xy" [1]=yv [2]=vz)'
$ farm=(dog cat sheep cow)
$ echo ${farm[@]}
dog cat sheep cow
$ X=(*) # create array from files in the folder
```

You can check the length

```
$ echo ${#A[*]} # length of array
3
$ echo ${#NAMES[1]} # length of element
5
```

Variables defaults and messages

We can check for default values. `a-c` are not set.

Offering default

```
$ echo ${a:-myval} # if not set use default
myval
$ echo $a; a=new; echo ${a:-myval}

new
```

Set default

```
$ echo ${b:=$(whoami)} # if not set use default
pvl
$ echo $b; b=new; echo ${b:=$(whoami)}
pvl
new
```

Testing if variable is set

```
$ cd ${c:?Variable is not set at $(date +%Y)}
bash: c: Variable is not set at 2014
```

Variables visibility

Visibility of variables is limited

```
$ XYVAR=7; echo $XYVAR # set and print variable
7
$ env | grep XYVAR # XYVAR is not in env. vars
$ bash # run new shell
$ echo $XYVAR # print varibale

$ exit # exit shell
```

Using environmental variables (evailable through `env`)

```
$ export XYVAR=6; echo $XYVAR # add varible to ... and print
6
$ env | grep XYVAR # XYVAR is in evn. vars.
XYVAR=6
$ bash
$ echo $XYVAR
6
$
```

Wildcards

If we do not want to the specify exact name, we can use wildcards

- * - no, one or many characters
- ? - one character
- [] - range of characters, e.g. $[a-z]$, $[1-9]$, $[a-d, x-z]$, $[145]$

```
$ ls *file # list files ending with "file"
linkfile  myfile
$ ls myfil? # list files having one character after "myfil"
myfile
$ ls fi* # list files starting fi
fi1  fi1234  fi2  fi3
$ ls fi[1-2]
fi1  fi2
$ ls fi[1-2]*
fi1  fi1234  fi2
$ ls fi[^1-3] # caret for excluding
fia  fiA  fib
```

Wildcards in practice

Run following commands to generate input for exercise

```
$ mkdir wexer; cd wexer
$ touch max{a..d}; touch max{1..16}
```

Brace expansions {a..d},{one,two,three}, {1..100} generate lists.

Exercise

- create variable mymax, set it to max* and print the exact value
- create array variable and store there all files having at least one digit in the name, print the array and its lenght
- save list of file names having exactly one digit in the file out.txt and variable wout, then print file out.txt and variable wout
- delete files having two digits in the name and count them
- generate 213 files, in form `my1file, my2file, ... my213file` and store their names in file `maxa`
- print content of all files having `a` in the name

Wildcards in practice solutions

Exercise

- create variable mymax, set it to max* and print the exact value

```
mymax=max*; echo "$mymax"
```

- create array variable and store there all files having at ..

```
a=(max[0-9]*); echo ${a[*]}; echo ${#a[*]}
```

- save list of file names having exactly one digit in the

```
wout=$(ls max[0-9] | tee out.txt);
echo $wout;cat out.txt
```

- delete files having two digits in the name

```
rm max[0-9][0-9]
```

- generate 213 files, in form `my1file, my2file, ...`

```
touch my{1..213}file
```

String matching

Bash provides string pattern matching operators that can manipulate pathnames and other strings.

- minimal matching prefix #

```
$ MV="34 wild dog, calm sheep, bad cat"
$ echo ${MV#*,}
calm sheep, bad cat
```

- maximal matching prefix ##

```
$ echo ${MV##*,}
bad cat
```

- % and %% is used for minimal and maximal matching suffixes, respectively

```
$ echo ${MV%%,*}
34 wild dog
```

Arithmetic expression

`let` command performs math calculations and expects string

```
$ let "SUM=5+5"; printf "%d" $SUM
10
$ let "SUM=SUM+3"; echo $SUM
13
$ let "SUM++"; echo $SUM
14
$ let "RES=SUM!=14"; echo $RES
0
$ let "RES=SUM!=1"; echo $RES
1
$ let "RES=SUM<20"; echo $RES
1
```

We can also combine `let` with other commands

```
$ let X=`cat ideff.csv | wc -l`; echo $X
15
$ let X=`cat ideff.csv | wc -l`*2; echo $X
30
```

More expansions

Arithmetic expansion by $((*exp*)), without $ you get only status code

```
$ cat my60
#!/bin/bash
echo -n "How old are you? "
read age  # wait for input from user
echo "Wow, you have $((60-age)) years to sixty!"
$ ./my60
How old are you? 10
Wow, you have 50 years to sixty!
```

(()) evaluates arithmetic expr., by $ you get output the value

```
$ x=5 y=8; echo $((2*$x + 10*y))  # $ in brackets is not necessary
90
$ (( w=x*y )); echo $w  # we did not used $ because of setting w
40
$ myvar=$(( $(wc -l < /proc/cpuinfo) - 100 ))
$ echo $myvar
4
$ (( x*y ))  # how you get output from the expr?
```

Exercise on expansions and arithmetics

Exercise

- get data by

```
$ wget http://botanika.prf.jcu.cz/fibich/ideff.csv
```

- create variable containing the last line of the file
- create variable corresponding day of week (1..7); 1 is Monday
- create a variable and set it to 3 * (times) number of lines of the file
- print first half of the file (lines 1 to N/2)
- from the last line of the file, get value of the last column (columns are separated by commas, try string matching)
- create variable for the count of all variables in the current shell (check `set`)

Exercise on expansions and arithmetics

Exercise

- check differences in values of `Z`, `S`, `T` and `R`

```
$ wget http://botanika.prf.jcu.cz/fibich/ideff.csv
$ Z=1+1
$ Z=$(1+1)
$ Z=$((1+1))
$ Z=$[1+1]
$ let S=Z+cat ideff.csv| wc -l
$ let T=Z+`cat ideff.csv | wc -l`
$ let "R=Z+`cat ideff.csv| wc -l`"
```

- how to do more complicated math? try `| bc -l` and set variable to the result of `2/3`
- write command that sets variable to the number of lines of some file minus 1 and divided by 2
- **go through all exercises**
- read next slides before the lesson

script basics

script basics

Short recapitulation

Recapitulation

- quotes and double quotes are used for texts (second one allow variable substitution)

```
$ MVAR="whoami"; echo $MVAR
whoami
```

- back quotes and `$(..)` run commands

```
$ MVAR=$(whoami); echo $MVAR
pvl
```

- `$[..]` and `$((..))` do arithmetic expansions

```
$ MVAR=$((2+2)); echo $MVAR
4
```

- `{1..100}`, `{a..g}`, ... generate lists
- `file[1-9]`, `*.pdf`, ... wildcards can be used for general patterns
- script should have `x` rights to allow running

Script basics

Our first script is the sequence of commands

```
$ cat whoson
#!/bin/bash
date
echo "Currtely logged in users"
who
$
```

#! defines interpreter for script (can be bash, sh, python, R, ...)

```
$ ls -l whoson
-rw-r--r-- 1 pvl pvl 53 Feb  4 10:06 whoson
$ ./whoson
bash: ./whoson: Permission denied
$ chmod u+x whoson # add permissions to run script
$ ./whoson
Tue Feb  4 10:06:56 CET 2014
Currtely logged in users
...
```

Script basics

How to run script (or binary) without path is defined in variable `PATH`

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
$ whoson
bash: whoson: command not found
$ export PATH=`pwd`:$PATH # prepend actual directory into PATH
$ whoson
Tue Feb  4 10:06:56 CET 2014
Currtely logged in users
...
$
```

Few commands for searching where command come from

```
$ which ls # locate command
/bin/ls
$ whereis ls # locate binary, source and manuals for command
ls: /bin/ls /usr/share/man/man1/ls.1.gz
$ locate ldd # find files by name
...
```

Script running

Several ways how to run script

```
$ ./script.sh
$ bash script.sh
$ . script.sh # copy script in the current env.
$ source script.sh # . is abbrevation for source
```

Debugging by argument to `bash`

- `-n` *no execution*: checks syntax errors without execution
- `-x` *debuging*: turn debugging mode (remember `set -o xtrace`)
- (advanced) watching single variable by trap

```
#!/bin/bash
declare -i CNT=0
trap ': CNT is now $CNT' DEBUG
while [ $CNT -lt 3 ] ; do
  CNT=CNT+1
done
$ bash -x ./trap.sh
```

Exercise on scripts

Exercise

Run

```
$ mkdir escripts; cd escripts; touch {a..k}files;
```

- write command or script that prints how many times it was executed
- store the names of all files in the file one per line and all in one line
- write script that count all files in the current folder and store value in the variable `MFI` (use export to make variable visible)
- improve the script that it stores actual date and the count at the end of file `mfi.log` (both at one line), check it be re-running
- generate some new files (e.g. by `1..20news`) and improve the script to print message about the difference from the previous value of `MFI` and chek if it works
- try to debug your script (e.g. by -x option to bash)

Exercise on scripts solutions

Exercise

- store the names of all files in the file one per line and all in one line

```
ls > all; echo $(ls) > one
```

- write script that count all files in the current folder and store it as ...

```
MFI=$(ls|wc -l)  # that run $ . scriptName
```

- improve the script that it stores actual date and the count at the ...

```
MFI=$(ls|wc -l); echo "`date` $MFI" >> mfi.log
```

- generate some new files (e.g. by `1..20news`) and improve ...

```
MFIO=$MFI; MFI=$(ls|wc -l); MDI=$((MFI-MFIO));
echo "`date` $MFI $MDI" >> mfi.log
```

- try to debug your script (e.g. by -x option to bash)

```
head -n 1 script.sh
#!/bin/bash -x
```

control structures

control structures

if then else fi

To control flow of commands, we can use if *test-command* then..else.. structure

```
$ cat mytest
echo -n "Write a: "; read a
echo -n "Write b: "; read b
if test $a == $b; then # check man pages of test
  echo "Match!"
  else
  echo "Do not match!"
fi
$ ./mytest
Write a: a
Write b: a
Match!
$
```

`else` part is not necessary, and we can also add `elif` part with *test-command*

if then else

Check the number of script arguments ($#)

```
$ cat cat chkargs
if test $# -eq 0; then
  echo "Supply arguments!"
  exit 1;
else
  echo "First argument of $0 is $1"
fi
$ ./chkargs
Supply arguments!
$ ./chkargs hello
First argument of ./chkargs is hello
```

Warnings

Always check arguments!

test command

Many options to check files, their permissions, arithmetic, ...

```
$ ls -l mytest*
-rwxr--r-- 1 pvl pvl 131 Feb  4 15:05 mytest
$ test -e mytest; echo $? # test of file existence
0
$ test -e mytestNONE; echo $? # non existing file
1
$ test ! -d mytest; echo $? # test for NON directory, ! for negation
0
# combined conditions -a for AND and -o for OR
$ test -e mytest -a -d mytest; echo $? # exists and is directory
1
$ test -e mytest -o -d mytest; echo $? # exists or is directory
0
# square brackets do the test command, check man test
$ if [ -e mytest ]; then echo "exists"; fi
exists
$ if [ 5 -gt 4 ]; then echo "definitely"; fi
definitely
```

test command with wildcards

For pattern matching and strings we can use [[ ]]

```
$ COMP="Faculty of Science"
$ if [[ $COMP = F* ]]; then echo "Start by F"; fi
Start by F
$ if [[ $COMP = [ABC]* ]]; then echo "Start by A, B, or C
    "; fi
$ if [[ $COMP = +(F)*Science ]]; then echo "More special"
    ; fi
More special
$
```

+ is used for one or more characters (recall * is for zero or more). You can use also [:alpha:], [:digit:], [:lower:], ..

```
$ if [[ $COMP = [[:alpha:]]* ]]; then echo "Contains only
     alphabetics"; fi
Contains only alphabetic
```

Conditions

To combine conditions you can you also notation

```
$ if [[ 30 -gt $age && $age < 60 ]]; then ..
$ if ((30 < age && age < 60)); then ..
```

Exercise

- write `if` command that checks that actual month is March
- write `if` command that checks if the last command was successful (remember `$?`)
- write `if` command that checks if value $VAR contains value $IN
- write script that cheks if in the current folder is more files than number you give as the first argument of the script
- try to debug your script (e.g. by -x option to bash)
- read further lesson's slides

Conditions - solutions

Exercise - solutions

- write if command that checks that actual month is March

```
if [ `date +"%m"` -eq 3 ]; then echo "Nice March";
    else echo "Not March"; fi
```

- write if command that checks if the last command was successful

```
if [ $? -eq 0 ]; then echo "Fine"; else echo "Not
    fine"; fi
```

- write if command that checks if value $VAR contains value $IN

```
if [[ $VAR = *$IN*  ]]; then echo "In"; else echo "
    Not in"; fi
```

- write script that cheks if in the current folder is more files than ...

```
if test $( ls | wc -l) -gt $1; then
 echo "There is more than $1 files"
else echo "There is not more than $1 files"
fi
```

Warm up Exercise

Exercise

- write command that sets variable to the number of lines of some file minus 1 and divided by 2
- write script that count all `txt` files in the folder given as 1st argument and store the value in the variable TXT (use export to make variable visible)
- write script that cheks if in the current folder is more files than number you give as 1st argument of the script

loops

loops

loops

Several commands can repeat actions:

`while` *condition*; `do` ... `done`, `for` ... `in` ... ; `do` ... `done`

```
$ ls -l | while read FILE; do echo $FILE; done
$ for i in `seq 1 10`; do echo $i; done
$ while read -p "Company: " COMPANY; do
  if [ -f "orders_$COMPANY.txt" ]; then
    echo "There is order from this company."
  else
    echo "There are no orders from this company!"
  fi
done
```

Use CTRL+c to get out from the loop.

Exercise

Improve the script to react on `quit`, e.g. by `break` command, jump out of the loop. Add printing of the order, if there is any. Write command that will print file names in the folder without .* suffix.

loops

`for` reads sequence of values into a variable and repeats the enclosed commands one for each value

```
$ for file in *.csv; do wc -l $file; done # lines of csvs
$ for i in $( ls ); do echo "file: $i"; done
$ for FILE_PREF in order invoice purchase_order; do
  if test -f "$FILE_PREF""_vendor.txt"; then
    printf "%s\n" "There is a $FILE_PREF file for vendor"
  fi
done
```

Embedded `let`

```
$ for (( CO=5; CO<50; CO=CO+5 )); do # c style
  printf "The counter is %d\n" $CO
done
```

Exercise

Use `for` loop to create file with numbers from 1 to 999, all in one line and each on separate line.

solutions

Exercise - solutions

- Write command that will print file names in the folder without .* suffix.

```
ls | while read FILE; do echo ${FILE%%.*}; done
```

- Use `for` loop to create file with numbers from 1 to 999 ...

```
i=1; while test $i -lt 1000; do
echo $i >> 999file; i=$((i+1)); done
```

functions and aliases

For faster execution, we can prepare functions to do the serie of commands.

```
$ function whoson() { date; echo "Hi $1, currtely logged
    in users"; who; }
$ whoson Pavel  # run function with parameter
...
```

We can also create aliases for the commands

```
$ ll
bash: ll: command not found
$ alias ll='ls -l'  # create alias ll
$ ll
...
$ alias  # list of all aliases
alias ll='ls -l'
alias ls='ls --color=auto'
```

To load aliases or functions automaticaly, we must store them in the `.bash...` files (e.g. in `.bashrc`).

getopts script options

For single minus options we can use getopts, but more standart is to handle also double minus options by getopt.

```
$ cat ./opt.sh
while getopts ":sp:" o; do
  case "${o}" in
   s) s="set";;
   p) p=${OPTARG};;
   *) echo "usage: ./opt.sh -s -p arg";;
  esac
done
if [ -n "$s" ]; then echo "s = ${s}"; fi
echo "p = ${p}"
$ ./opt.sh -s -p 4
s = set
p = 4
```

First argument of getopts defines if option has argument by adding :

Combining commands

Grouping of commands by curly brackets (current shell)

```
$ MX=4
$ { sleep 5; MX=2; echo "Slept for 5s"; }
Slept for 5s
$ echo $MX
2
```

by round brackets (new shell)

```
( sleep 5; MX=3; echo "Slept for 5s" )
Slept for 5s
$ echo $MX
2
```

Conditional sequences

```
$ cat file && wc -l file # run wc if cat succeed
cat: file: No such file or directory
$ cat file || wc -l file # run wc even if cat not succeed
cat: file: No such file or directory
wc: file: No such file or directory
```

jobs and signals *

Running jobs can be watched by `top` command. If you plan to run command and then log out, it is good to run it by `nohup`

```
$ nohup myscript &
[1] 16858 # PID
```

By pressing CTRL+c we are sending signal SIGTERM. We can manually send also other signals

```
$ {sleep 60; echo "DONE";} &
[1] 16863
$ kill -SIGSTOP 16863
[1]+  Stopped                 { sleep 60; echo "DONE"; }
$ kill -SIGCONT 16863
DONE
```

We can adjust our script to be able to react to the signals (e.g. storing results when job is killed). Write `trap` at the beginning of the script

```
trap "cp outputs /storage/" TERM EXIT
```

will do `cp` if signal for cancel or exit is catched.

finding files

To find a file, you have many options

- recursive `ls` and combine with string matching (see man to `ls`)
- more powerful is `find`

```
$ find . -name "*.txt" # all files .txt from the current dir
$ find . -type d -or -name "*.txt" # ... or directories
$ find . ! -name "*.txt" # files not having suffix *.txt
```

You can check file size (`-size`), time of modification (`mtime`), ... and run some command on the found files

```
$ find . -name "*.txt" -exec cat {} \;
...
$ find . -name "*.txt" -ls # predefined command
...
```

`xargs` takes input and executes your chosen command on it, so we can avoid loops sometimes. It is often use instead of `-exec` in `find`.

```
$ echo Hello | xargs echo # simple ilustration
Hello
$ find . -name "*.bak" -type f -print | xargs /bin/rm
```

Exercise on loops and signals

Exercise

- Write a script that for the file (first script argument) print the second part of the file (i.e. for file with 10 lines, it prints the last 5).
- Create a script to print every x line of the file (e.g. use variable in loop).
- Add check that the argument is the file and have more than x (second argument) lines. If not, print corresponding error message.
- Run 100 `sleep 30` commands in background and store their PIDS in file (eg. by `ps`).
- Read the file and let all `sleep` commands to suspend (for each line get PID as substring of variable where is stored whole line, e.g. `${MYV:9:8}`).
- Write general function for it (file with PIDS and signal to be send as two arguments of function).

Exercise solutions

Exercise

- Write a script that for the file (first script argument) print ....

```
NLINES=$( cat $1 | wc -l ); NLI=$((NLINES/2))
tail -n $NLI $1
```

- Add checks that the argument is the file and have more than x (second argument) lines. If not, print corresponding error message.
- Create a script to print every x line of the file (e.g. use variable in loop).
- Run 100 `sleep 30` commands in background and store their PIDS in file (eg. by `ps`).
- Read the file and let all `sleep` commands to suspend (for each line get PID as substring of variable where is stored whole line, e.g. `${MYV:9:8}`).
- Write general function for it (file with PIDS and signal to be send

Homework 1 - ppsaver

Homework1

- Write script that will change all wired interpreters of python or perl in the folder.
- In shortcut, just change in all files 1st line (if there is such line)

```
#!/bin/bin/perl
#!/bin/bin/python
```

  change for (respectively)

```
#!/usr/bin/env perl
#!/usr/bin/env python
```

- Script will take 2 or 3 arguments: 1st is folder, where it will be recursivelly looking for files, 2nd is interpreted name (only `perl` or `python` will be specified) and if 3rd argument will be `make`, than changes will be applied, if 3rd is not specified, than just print file names with wired interpreter.

Homework 1 - ppsaver

Homework1

- Script will go through all files in the folder (1st arg) and his subfolder (and so on), check if files contain interpreter (2nd arg) and according 3rd arg will make change in the file or just print the file name.
- Example run: no 3rd arg, just list of files with wired header

```
$ ./ppsaver ttest/ perl
FILE: ttest/myfile1
FILE: ttest/ttest2/myfile1
```

- Example run: changes to be done

```
$ ./ppsaver ttest/ perl make
FILE: ttest/myfile1
MODIFIED!
FILE: ttest/ttest2/myfile2
MODIFIED!
```

Homework 1 - ppsaver

Homework1

- Check the number of arguments.

```
$ ./ppsaver ttest/
Not enough args: ./ppsaver FOLDER INTERPRETER {make}
```

- Leave all files without wired interpreter on 1st line untouched.
- Follow exact format of the input and output.
- Read the homework and test it again before sending it.
- Homework is individual, not team, work!
- Send me the solution (script name according your surname: `ppsaver.SURNAME`) by email before 18:00 28 March
- Score: 0..10

By not presenting at least some efort, you can not go for the exam. Final score from the course is mostly dependent on the scores of homeworks.

text files

text files

Text files manipulations

To get file from the web, you can easily use `wget` or `curl`

```
$ wget http://botanika.prf.jcu.cz/fibich/ideff.csv
...
```

We often work with csv or somehow delimited files. It is easy to exact columns from them.

```
# to get first two lines from 2. and 4. column (-d defines delimiter)
$ cut -d',' -f2,4 ideff.csv | head -n 2  # cut can N-,N-M,-M
sp,RYO
Plantago,1.11
```

We can also easily paste files together

```
$ cut -d',' -f4,1,3 ideff.csv > file1
$ cut -d',' -f4,1,5 ideff.csv > file2
$ paste -d'|' file1 file2  # paste files together
exp,IDKirw,RYO|exp,RYO,Sel
GE1,3.93,1.11|GE1,1.11,0.2
GE1,2.98,0.67|GE1,0.67,0.32
...
```

Text files manipulations

To continue in pasting, we can also `join` files, its options define on which field to match files together

```
$ join -1 3 -2 2 -t, file1 file2
RYO,exp,IDKirw,exp,Sel
1.11,GE1,3.93,GE1,0.2
0.67,GE1,2.98,GE1,0.32
...
```

Beside printing unique lines in the file (by `uniq` command), we can also print common lines for two files (by `comm` command).

Exercise

For `ideff.csv` (look at the 1st lesson for help)

- get unique names from the first column
- sort reversarily by 3rd column
- append line numbers (check man of `cat`)

`tr` as *translate*

`tr` works on characters and can

- delete (`-d`)

```
$ printf "%s\n" "12 Files can be found in 54
    directories" | tr -d '1-9'
Files can be found in  directories # remove numbers
$ printf "%s\n" "My BIG files are ..." | tr -d '[:
    upper:]'
y  files are ... # remove upper cases
```

- substitute

```
$ printf "%s\n" "My BIG files are ..." | tr 'MB' 'mb'
my bIG files are ...
$ cat ideff.csv | tr '[:upper:]' '[:lower:]'
exp,sp,idkirw,ryo,sel,compl,net,oi
ge1,plantago,3.93,1.11,0.2,0.8,1,1.06
...
```

Regular expressions

Regular expression (regex) describes a set of possible input strings; are built-in `vi`, `emacs`, `sed`, `awk`, `perl`, `python`, ... various syntax!

- if string is substring of given string or text file
- . (dot) matches any character (use backslash for regular dot)
- [] is for character class, eg. `[abc]` maches any of character abc, `[Bb]ye` matches bey and Bye, we can use ranges `[1-9]`, `[a-e]`, `[1-9a-e]`
- negation by caret `[^ eo]`
- named classes `[a-zA-Z]` for `[[:alpha:]]`, `[a-zA-Z0-9]` for `[[:alphanum:]]`, `[45a-z]` for `[45[:lower:]]`
- anchors match beginning ^ or end $
- `*` means zero or more repetitions (`ya*y` matches yay, yaaaay, ...), `{n}` n occurences, `{n,}` n or more, `{n,m}` n but max m (.0, same as .*, a2,3 matches aaa and aaaa)
- brackets: abc* matches ab, abc, abcc, .. (ab)2,3 matches abab, ababab

Searching for lines

`grep` is a global regular expression print, eg.

```
$ grep SE ideff.csv # print lines with SE string
SE,ss,20327.8,1.15,2157.34,1721.18,3878.52,-0.29
...
$ grep "^G" ideff.csv # print lines starting with G, (caret)
GE1,Plantago,3.93,1.11,0.2,0.8,1,1.06
$ grep "2$" ideff.csv # lines ending with 2
GE2,Holcus,3.773,2.52,0.97,0.265790,1.231985,-0.053142
$ grep -v Holcus ideff.csv # print lines do not have Holcus
exp,sp,IDKirw,RYO,Sel,Compl,Net,OI
...
$ grep "Holcus\|Briza" ideff.csv # lines having Holcus or Briza
GE2,Holcus,3.773,2.52,0.97,0.265790,1.231985,-0.053142
```

Exercise

Write command that find all occurences of the string Holcus (case insensitive) in all .csv files in the current directory.

`sed` as *stream editor* - substitute

or Stream oriented non-interactive text EDitor. `sed` is filter, do not modify original file, but std. output. Sed is fast and concise.
Answer for: How to substitute 'Prunella' for 'allheal' or numbers in the file? How to delete first line?

```
$ cat ideff.csv | grep Prunella
GE1,Prunella,2.98,0.67,0.32,0.52,0.84,0.99
GE2,Prunella,1.526,0.62,0.83,0.295264,1.125914,-0.028994
$ sed "s/Prunella/allheal/g" ideff.csv | grep allheal
GE1,allheal,2.98,0.67,0.32,0.52,0.84,0.99
GE2,allheal,1.526,0.62,0.83,0.295264,1.125914,-0.028994
$ sed "s/[[:digit:]]/_/g" ideff.csv | tail
GE_,Holcus,_.___,_.__,_.__,_.______,_.______,-_.______
```

Pattern have often 3 parts:

- command: substitute `s` (`5 s`,`5!s`, `5,10s`), append `a`, insert `i`, delete `d` (`1d`, `1`$\tilde{2}$`d`), `p` print (`1,5p`), ...
- what/for what to change
- range on line `g` - all occurances on line; without it only the first

sed as *stream editor* - print,append

How to delete first line or print exact lines?

```
$ cat ideff.csv | sed '1d'
GE1,Plantago,3.93,1.11,0.2,0.8,1,1.06
...
$ sed '/^$/d' ideff.csv # delete blank lines, ^ beginning, $ end of line
...
$ sed -n '5,6p' ideff.csv # print 5-6. line, -n print only lines with p
GE1,Agrostis,4.55,1.13,0.13,0.66,0.78,0.99
GE2,Holcus,3.773,2.52,0.97,0.265790,1.231985,-0.053142
# delete lines in matched range
$ sed '/Agrostis/,/Lychnis/d' ideff.csv
...
```

Append line before the matched lines

```
$ sed "/Plantago/i NEWONE:" ideff.csv
exp,sp,IDKirw,RYO,Sel,Compl,Net,OI
NEWONE:
GE1,Plantago,3.93,1.11,0.2,0.8,1,1.06
...
```

sed as *stream editor* - complex

To combine more command use `-e`

```
$ sed -e '1,2s/[1-9]/x/g' -e '1d' ideff.csv
GEx,Plantago,x.xx,x.xx,0.x,0.x,x,x.0x
GE1,Prunella,2.98,0.67,0.32,0.52,0.84,0.99
```

More complex matching can be done by specifying patterns in () and later used by backslach and order of ()

```
# print non digit start of line before , matched by () pasted by backslash 1
$ sed 's/\([^1-9]*\),\(.*\)/\1/' ideff.csv
exp,sp,IDKirw,RYO,Sel,Compl,Net
GE1,Plantago
...
# switch 1. and 2. column divided by ,
$ sed 's/\(.*\),\(.*\)/\2,\1/' ideff.csv
OI,exp,sp,IDKirw,RYO,Sel,Compl,Net
1.06,GE1,Plantago,3.93,1.11,0.2,0.8,1
```

Drawbacks: do not remember text from one line to another, no facilities to manipulate numbers, cumbersome syntax

Exercise

Run and get `ideff.csv` (beginning of this lesson)

```
$ mkdir textfiles; cd textfiles; touch fil{k..o}
$ touch fil{0..17}.log; touch {20..25}; touch a.x{a..e}
```

Exercise

- print all lines with negative numbers from `ideff.csv`
- print all files having: (1) name from 4 characters (2) number together with alphabetic character in the name
- create new `ideffP.csv`, removing - mark from the original file
- print all files in the current folder with .* suffixes in upper style
- create new `ideffL.csv` by changing the first column of `ideff.csv` in lower style
- print all files with . in name and change . for `DOT` (use backslash)
- write command that will find `text` in all files in the current folder except of files that end with `.log`

Exercise solutions

Exercise

- print all lines with negative numbers from `ideff.csv`

```
grep - ideff.csv
```

- print all files having: (1) name from 4 characters (2) number ...

```
ls ????; ls | grep '^.\{4\}$'  # same result
ls | grep "[[:alpha:]][[:digit:]]"
```

- create new `ideffP.csv`, removing - mark from the original file

```
sed 's/-//g' ideff.csv
```

- print all files in the current folder with .* suffixes in upper style

Exercise solutions

Exercise

- create new `ideffL.csv` by changing the first column of ...

- print all files having . (dot) in name and change . in `DOT` (use backslash)

```
ls | sed 's/\./DOT/g'
```

- write command that will find `text` in all files in the current folder ...

Homework 2

Download
`http://botanika.prf.jcu.cz/fibich/bash/home2.tar.gz`, unpack (e.g. `tar -xvf home2.tar.gz`). Write a script that if you run it inside `home1` folder, will print (instead of ... there are values or lines)

```
$ createSum.sh  # print folders and number of subfolders
SE  6
GE1  6
GE2 15
$ createSum.sh -c  # compound files
SE,plpru,-0.9117911229,-0.4025109039,...,-0.4606028115,0
SE,plach,0.3984090416,0.0980536463,...,-0.4888029837,1
...
GE1,ssws,1.4582010773,0.4898979486,...,1.1983342482,1
...
```

Homework 2 - details

```
$ createSum.sh -c -p :  # compound files and specify separator
SE:plpru:-0.9117911229:-0.4025109039:...:-0.4606028115:0
SE:plach:0.3984090416:0.0980536463:...:-0.4888029837:1
...
GE1:ssws:1.4582010773:0.4898979486:...:1.1983342482:1
...
$ createSum.sh -h  # will print short help
```

Homework 2 - details

Homework1

- Script will create text on std. output based on files in folders in the current folder.
- It takes 1st folder as value for 1st column, 2nd folder (inside 1st folder) as the second column and values inside 1st/2nd/data.csv are as following columns.
- User can specify 3 options: `-h` for short help, `-c` to compound files, `-p` takes argument that is used as separator of columns (eventhrough there are , in data.csv). Use `getopts`!
- Homework is not team work, do it yourself.
- Send me the solution (script name according your surname: createSum.SURNAME) by email before 18:00 10 April
- Shorter script better script!

awk

awk

awk

AWK is an interpreted programming language designed for text processing and typically used as a data extraction and reporting tool. It is a standard feature of most Unix-like operating systems.

```
$ awk -F, '{print $2,$1}' ideff.csv # print 2. and 1. column
sp exp
Plantago GE1
..
$ echo $PATH | awk -F: '{print toupper($1)}' # -F is field sep.
/USR/LOCAL/BIN
$ awk -F, 'BEGIN{print "HI"} {print $2,$1}' ideff.csv
HI
sp exp
..
```

Basic structure of the script is in ”

```
BEGIN{ print "START" } # what is run before 1. line
 { print } # what is run for each line of the input
END { print "STOP" } # what is run after the last line
```

awk over sed

awk pattern action language like sed (but convenient number processing, conventional way of accesing fields, flexible printing, built-in arithmetics and string functions). No variable declaration.

```
$ awk 'BEGIN{sum=0} {sum++} END{print sum}' ideff.csv
15
$ awk '{print $0}' ideff.csv # the same as cat
...
$ awk -F, '{print $3*$4}' ideff.csv # multiply 3. and 4. column
...
```

Few built-in variables

FS (OFS) field separator (passed by -F option) and output separator

NF number of fields in the line (`print $(NF-1)` to print pre-last column)

NR line number (`print NR,$NF` print last column with line numbers)

- FILENAME, ARGC, ARGV ...

awk - selection

```
awk 'searchPattern {commands}'
```

Easy conditional selection by comparison, computation or by string

```
$ awk -F, '$3 > 3 {print $3}' ideff.csv # 3. column bigger than 3
$ awk -F, 'log($3) > 3 {print $3}' ideff.csv # log of 3. column
$ awk -F, '$1 == "SE" {print $3}' ideff.csv # column matching
$ awk -F, '/ll/ {print $3}' ideff.csv # string matching
$ awk -F, '/1$/ {print $0}' ideff.csv # lines end with 1
...
```

To combine conditions, use && or `||`

```
$ awk -F, '$3 > 3 && $2 > 1 {print $3}' ideff.csv
...
```

Line can by selected by number of fields

```
$ awk -F, 'NF >3 {print $3}' ideff.csv # more than columns
...
```

`awk` - built-in

`awk` contains a number of built-in functions

- string - `length` (length of string), `substr`(s,m,n) substring of s from m-th position at most n characters, `split`(s,a,d) place elements of s delimited by d into array a, `sub`, `toupper`, `tolower`, `printf` print formating in c style
- arithmetics - `sin`, `cos`, `atan`, `int`, `exp`, `log`, `rand`, `sqrt`, ...
- special - `system` (executes a linux command, `system("clear")`), `exit` (stop and go to END)

To get environmental variable, `ENVIRON["VARNAME"]` is used.

```
$ awk 'BEGIN {print ENVIRON["PATH"]}' ideff.csv
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

awk - built-in

Also control of the flow by if-else, while and for loops.

```
... END { if (n>0) print n} ...
... {while (i <= $3) { printf("%f",$1*$2+i); i = i +1} }
```

Arrays subscripts can have any value (also associative arrays), elements are not declared.

```
$ awk -F, '{line[NR]=$0} END { for(i=NR;i>0;i=i-1) print(
    line[i])}' ideff.csv
...  # print file clockwisely
# for loop for associative array
for (v in array) { print array[v] }
```

Own functions can be defined too

```
$ awk -F, 'function foo() { a = exp($4); printf("%f\n",a)
     } { foo() }' ideff.csv
...
```

awk script

We can have awk script

```
$ cat awk.a
#! /usr/bin/awk -f
{ print $1}
$ chmod u+x awk.a
$ ./awk.a ideff.csv
```

Awk can be easily combined with other commands

```
$ ls -l | awk '{print $5}'  # list of file sizes
```

First version released 1977 by Aho-Weinberg-Kernighan, actual verion is often called as new awk. Beside `awk` there is GNU awk, often run by `gawk` that is also ported to many OSs.

`awk` - exercise

Download

`http://botanika.prf.jcu.cz/fibich/bash/inteff.csv`

Exercise on `awk`

- get only number of lines from the command `wc -l inteff.csv`
- print second column from `inteff.csv` having ”pru” inside
- use `inteff.csv` to print file in form

```
exp=GE1,mix=plpru,int=5.74,Sel=0.35
exp=GE1,mix=plach,int=3.5,Sel=0.29
```

- print number and length of each line in given file
- print every third line of the file, later get value that determines which line to print from environmental variables
- print sum, max and mean of 3. and 5. column of `inteff.csv`
- count frequencies of values in the first column of `inteff.csv`
- write a command to print the columns in a text file in reverse order

big exercise 1

big exercise 1

Exercise

Exercise

Write command or script that

- print the content of variable `PATH` on more lines, change : by new line
- counts characters of the current user name and number of user's processes
- for variable `i` having content: some aplhas, than some digits and than some alphas, print just last alphas, e.g. `i=abc76hell` print `hell`, `i=hh3five` print `five`
- count sum of all lines of all files in the current directory
- print file names and their owners in the current folder, nothing else
- print only the shortest and the longest file names in the current directory

Exercise

Exercise

- print all files having 'a' as the second character
- write command that will create directories, sub-directories, sub-sub-dir... according variable MFO, e.g. for MFO=first_second_third, will create three directories first/second/third
- write command that creates MFO variable from the current directory
- create function `mkcd` that creates and goes into directory (1. arg.)
- write script having 3 arguments: in the file (name as the 1. argument), change/substitute string (2. argument) for the string (3. argument)
- write script having 2 arguments: 1. is file name (`f`) and 2. is number (n), it creates two files `f_begin` and `f_tail`, that contains first n lines and the rest lines of the file `f`, respectively

GNUplot

GNUplot

GNUplot

GNUplot is command-line interactive plotting program available for many OSs, see `http://www.gnuplot.info`. To run and quit

```
$ gnuplot
G N U P L O T
Version 4.6 patchlevel 0     last modified 2012-03-04
...
gnuplot> quit
```

- for help with commands, use `help command`
- commands can be shortened (eg. `p` for `plot`)
- commands are separated by ;
- comments are done by #
- shell commands start with ! (eg. `!  ls`)
- filenames have to be enclosed by quotes
- we can prepare scripts, sequence of gnuplot commands

```
$ gnuplot gpscript # run gnuplot script
...
gnuplot> load 'gpscript' # run script from gnuplot
```

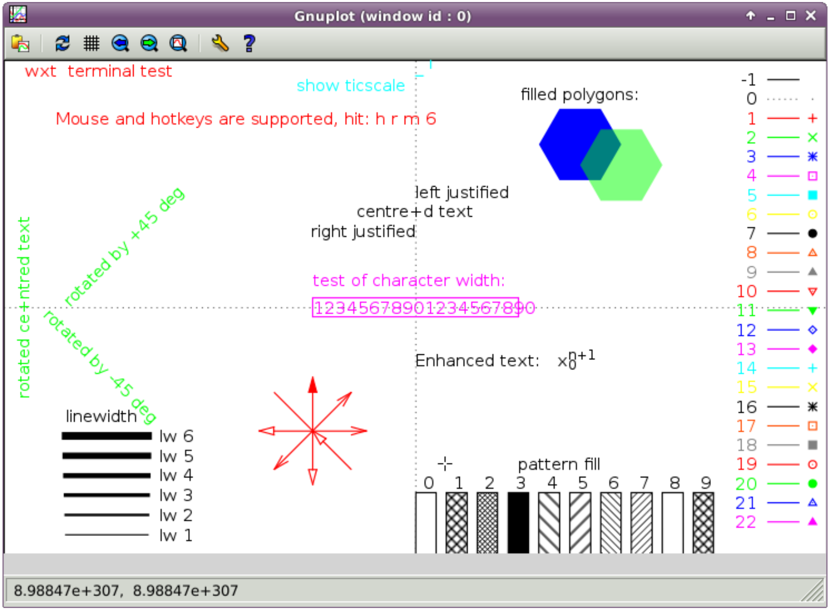
GNUplot output device

```
gnuplot> set terminal  # list of output devices
Available terminal types:
       cairolatex  LaTeX picture environment using
          graphicx package and Cairo backend
           canvas  HTML Canvas object
              cgm  Computer Graphics Metafile
...
$ cat gpscript   # list of gnuplot script
set terminal png
p sin(x)/x
$ gnuplot gpscript > obr.png # run script and store figure
```

After start, if you see Terminal type set to ’unknown’, than install gnuplot-x11 package to see figures directly (not exported into files).

```
$ gnuplot
...
gnuplot> p sin(x)  # print graph of sin(x)
gnuplot> test  # print test figure for terminal
```

GNUplot test device



8.98847e+307, 8.98847e+307

GNUplot formatting options

- place or hide key

```
set key top center
set no key
```

- set a title

```
set title "the title"
```

- axis labels

```
set xlabel "pH"
set ylabel "a"
```

- plot an arrow `set arrow from 0.5,0 to 0.5,0`
- define a label `set label "b point" at 0.5,0`
- border `set border lw 3`
- ticks `set ytics 0.5`, `set xtics 0.1`
- linewidth (`lw`), pointsize (`ps`), line or point type `lt, pt`

After changing of settings, it is often necessary to run replot `rep`.

GNUplot input files

gnuplot can read data from files

```
# which changes are done by following command?
$ sed -e '1d' -e 's/,/ /g' ideff.csv >ideff2.csv
gnuplot> plot 'ideff2.csv' using 4:8 # 4. and 8. col scatter plot
# short commands with point type
gnuplot> p 'ideff2.csv' u 4:8 pt 8
# multiple data series
gnuplot> p 'ideff2.csv' u 4:7 pt 8, 'ideff2.csv' u 4:6
gnuplot> p 'ideff2.csv' u 4:($7/5) # do math on column
```

Columns are indexed from $1 and $0 is for running index.
To change scale, zoom or style

- `[un]set logscale [xy]`
- `set xrange [0:10]; set yrange [*:*]` y is set automatically
- multiplot options by `set multiplot`

You can specify column separator `set datafile separator ";"` and also presence of headline `set key autotitle columnhead`.

GNUplot built-in

Many built-in function for

- math - abs, sin, log, exp, round, floor, pi, ..
- strings - gprintf, sprintf, strlen, substr
- other - system (calling shell command), column (get column), value (value of variable), rand (random value), fitting of functions, `reset` settings.

and also operators. User can easily prepare own variable or functions

```
gnuplot> a = floor(tan(pi/2 - 0.1))
gnuplot> print a
gnuplot> sinc(x) = sin(pi*x)/(pi*x)
gnuplot> p sinc(x)/x
gnuplot> min(a,b) = (a < b) ? a : b
```

Nice demo files at `http://gnuplot.sourceforge.net/screenshots/index.html#demos`

GNUplot example 1

```
$ wget http://botanika.prf.jcu.cz/fibich/bash/using.dat
$ less using.dat
$ gnuplot
```

We will plot 4-6. column (y-axis) based on 3. column (x-axis) by impulses, points and lines.

```
gnuplot> plot 'using.dat' using 3:4 title "Logged in"
    with impulses,\
'using.dat' using 3:5 t "Load average" with points,\
'using.dat' using 3:6 t "%CPU used" with lines
```

To set title, move the legend and xlab, you must call before the plot

```
gnuplot> set title "Computer load"
gnuplot> set key below
gnuplot> set xlab "Days"
```

To change the color and point size and type use

```
'using.dat' using 3:5 t "Load average" with points pt 3
    ps 4 lc rgb "cyan",
```

GNUplot exercise

Exercise

- write function that will get data for specified experiment from the file `ideff.csv`, argument of the function will be the name of the experiment (1. column)
- plot dependence of 6. col on 5. col (`u 5:6`) for GE1 and GE2 in one figure
- write script (or function) that will have 1 argument (filename), script will run gnuplot, that will print scatter plot of the first and second column into some graphical file (eg. add png suffix for the input filename)
- add title to the figure according to the given filename
- use boxplot to differentiate values of the fifth column of GE1 and GE2

```
gnuplot> set style data boxplot # pre-set the data style
```

- add xlab according the experiment and ylab

R

R

R introduction

R is ’GNU S’, a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. See `http://cran.r-project.org/`.

To run and quite R

```
$ R
R version 3.0.3 (2014-03-06) -- "Warm Puppy"
Copyright (C) 2014 The R Foundation for Statistical
   Computing
Platform: x86_64-pc-linux-gnu (64-bit)
...
> plot(1:10) # plot number from 1 to 10
> q() # quit R
Save workspace image? [y/n/c]: n
$
```

R introduction

Quite intuitive environment, in examples

```
> a=4+6  # assign the value to variable
> a  # print variable
[1] 10
> 4:10  # create sequence
[1]  4  5  6  7  8  9 10
> a+4:10  # sum value with sequence
[1] 14 15 16 17 18 19 20
> ls()  # list of current variables
[1] "a"
> rm(a)  # removing the variable
> getwd()  # get current directory, setwd() set it
[1] "/home/pvl/Documents/prf.jcu/bash"
> sqrt(5)  # square root of 5
[1] 2.236068
> seq(3,12,1.2)  # sequence defined by step
[1]  3.0  4.2  5.4  6.6  7.8  9.0 10.2 11.4
> ? seq  # get help for the function
```

R why

Why to know R?

- currently the most widespread statistical software
- great graphical features
- thousands of packages (addons with specialized functions) mostly for everything
- many tools for data manipulation (`grep`, `merge`, `split`, `aggregate`, `apply`, `uniq`, `sort`, `gsub`...)
- power of programming language (variables, conditions, loops, function, classes, regex, easy math, ...)
- binaries for Mac, Windows and Linux
- it is free

Often connected with some integrated development environment (IDE), e.g. R studio (`https://www.rstudio.com/`)

R start

Instead of interactive mode, R can be run with script (see `man R`)

```
$ cat script.r # script with R code
sqrt(2:20)
$ R -q --vanilla < script.r > out.r # run script
$ cat out.r # see the output
> sqrt(2:20)
 [1] 1.414214 1.732051 2.000000 2.236068 2.449490
     2.645751 2.828427 3.000000
...
```

or we can specify interpreter in the script

```
$ cat run.R # R script with interpreter
#!/usr/bin/Rscript
sqrt(2:20)
$ ./run.R # can be run directly in the terminal
 [1] 1.414214 1.732051 2.000000 2.236068 2.449490
     2.645751 2.828427 3.000000
...
```

R start

R often load and store current environment in the file `.RData` (there are stored current variables, function, settings, ...), but it depends which options where specified during the run of R.

- `--no-environ` do not load `.RData`
- `--no-save` do not save `.RData`
- `--vanilla` is shortcut for `--no-save, --no-restore, --no-site-file, --no-init-file` and `--no-environ`

In the interactive mode after quit (`q()`), R is asking if to save environment in to `.RData`. Image can be saved during the sesstion too (`save.image()`).

R packages

Packages are often installed in the personal library in the home directory `/home/$USERNAME/R`.

```
> search()  # what is currently loaded
> library()  # see all installed packages
```

Sometimes it is usefull to tell R where to install and where to get packages (e.g. for package called `fork`)

```
> install.packages("fork", lib="/home/pvl/Documents/prf.
   jcu/bash/r")
> library("fork", lib="/home/pvl/Documents/prf.jcu/bash/r
   ")
```

To install already downloaded package you can use

```
$ wget http://cran.r-project.org/src/contrib/fork_1.2.4.
   tar.gz
$ R CMD INSTALL fork_1.2.4.tar.gz
```

Many of packages are in repositories of OSs.

R data

To get data in, or store them in files, there are `read.*` and `write.*` functions (press TAB after dot to get the list of available function like in terminal)

```
> read.
read.csv read.delim2 read.table read.csv2 read.delim ...
> write.
write.csv write.csv2 write.ftable write.table ...
```

To read csv from the homework 2, you can use

```
> sla = read.csv("sla.csv") # read csv file
> class(sla) # check the data type
[1] "data.frame"
> summary(sla) # see the summary
      species           plot          LEAFAREA_mm2
 FestRubr : 11   Min.   : 1.00   Min.   :   52.37
 CirsPalu :  9   1st Qu.: 6.00   1st Qu.:  793.94
 AgroCani :  8   Median :12.00   Median : 1508.18
 GaliUlig :  8   Mean   :12.53   Mean   : 2252.80
...
```

R data out

You can easily store your results of data manipulations

```
> sla2 = sla  # copy of variable
# to work with column you often use $ after variable name, then col. name
> sla2$LEAFAREA_mm2 = log(sla2$LEAFAREA_mm2 + 1 )
> summary(sla2)  # see the summary
      species          plot         LEAFAREA_mm2
 FestRubr : 11    Min.    : 1.00    Min.    :3.977
 CirsPalu :  9    1st Qu.: 6.00    1st Qu.:6.678
 AgroCani :  8    Median :12.00    Median :7.319
 GaliUlig :  8    Mean   :12.53    Mean   :7.284
...
```

and save it into file

```
# to write csv without rownames and quoting the text
> write.csv(sla2,"sla2.csv",row.names=FALSE,quote=FALSE)
# in write.table you can specify delimiter
> write.table(sla2,"sla2a.csv",row.names=FALSE, quote=
    FALSE,sep=":")
```

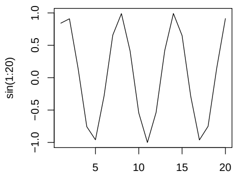
R graphics

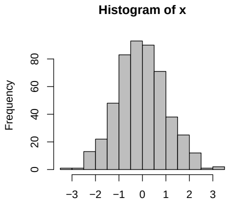
R terminal often open R graphic device when you plot, but you can easily redirect output to the file

```
> ?pdf  # or tiff, jpeg, png or bmp
> pdf("myout.pdf",width=4,height=4)  # graphical device in inches
> plot(sin(1:20),type='l')  # some plot command
> dev.off()  # store the output into file
```

To easily create histogram you can use

```
> ?hist
> x<-rnorm(500)  # 500 random numbers in Gaussian distribution, ?rnorm
> hist(x,col="gray")  # histogram of x in gray color
```





R habits

Working in R, you should

- check help pages (`?command`)
- check data, e.g. for variable `a`

```
> summary(a)
> a  # print content of variable
> dim(a)  # size of variable, e.g. for tables (data.frame, matrix)
> length(a)  # length of variable
> class(a)  # type of variable
```

- be careful with = (or <-) it can destroy your data in memory
- prepare a script with sequence of command (e.g. in RStudio where you can run script directly) for futher re-use
- be careful what is loaded (e.g. use `ls()` to check which variables were loaded from `.RData`)
- use comments (after #) in your code

R

Exercise

- plot histogram of LEAFAREA from the `sla.csv` (homework 2)
- create boxplot (`?boxplot`) of column name `RYO` from `ideff.csv` *per* column name `exp` (one figure with 3 boxes)

```
> boxplot(COLUMNforYaxis~COLUMNforXaxis)
```

- write script (or function) that will have 2 arguments (1. is filename), script will run R, that will do histogram of the column (2. argument), result will be saved into some graphical file (eg. add pdf suffix for the input filename)
- try to create the same plot as in GNUplot example based on `using.dat`

R

Exercise - solution

- write script (or function) that will have 2 arguments ...

```
$ cat rscr # R script version
#!/usr/bin/Rscript
args <- commandArgs(trailingOnly = TRUE)
mf=read.csv(args[1])
hist(mf[,args[2]])
$ ./rscr ideff.csv RYO
$ cat bscr # bash script version
echo "mf=read.csv(\"$1\")" > rfile
echo "pdf(\"our.pdf\")" >> rfile
echo "hist(mf\$$2)" >> rfile
echo "dev.off()" >> rfile
R --vanilla < rfile
$ ./bscr ideff.csv RYO
```

perl

Practical Extraction and Report Language alias `perl` is complete language with many adjectives introduced by Larry Wall in 1987. Perl uses syntax and concepts of `awk`, `sed`, `C`, `bash`, ... It stands on thousands of third-party modules stored in the repository Comprehesive Perl Achive Network (CPAN).

```
$ cat hello.pl  # perl files often end with .pl
#!/usr/bin/perl
print "Hello World.\n";
$ ./hello.pl
Hello World.
$ perl hello.pl  # other way
$ perl -d hello.pl  # to use perldebug
```

For basic help and man pages use

```
$ perl --help
$ man perl
```

perl characteristic

- was meant to be a sort of shell-script on steroids (condense syntax)
- large library support
- many OS and it is free
- powerful text processing facilities without the arbitrary data-length limits of many contemporary Unix commandline tools
- actual version 5.18.2 (January 7 2014)

Programming language features

- various variable types
- OO
- ability to package code in reusable modules
- automatic memory management

perl modules

The easy way to install perl module (library) is through CPAN (first run often pre-set environment)

```
$ perl -MCPAN -e shell
cpan> install HTML::Template
cpan> quit
```

It often pre-set `~/.cpan/` and install libraries in the `~/perl5/`.
Manual way of install package is done by getting code of module from `http://search.cpan.org/`, e.g.

```
$ wget http://search.cpan.org/CPAN/authors/id/W/WO/WONKO/
    HTML-Template-2.95.tar.gz
$ tar -xvf *.tar.gz
$ cd HTML-Template-2.95
$ perl Makefile.PL
$ make; make test
$ make install
```

You can also specify prefix where to install module (`PREFIX=/folder` after `Makefile.PL`).

perl modules, example

To check if module is properly installed, you can by

```
$ export PERL5LIB=~/perl5/lib/perl5 # export PATH to perl modules
$ perl -e "use HTML::Template" # run script that load module
# if the output is no error, that in is ok
```

Many of packages are also in standard operating systems repositories (e.g. `apt-cache search perl MODULENAME`).

Few examples

```
# print first 3 columns
$ perl -pale '$_="@F[0..2]"' using.dat
# change GE for GL in the file and store the original
$ perl -pi'.orig' -e 's/GE/GL/' ideff.csv
$ cat b64.pl # script to decode argument from Base64 code
#!/usr/bin/perl
use MIME::Base64;
print decode_base64($ARGV[$1])
```

python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms. See `https://www.python.org`.

```
$ cat hello.py   # python sctipt often end with .py
#!/usr/bin/python
print 'Hello, world!'
$ ./hello.py # run python script
Hello, world!
$ python hello.py # other way of run
$ python -v hello.py # to debug what is loading during the execution
```

For basic help and man pages use

```
$ python --help
$ man python
```

python characteristic

- elegant syntax
- large library support
- easy extendable by modules, even in C or C++
- many OS and it is free
- ideal for prototype development and other ad-hoc programming tasks
- actual version 3.4.0 (March 17 2014)

Programming language features

- variety of basic variable types
- OO, generator and list comprehensions
- code in modules or packages
- supports exceptions
- automatic memory management

python packages

To install python package you can use easy ways by `pip` or `easy_install`

```
$ sudo python get-pip.py # you must download get-pip
# previous command is run only once
$ pip search PACKAGENAME # to search package
$ sudo pip install bioinfo --process-dependency-links
# install bioinfo package with dependencies
```

To chek if it was successful

```
$ python -c "import bioinfo;" # run script that load the package
```

The more manual way on already downloaded package

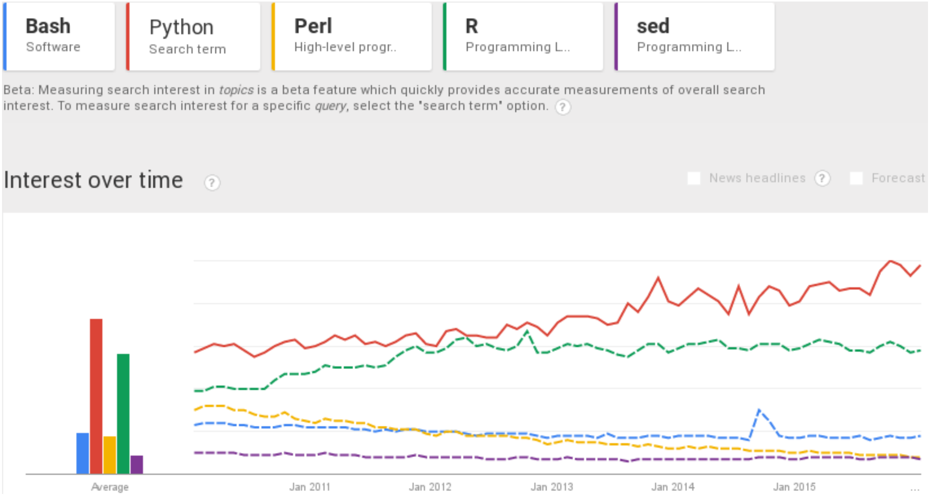
```
# extract the package, get in directory
$ TOIN=/software/EXPECTED_FOLDER # install folder
$ python setup.py install --install-scripts=$TOIN/bin/ --
    install-purelib=$TOIN/lib --install-lib=$TOIN/lib
```

Later, sometimes it is necessary to set `PYTHONPATH` and `PATH` variables to `$TOIN/lib` and `$TOIN/bin`. Many modules are also in OS repositories.

python examples

```
$ cat name.py # get input from user
#!/usr/bin/python
name = raw_input('What is your name?\n')
print 'Hi, %s.' % name
$ ./name
What is your name?
pvl
Hi, pvl.
$ cat sum.py # sum of integer arguments
#!/usr/bin/python
import sys
try:
    total = sum(int(arg) for arg in sys.argv[1:])
    print 'sum =', total
except ValueError:
    print 'Please supply integer arguments'
$ ./sum.py 3 4 5
sum = 12
```

tools - google trends


Bash
Software

Python
Search term

Perl
High-level progr..

R
Programming L..

sed
Programming L..

Beta: Measuring search interest in *topics* is a beta feature which quickly provides accurate measurements of overall search interest. To measure search interest for a specific *query*, select the "search term" option.

Interest over time

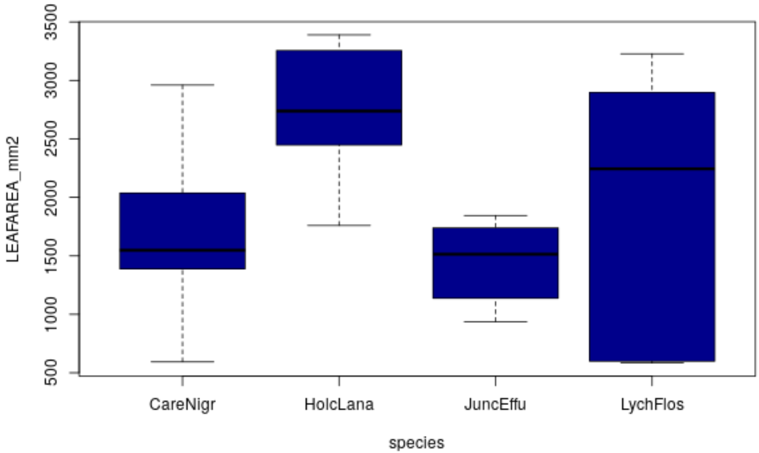
Homework 3

Download
`http://botanika.prf.jcu.cz/fibich/bash/sla.csv`.

- Write a script that have two arguments: filename (eg. sla.csv) and non-negative number (eg. X).
- Script will create one picture (whatever format pdf, jpg, tiff, png, ...), but having the same name as the first argument (e.g. it will create sla.pdf), where one can observe difference of LEAFAREA (the third column in the file) of different species (name from the first column of the file).
- For the plotting, we use ONLY species that have exactly X (2. argument of the script) occurences in the file.
- Send script (name according your surname: `plotdif.SURNAME`) by email until 24:00 8.5.
- For the example see following slide (it is not necessary to use boxplot).

Homework 3

```
$ plotdif.sh pic.csv 6  # create figure pic.pdf for species with exactly 6
      occurences
```



big exercise 2

big exercise 2

Big Exercise

Exercise

Write one line command that will

- check if the number of files in the current directory is bigger than 5
- set variable to the names of all files in the current directory
- print current year, month and day, in format, eg. `2014 Jan Mon`
- create directories from 1 to 99, each with the file `no` containing directory number plus 130
- create variable `evenNAME` containing names of directories with even number
- create variable `oddNO` containing number of directories with odd number
- delete directories containing 0 in the name
- for file `test` print number of lines having some upper case character

Big Exercise

Exercise

Write a function that will

- make its only one argument executable file, add argument checking
- print the last and the first line of the file given as the argument
- count lines that do not contain character specified as 1. argument in the file specified as 2. argument
- inform you if the seconds of the current time are more or less than 30
- run `date` command only if there are more than 3 files in the current folder
- substitute string (first argument) for string (second argument) in the file (third argument)
- remove all white spaces from the argument (eg. output of `date`)

Big Exercise

Exercise

Write a script or function that will

- check if varible `MYVAR` is in range 10-30 and inform about it
- create files `myfileX`, where X will be changed for all small case letter in the alphabet
- for aguments (always one arg.) like: `xxfileRRTT34`, `yzrrrru33`, remove and print strings without 3-6. characters
- create file every 3 seconds, name will be according to the seconds of the current time
- print `sla.csv` file with log transformed last column
- print range of numbers between lines (1. and 2. argument) from the file (3. arg)

Big Exercise

Exercise – regex

- write function that for aguments (there will be always only one argument) like: `abcd34g`, `a1`, `uhrfdsa355jj` print lengths of characters before digits (digits have various length) and after digits; so 2 numbers appear as the output, e.g. `4 1`, `1 0`, `7 2`
- write function or script that for arguments (there will be always only one argument) like `1232fa33a`, `2errerew9bb`, `11a9ere` will print difference of the numbers in the argument, e.g. `1199`, `-7`, `2`

```
$ mydiff 1232fa33a
1199
$
```

- write function or script that for arguments (there will be always only one argument) like `1232fa33a`, `2errerew9bb`, `11a9ere` will print only characters without digits at the beginning and also print theirs length, e.g. `fa33a 5`, `errerew9bb 10`, `a9ere 5`