Praktiline referents

Linux/Unix/macOS käsurea kiirõpik

Autor: ChatGPT

Teda abistas: Jaak Vilo

Mustand: sisu ei ole veel tehniliselt ega keeleliselt täielikult kontrollitud ega toimetatud.

Kolofon

Linux/Unix/macOS käsurea kiirõpik

Autor: ChatGPT

Teda abistas: Jaak Vilo

Mustand: sisu ei ole veel tehniliselt ega keeleliselt täielikult kontrollitud ega toimetatud.

Versioon: v0.2.0-draft

Kood: v0.2.0-draft-12-g5af973a

Commit: 5af973a

Tag: v0.2.0-draft

Kuupäev: 2026-04-15 13:57:20 BST

Sisukord

Osa I: Esimesed sammud

- Kuidas seda õpikut kasutada
- Õpitee ja õppetunnid
- Terminali esimesed sammud
- Abi leidmine: man, –help ja info
- Kataloogid ja failid
- Teksti vaatamine ja liikumine
- Failide vaatamine ja muutmine: cat, less, nano, vim
- Käskude kuju ja argumentide loogika
- Sisend, väljund, torud ja suunamine
- Esimene tervikharjutus: 30 minutit

Osa II: Süsteemi pilt ja haldus

- Linux, Unix, GNU, macOS, Windows ja shellid
- Failisüsteemi kaart
- Kettaruum ja süsteemi maht
- Õigused, omanikud ja täitmisbitid
- Kasutajad, grupid ja sudo
- Muutujad, keskkond, PATH ja aliased
- Paketihaldus: apt, dnf, pacman, brew
- Lihtne veaotsing käsureal
- Võrgu põhitööriistad

Osa III: Failid, võrk ja süsteemitöö

- Failide kopeerimine ja sünkroonimine
- Kauglogimine ja SSH
- Veebist sisu toomine ja tekstivaade: curl, wget, lynx
- Arhiivid ja pakkimine
- Tervete kataloogipuude haldus ja jagamine
- Protsessid, tööd ja signaalid
- Logid ja teenused
- Püsivad terminalisessioonid: tmux ja screen
- Graafilised rakendused kaugmasinast

Osa IV: Tekst, otsing ja automatiseerimine

- Teksti otsimine: grep ja sugulased
- Teksti teisendamine: tr, cut, paste, column, strings
- Vood ja tabelid: sort, uniq, wc, pr, join
- sed, awk ja perl praktiliselt
- `find` ja `xargs` ohutumalt

- Esimene shelliskript
- `cron` ja ajastatud tööd

Osa V: Arendus ja töövood

- Git, GitHub ja töövoog
- Pythoni venv ja eraldatud keskkonnad
- Dockeri alused
- IDE-d ja arenduskeskkonnad
- Andmeteaduse eelteadmised käsurea vaates
- CSV, JSON ja XML käsureal
- Andmebaasi algus: sqlite ja Python
- Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java
- LaTeX käsurealt

Lisad

- Lisa A: kopeeritavad minitestid
- Lisa B: spikrite register
- Lisa C: sõnastik ja terminoloogia
- Lisa D: mis veel on puudu ja mida lisada järgmisena

Kuidas seda õpikut kasutada

See õpik ei ole eelkõige pikk jutustav raamat. See on praktiline tööriist:

- kiireks meeldetuletuseks
- käskude loogika õppimiseks
- näidete kopeerimiseks ja läbi proovimiseks
- harjutuste tegemiseks

Iga peatüki soovituslik ülesehitus

Iga peatükk võiks sisaldada nelja plokki:

1. Lühike seletus, mis asi see on ja milleks seda vaja on.
2. Spikker: kõige tähtsamad käsud ja valikud ühel ekraanil.
3. Näited: kopeeritavad käsud, mida saab kohe terminalis proovida.
4. Minitest või harjutus: 3 kuni 10 lühikest ülesannet.

Näidete põhimõte

Näited peaksid olema:

- piisavalt ohutud, et ei muudaks kogemata päris tööfaile
- väikeste sammudena
- kopeeritavad

- kontrollitava tulemusega

Hea näide teeb korraga kaks asja:

- õpetab ühe käsu või töövõtte loogikat
- näitab väikest tervet töötsüklit algusest kontrollini

Näiteks:

```
pwd
ls
mkdir proov
cd proov
mkdir naide
cd naide
printf 'tere\nmaailm\n' > sonad.txt
wc -l sonad.txt
```

See näide tähendab:

- vaata, kus sa alustad
- loo eraldi harjutuskaust
- liigu selle sisse
- kirjuta faili kaks rida
- kontrolli, mitu rida failis on

Kaks lugemisviisi

Õpikut tasub kasutada kahel viisil:

- järjest õppides peatükist peatükki
- käsiraamatuna, kui on vaja kiirelt midagi meelde tuletada

Väljundid

Siit võiks hiljem teha:

- staatilise HTML-versiooni veebis lugemiseks
- ühe koond-PDF-i
- soovi korral ka peatükkide kaupa PDF-id

Õpitee ja õppetunnid

See peatükk aitab otsustada, millal mida vaadata. Ülejäänud raamat on kirjutatud nii, et seda saaks kasutada ka käsiraamatuna, kuid alguses on lihtsam liikuda kindla õpitee järgi.

Kui tahad liikuda võimalikult rahulikult lihtsamast keerulisemani, alusta peatükist Terminali esimesed sammud. Peatükk Esimene tervikharjutus: 30 minutit on mõeldud hiljem, kui baas on juba all.

Kuidas seda peatükki kasutada

Kui oled täiesti alguses, ära loe raamatut järjest algusest lõpuni nagu romaani. Vaata seda pigem osade kaupa:

1. kõigepealt õpi, kuidas käsurida lugeda ja kasutada
2. siis ehita juurde süsteemipilt: failisüsteem, õigused, kettaruum, paketid
3. seejärel mine failide, võrgu ja süsteemitöö juurde
4. alles pärast seda võta suuremad töövood nagu Git, Docker ja arenduskeskkonnad

See järjekord on oluline, sest hilisemad teemad ehituvad varasematele.

Näiteks:

- `ssh` kasutab sama käsurea loogikat, mida õpid varem
- `git` käsud kasutavad samu valikute ja argumentide mustreid
- `rsync`, `grep`, `find` ja torud muutuvad arusaadavaks alles siis, kui failide ja voogude põhimõte on selge
- veaotsing muutub palju lihtsamaks, kui tead juba, kus failid süsteemis elavad

Õpitee 1: täiesti algaja

Vaata peatükke selles järjekorras:

1. Kuidas seda õpikut kasutada
2. Terminali esimesed sammud
3. Abi leidmine: man, –help ja info
4. Kataloogid ja failid
5. Teksti vaatamine ja liikumine
6. Failide vaatamine ja muutmine: cat, less, nano, vim
7. Käskude kuju ja argumentide loogika
8. Sisend, väljund, torud ja suunamine
9. Esimene tervikharjutus: 30 minutit
10. Linux, Unix, GNU, macOS, Windows ja shellid

See on hea algus, sest selle järel oskad juba:

- terminalis liikuda
- faile leida, vaadata ja muuta
- abi otsida
- aru saada, miks käsud käituvad nii nagu nad käituvad
- ning alles siis paigutada need oskused Linuxi, macOS-i ja Windowsi laiemasse konteksti

Õpitee 2: süsteemi pildi loomine

Kui baas on all, liigu edasi siia:

1. Failisüsteemi kaart
2. Kettaruum ja süsteemi maht
3. Õigused, omanikud ja täitmisbitid
4. Kasutajad, grupid ja sudo
5. Muutujad, keskkond, PATH ja aliased
6. Paketihaldus: apt, dnf, pacman, brew
7. Lihtne veaotsing käsureal
8. Võrgu põhitööriistad

See plokk on tähtis, sest siin tekib tunne, et süsteem ei ole enam “must kast”.

Õpitee 3: igapäevane Linuxi ja serveri kasutaja

Kui tahad teha päris töid masinate, failide ja kaugühendustega, siis vaata eriti neid peatükke:

1. Failide kopeerimine ja sünkroonimine
2. Kauglogimine ja SSH
3. Veebist sisu toomine ja tekstivaade: curl, wget, lynx
4. Arhiivid ja pakkimine
5. Tervete kataloogipuude haldus ja jagamine
6. Protsessid, tööd ja signaalid
7. Logid ja teenused
8. Püsivad terminalisessioonid: tmux ja screen

See plokk on seotud praktilise süsteemikasutusega:

- failid liiguvad masinate vahel
- protsessid võivad kinni jääda või kaua joosta
- logidest tuleb probleeme otsida
- katkestuste vastu on vaja püsivaid sessioone

Õpitee 4: tekst, filtrid ja automatiseerimine

Kui tahad saada tugevaks Unix-laadsete tekstivoo tööriistade kasutajaks, siis liigu nii:

1. Teksti otsimine: grep ja sugulased
2. Teksti teisendamine: tr, cut, paste, column, strings
3. Vood ja tabelid: sort, uniq, wc, pr, join
4. sed, awk ja perl praktiliselt
5. `find` ja `xargs` ohutumalt
6. Esimene shelliskript
7. `cron` ja ajastatud tööd

See on üks raamatu tähtsamaid õpiteid, sest just siin tekib “väikeste tööriistade ühendamise” tunnetus.

Õpitee 5: arendaja suund

Kui eesmärk on tarkvara arendamine, siis pärast baasi vaata eriti neid peatükke:

1. Git, GitHub ja töövoog
2. Pythoni venv ja eraldatud keskkonnad
3. Dockeri alused
4. IDE-d ja arenduskeskkonnad
5. Andmeteaduse eelteadmised käsurea vaates
6. CSV, JSON ja XML käsureal
7. Andmebaasi algus: sqlite ja Python
8. Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java
9. LaTeX käsurealt

See järjekord on mõistlik, sest:

- Git tuleb peaaegu igas projektis enne
- `venv` aitab projektisõltuvused korras hoida
- Docker ja IDE on mugavus- ning töövooteemad
- andmeteaduse eelteadmiste osa aitab siduda käsurea, failivormingud ja SQL-i
- SQLite, kompileerimine ja LaTeX on head näited eri tööriistamaailmadest

Õpitee 6: andmeteaduse stardirada

Kui eesmärk on andmeteaduse või andmeanalüüsi suund, siis pärast käsurea baasi vaata eriti neid peatükke:

1. Sisend, väljund, torud ja suunamine
2. Teksti otsimine: grep ja sugulased
3. Teksti teisendamine: tr, cut, paste, column, strings
4. Vood ja tabelid: sort, uniq, wc, pr, join
5. Andmeteaduse eelteadmised käsurea vaates
6. CSV, JSON ja XML käsureal
7. Andmebaasi algus: sqlite ja Python
8. Pythoni venv ja eraldatud keskkonnad

See rada on hea sellepärast, et:

- kõigepealt õpid andmeid failidest lugema ja filtreerima
- siis saad aru, mis vahe on tabelil, JSON-il ja XML-il
- pärast seda muutub SQL palju loomulikumaks
- lõpuks saad sama töövoo viia Pythoni projekti või andmetöötlusse

Minimaalne 7 päeva plaan

Kui tahad võtta ühe lühikese esimese ringi, siis üks praktiline plaan on:

1. päev: peatükid `05`, `07`, `08`
2. päev: peatükid `09`, `10`, `06`

3. päev: peatükid `11`, `03`, `04`
4. päev: peatükid `12`, `13`, `14`
5. päev: peatükid `15`, `16`, `17`, `18`
6. päev: peatükid `19`, `20`, `21`, `22`
7. päev: peatükid `29`, `30`, `31`, `34`, `36`

Iga päeva puhul:

- loe peatüki loogika läbi
- proovi vähemalt pooled näited ise läbi
- tee peatüki minitest

Millal kasutada raamatut referentsina

Kui oled juba baasi läbinud, siis ei pea enam liikuma õpitee järgi. Siis on parem kasutada peatükke probleemipõhiselt:

- "mul on vaja faile leida" -> Kataloogid ja failid
- "mul on vaja aru saada, mis süsteemikaust kuhu käib" -> Failisüsteemi kaart
- "mul on vaja mustrit otsida" -> Teksti otsimine: grep ja sugulased
- "mul on vaja serverisse saada" -> Kauglogimine ja SSH
- "mul on vaja veebileht alla tõmmata või linke kokku koguda" -> Veebist sisu toomine ja tekstivaade: curl, wget, lynx
- "mul on vaja aru saada, miks käsk ei tööta" -> Lihtne veaotsing käsureal
- "mul on vaja sõltuvused paigaldada" -> Paketihaldus: apt, dnf, pacman, brew

Terminali esimesed sammud

Terminal on tekstipõhine viis arvutiga suhtlemiseks. Iga käsk kirjutatakse reale, vajutatakse `Enter` ja seejärel näidatakse tulemust.

Loogika

Kõige rahulikum algus on käskudega, mis ainult näitavad infot ega muuda midagi. Nii saad kõigepealt aru:

- kus sa oled
- mis selles kaustas on
- mis kasutajaga sa töötad
- mis aega süsteem näitab

Alles pärast seda tasub teha esimene väike muudatus, näiteks luua oma harjutuskaust.

1. Viip ehk prompt

Terminali real näed tavaliselt enne käsku lühikest teksti. Seda nimetatakse viibaks ehk promptiks.

Näiteks:

`~/proov %`

või:

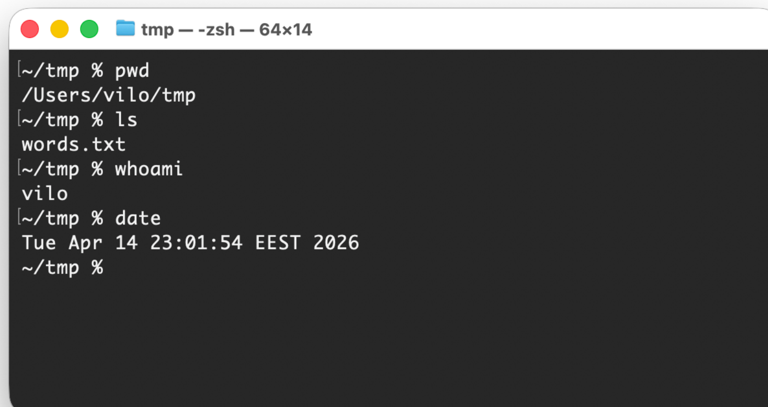
`kasutaja@arvuti:~$`

Prompt näitab tavaliselt mõnda neist asjadest:

- kasutajanime
- arvuti nime
- praegust kausta
- seda, kas oled tavaline kasutaja või kõrgemate õigustega kasutaja

Prompti täpne kuju võib olla erinev. Sellepärast on hea meeles pidada lihtsat rusikareeglit:

- prompt on kasutajaliides
- `pwd` ütleb kindlalt, kus sa päriselt oled



Joonis 1: Terminali näide, kus kasutatakse ainult infot andvaid käske `pwd`, `ls`, `whoami` ja `date`, et vaadata rahulikult olukorda enne esimese muudatuse tegemist.

Selle pildi sees juhtub järgmine:

1. prompt on tehtud lühikeseks, et käsud oleksid paremini loetavad

2. `pwd` näitab praegust kausta
3. `ls` näitab selle kausta nähtavat sisu
4. `whoami` näitab kasutajanime
5. `date` näitab süsteemi kuupäeva ja kellaaega

2. Esimesed ohutud käsud

Need neli on head esimesed käsud, sest nad ei loo ega kustuta midagi:

```
pwd
ls
whoami
date
```

Mida need teevad

- `pwd` näitab praegust kausta
- `ls` näitab selle kausta sisu
- `whoami` näitab kasutajanime
- `date` näitab süsteemi aega

Näide

```
pwd
ls
whoami
date
```

Kui sa ei tea, mida teha edasi, siis on need neli käsku peaaegu alati hea algus.

3. Kaustade vahel liikumine

Kui esimesed vaatavad käsud on tuttavad, saad hakata liikuma ühest kaustast teise.

Süntaks

```
cd kaust
cd ..
cd ~
```

Tähendus

- `cd kaust` liigub kausta sisse
- `cd ..` liigub ühe taseme võrra üles
- `cd ~` viib kodukataloogi

Lisaks kohtad tihti ka neid kujusid:

- . tähendab praegust kausta
- .. tähendab ülemkausta
- ~ tähendab kodukataloogi

Näide

```
pwd
cd ..
pwd
cd ~
pwd
```

4. `Tab` aitab pikki nimesid lõpetada

Kui faili-, kausta- või käsunimi on pikk, ei pea seda alati lõpuni käsitsi kirjutama. Tavaliselt piisab sellest, et kirjutad nime alguse ja vajutad `Tab`.

Mida `Tab` teeb

- kui vaste on üks, lõpetab shell nime tavaliselt ise ära
- kui vasteid on mitu, lõpetab shell nime ühise osani
- kui valikuid on mitu ja neist ei piisa eristamiseks, näitab shell sageli järgmise `Tab` vajutuse järel valikuid

Näide: üks sobiv nimi

```
mkdir pikk-kaustanimi
cd pik<Tab>
pwd
```

Siin juhtub tavaliselt järgmine:

1. kirjutad `cd pik`
2. vajutad `Tab`
3. shell pakub ette kogu nime `pikk-kaustanimi`
4. vajutad `Enter` ja liigud sellesse kausta

Näide: mitu sarnast nime

```
mkdir pildid
mkdir pildid-varu
cd pil<Tab>
```

Siin ei saa shell veel üht kindlat valikut teha, sest mõlemad nimed algavad samamoodi. Tavaliselt juhtub üks neist kahest:

- shell lõpetab nime ainult ühise osani, näiteks `pildid`
- või ootab uut `Tab` vajutust ja näitab valikuid

Praktiline rusikareegel on lihtne:

- kirjuta nii palju nime algusest, kui tead
- vajuta `Tab`
- kui nimi ei saanud veel üheselt selgeks, kirjuta mõni järgmine täht juurde ja vajuta uuesti `Tab`

5. Esimene teadlik muudatus

Kui vaatavad käsud ja liikumine on juba arusaadavad, tee endale väike harjutuskaust:

```
mkdir proov
cd proov
pwd
ls
```

See on hea algus, sest:

- kaust on sinu enda alal
- saad seal rahulikult katsetada
- midagi ei lähe päris projektis kogemata segi

Kui kaust `proov` on sul juba olemas, vali lihtsalt mõni teine nimi.

6. Kuidas abi küsida

Kui käsu mõte läheb meelest, siis kõige kindlam esimene samm on:

```
man ls
```

See avab käsu manuaali. Paljud käsud toetavad ka kujusid `--help` või `-h`.

Näited:

```
man ls
ls --help
```

Oluline on meeles pidada, et `-h` ei tähenda kõigis käskudes tingimata abi. Seepärast on `man` sageli kindlam põhireegel.

7. Käsuajalugu

Shell jätab tavaliselt käsud meelde. Kõige lihtsam kuju on:

```
history
```

Alguses piisab täiesti sellest. Kui ajalugu on veel lühike, ei ole mõtet teda kohe "viimase 20" kujule lõigata.

Kasulikud lühikujud:

```
!!
!25
!ls
```

Need tähendavad:

- `!!` kordab eelmist käsku
- `!25` käivitab ajaloo kirje numbriga 25
- `!ls` käivitab viimase käsu, mis algas sõnaga `ls`

Kasulikud lisad:

- ülesnool toob eelmise käsu
- allanool liigub uuema käsu poole tagasi
- `Ctrl-r` otsib käsuajaloost

Kui kordad ajaloost käsku, mis midagi muudab, kontrolli see enne üle.

8. Esimesed kasulikud klahvid

Mõned klahvikombinatsioonid aitavad juba esimestel päevadel väga palju:

- `Ctrl-c` katkestab parajasti töötava käsu
- `Ctrl-a` liigub käsurea algusesse
- `Ctrl-e` liigub käsurea lõppu
- `Ctrl-k` kustutab kursori paremalt poolelt rea lõpu

Kui mõni programm tundub "kinni olevat" või kestab liiga kaua, siis on `Ctrl-c` esimene asi, mida proovida.

9. Vaikne käsk ei ole automaatselt vigane

Mõni käsk töötab edukalt, aga ei kuva midagi.

Näide:

```
vilo@macbook proov % touch tyhi.txt
vilo@macbook proov % cat tyhi.txt
vilo@macbook proov % ls -l tyhi.txt
-rw-r--r--  1 vilo  staff  0 Apr 13 09:21 tyhi.txt
vilo@macbook proov %
```

Siin:

- `touch tyhi.txt` loob tühja faili või uuendab olemasoleva faili ajatemplit
- `cat tyhi.txt` ei näita midagi, sest fail on tühi
- `ls -l tyhi.txt` kinnitab, et fail on olemas

Seega uus prompt ei tähenda automaatselt viga. Mõnikord tähendab see lihtsalt, et käsul ei olnud midagi ekraanile näidata.

10. Prompt võib olla eri kujuga

Need kõik võivad olla täiesti tavalised promptid:

```
$ pwd
/Users/vilo/proov
```

```
vilo@macbook proov % pwd
/Users/vilo/proov
```

```
(.venv) vilo@server:~/proov$ pwd
/home/vilo/proov
```

Kui prompt lõpeb `#`, siis oled sageli kõrgemate õigustega shellis ja pead eriti hoolikalt vaatama, mida teed.

Joonis 2: Terminali näide, kus prompt tehakse järjest lühemaks: kõigepealt on näha pikk tee, siis ainult `$` ja lõpuks lühike kuju `~/uuskaust/pildid %`.

Selle pildi mõte on järgmine:

1. esimene `pwd` näitab, et kasutaja asub kaustas `/Users/vilo/uuskaust/pildid`
2. seejärel seatakse prompt ajutiselt väga lühikeseks kujuga `$`
3. uus `pwd` näitab, et töökoht ei muutunud, muutus ainult see, kuidas prompt välja näeb
4. lõpuks seatakse prompt kujule `%~ %#`, mis näitab lühikest rada nagu `~/uuskaust/pildid %`

See on hea meeldetuletus, et prompt on ainult kuvatav liides. Tegelik asukoht tuleb endiselt käsust `pwd`.

10. Väike turvamärkus

Ära kopeeri terminali käsuridu pimesi lihtsalt sellepärast, et need näevad veebis või vestluses usaldusväärsed välja.

Eriti ettevaatlik tasub olla käskudega, mis:

- tõmbavad midagi veebist
- muudavad palju faile korraga
- käivitavad teise käsu automaatselt

Kui sa ei saa aru, mida käsk teeb, siis peata korraks töö ja loe enne abi.

Minitest

1. Käivita `pwd`, `ls`, `whoami` ja `date`.
2. Liigu `cd ..` abil ühe taseme võrra üles ja tule `cd ~` abil kodukataloogi tagasi.
3. Loo kodukataloogi alla kaust `proov` ja liigu selle sisse.
4. Käivita `history`.
5. Korda eelmist käsku kujuga `!!`.
6. Selgita ühe lausega, miks `pwd` on sageli kindlam kui prompti kuju.

Abi leidmine: man, –help ja info

Kui uus käsk ei tööta või selle mõte ei ole selge, siis esimene mõistlik reaktsioon ei ole juhuslik veebileht, vaid käsu enda abi.

Loogika

Abi otsimisel on hea liikuda selles järjekorras:

1. ava `man`, kui tahad rahulikku tervikpilti
2. proovi `--help`, kui tahad lühikest meeldetuletust
3. kasuta `whatis` või `apropos`, kui sa ei mäleta käsu nime
4. vaata `info`, kui teema on suurem GNU tööriistade kogum

See peatükk on seotud kogu ülejäänud õpikuga, sest iga uue käsu õppimine peaks algama just siit.

1. Käsu manuaal: `man`

Kõige tavalisem kuju on:

```
man ls
```

See avab käsu manuaali.

Mida seal teha saab

- `Space` liigub järgmise lehe peale
- `b` liigub tagasi
- `/muster` otsib tekstist
- `q` väljub

Näited

```
man ls
man grep
man less
```

Kui sa ei tea veel kõiki detaile, siis piisab alguses täiesti sellest, et loed manuaali alguse läbi ja otsid üles kõige tavalisemad valikud.

2. Lühike abi: `--help`

Paljud käsud toetavad lühikest abi kujul:

```
ls --help
grep --help
```

See on hea siis, kui tahad kiirelt näha:

- milliseid lippe käsk toetab
- milline on põhisüntaks
- mis järjekorras argumendid käivad

`--help` on eriti mugav siis, kui sa ei taha kohe pikka manuaali lugema minna.

3. `-h`, `--help` ja `-H` ei ole sama asi

Kõik käsud ei kasuta samu võtmeid.

- `--help` on levinud GNU stiil
- `-h` tähendab mõnes käsus abi, mõnes käsus midagi muud
- `-H` tähendab sageli hoopis teist käitumist

Seepärast ei maksa eeldada, et `-h` on alati "help".

Praktiline rusikareegel:

- proovi kõigepealt `man käsk`
- seejärel vaata `käsk --help`

4. Kui käsu nime ei mäleta

Mõnikord tead teemat, aga mitte käsku. Siis on abiks:

```
whatis ls
apropos archive
```

Vahe nende vahel

- `whatis käsk` annab ühe lühikirjelduse tuntud käsu kohta
- `apropos sõna` otsib märksõna järgi seotud käske

Näited:

```
whatis awk
apropos copy
apropos archive
```

5. GNU info-dokumendid

Mõne suurema GNU tööriistakogumi puhul kohtad ka käsku:

```
info coreutils
```

See ei ole alguses kõige tähtsam tööriist, aga hea on teada, et ta on olemas.

`info` on kõige kasulikum siis, kui:

- `man` tundub liiga lühike
- teema koosneb tervest tööriistaperest
- vajad sügavamat dokumentatsiooni

6. Väike praktiline rada

Kui sa ei mäleta, kuidas `tar` töötab, siis hea järjekord on:

```
man tar
tar --help
apropos archive
```

Siin:

1. `man tar` annab tervikpildi
2. `tar --help` näitab lühikest meeldetuletust
3. `apropos archive` aitab leida ka teisi samasse teemasse kuuluvaid käske

Minitest

1. Ava `man less`.
2. Kontrolli, kas käsk `tar` toetab kuju `--help`.
3. Leia `apropos` abil mõni pakkimisega seotud käsk.
4. Vaata käsu `ls` lühikirjeldust käsuga `whatis`.

Kataloogid ja failid

Unix-laadsetes süsteemides on failide ja kaustadega töötamine üks põhioskusi.

Loogika

Failidega töötamisel on hea hoida meeles lihtsat rütmi: leia õige koht, tee muudatus, kontrolli tulemus üle. See peatükk on seotud peaaegu kõigi teistega, sest enamik Linuxi tööst toimub lõpuks failide ja kataloogidega.

Liikumine failisüsteemis

```
pwd
ls
cd kaust
cd ..
```

Enne kui lood kaustu

Enne käsku `mkdir` tasub teha kaks lihtsat kontrolli:

```
pwd
ls
```

Need kaks küsimust on:

1. kus ma praegu olen
2. kas see on koht, kus ma tahan päriselt muudatusi teha

Alguses on hea harjutada oma kodukataloogis või selle all olevas eraldi harjutuskaustas. Nii on lihtsam vältida seda, et teed muudatusi mõnes päris projektis või süsteemi tähtsas kohas.

Väga hea praktiline nimi sellise kausta jaoks on näiteks `proov`, `harjutus` või `faili-naited`:

```
mkdir proov
cd proov
pwd
```

või:

```
mkdir faili-naited
cd faili-naited
```

Kui need nimed on sul juba olemas, vali lihtsalt mõni teine uus nimi. Hiljem näed ka kuju `~/tmp`, mis tähendab sinu kodukataloogi all olevat kausta `tmp`.

Teed: `proov`, `./proov` ja `~/proov`

Need kolm kuju on sugulased, aga mitte päris sama asi.

- `proov` tähendab kausta nimega `proov` siin samas praeguses kataloogis
- `./proov` tähendab täpselt sama, aga ütleb selle veel selgemalt välja
- `~/proov` tähendab kausta `proov` sinu kodukataloogi all

Algaja jaoks on sageli kõige lihtsam alustada kujuga `proov` või `./proov`, sest siis on seos käsuga `pwd` kohe nähtav.

Mida tähele panna

- `rm` kustutab faili ilma prügikastita
- `mv` võib nii ümber nimetada kui ka faili teise kohta liigutada
- `cd ~` viib kodukataloogi

Esimesed käsud

- `pwd` näitab, kus sa oled
- `ls` näitab, mida siin leidub. Nimi tuleb sõnast `list`.
- `cd kaust` liigutab sind teise kausta. Nimi tuleb väljendist `change directory`.
- `mkdir kaust` loob uue kausta. Nimi tuleb väljendist `make directory`.
- `touch fail.txt` loob tühja faili
- `cp allikas siht` kopeerib faili või kausta. Nimi tuleb sõnast `copy`.
- `mv vana uus` liigutab või nimetab ümber. Nimi tuleb sõnast `move`.
- `rm fail` kustutab faili. Nimi tuleb sõnast `remove`.
- `rmdir kaust` kustutab tühja kausta
- `sha256sum fail` arvutab faili räsi Linuxis
- `shasum -a 256 fail` arvutab faili räsi macOS-is
- `find . -name 'muster'` otsib faile nime järgi

Samad käsud koos lisadega

- `ls -l` näitab detailvaadet
- `ls -a` näitab ka peidetud kirjeid
- `ls -la` teeb mõlemat korraga
- `ls -A` näitab peidetuid, aga jätab `.` ja `..` välja
- `mkdir -p tee/kaust` loob ka puuduva teekonna vahekaustad
- `cp -R` kopeerib kataloogi rekursiivselt
- `rm -r kaust` kustutab kausta koos sisuga rekursiivselt
- `find . -type f` piirab otsingu tavaliste failidega
- `find . -type d` piirab otsingu kataloogidega

Käivita need käsud

See näide õpetab järgmist loogikat:

- kontrolli, kus sa oled
- loo samasse kohta uus harjutuskaust
- liigu selle sisse
- loo fail ja kopeeri see
- kontrolli tulemust

```
pwd
ls
mkdir proov
cd proov
mkdir failid
cd failid
touch esimene.txt
cp esimene.txt teine.txt
mkdir arhiiv
mv teine.txt arhiiv/
ls
ls arhiiv
```

Näide terminalis

Siin on sama loogika kujul, kus on näha prompt, käsk, väljund ja uus prompt:

```
vilo@macbook ~ % pwd
/Users/vilo
vilo@macbook ~ % ls
Desktop
Documents
Downloads
vilo@macbook ~ % mkdir proov
vilo@macbook ~ % cd proov
vilo@macbook proov % mkdir failid
vilo@macbook proov % cd failid
vilo@macbook failid % touch esimene.txt
vilo@macbook failid % cp esimene.txt teine.txt
vilo@macbook failid % mkdir arhiiv
vilo@macbook failid % mv teine.txt arhiiv/
vilo@macbook failid % ls
arhiiv
esimene.txt
vilo@macbook failid % ls arhiiv
teine.txt
vilo@macbook failid %
```

Kui `proov` on juba olemas, siis saad selle asemel lihtsalt teha:

```
cd proov
```

Hiljem, kui `~` ja `mkdir -p` on juba selged, võib sama näite kirjutada ka lühemalt. Alguses on aga parem, kui iga samm on silmaga jälgitav.

`touch` loob faili või muudab ajatemplit

`touch` on hea näide käsust, mis võib teha nähtava muudatuse ilma midagi ekraanile kirjutamata.

Oluline loogika on selline:

- kui faili veel ei ole, siis `touch fail.txt` loob tühja faili
- kui fail on juba olemas, siis `touch` ei lisa sinna sisu
- olemasoleva faili puhul uuendab `touch` faili ajatemplit, tavaliselt muutmisaega

Seepärast on see näide oluline:

```
touch tyhi.txt
cat tyhi.txt
ls -l tyhi.txt
```

Siin:

- `touch tyhi.txt` töötab edukalt
- `cat tyhi.txt` ei näita midagi, sest fail on tühi
- `ls -l tyhi.txt` kinnitab, et fail on olemas

See tähendab, et "midagi ei ilmunud ekraanile" ei ole veel viga. Mõnikord ei ole käsul lihtsalt midagi näidata.

`touch` ja `ls -t`

Kuna `touch` muudab faili ajatemplit, võib see mõjutada ka käsku `ls -t`.

- `ls -t` sorteerib failid aja järgi
- uuem fail või äsja `touch`-itud fail liigub tavaliselt ettepoole
- `ls -lt` näitab sama loogikat koos detailvaatega

Näide:

```
printf 'vana\n' > esimene.txt
sleep 1
printf 'uus\n' > teine.txt
ls -lt
touch esimene.txt
ls -lt
```

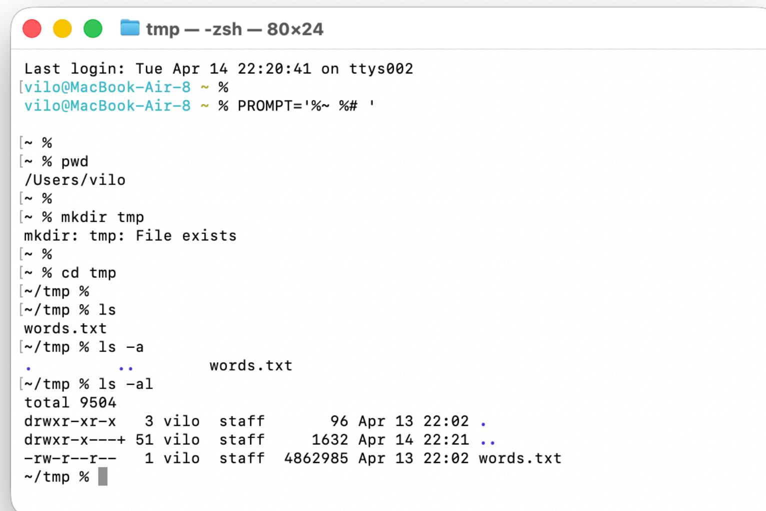
Tõenäoline tulemus on:

- enne `touch`-i on `teine.txt` nimekirjas eespool
- pärast `touch esimene.txt` võib `esimene.txt` minna ettepoole, kuigi faili sisu ei muutunud

See on hea meeldetuletus, et `touch` puudutab sageli aega, mitte sisu.

Kuvatõmmise lahtilugemine: `ls`, `ls -a` ja `ls -al`

Kuvatõmmise lugemisel ei piisa ainult pildist. Oluline on ka lahti öelda, mis seal samm-sammult juhtus.



Joonis 3: Terminali näide, kus võrreldakse käske `ls`, `ls -a` ja `ls -al` ning nähakse ka olukorda, kus `mkdir tmp` annab teate `File exists`.

Sellise akna puhul tasub tähele panna näiteks seda järjekorda:

1. prompt muudeti lühemaks, et pildil oleks vähem isiklikku infot ja rohkem ruumi käsule endale
2. `pwd` näitas, et kasutaja oli kodukataloogis, näiteks `/Users/vilo`
3. `mkdir tmp` proovis luua kausta nimega `tmp`, aga sai teate `File exists`, mis tähendab, et see kaust oli juba olemas
4. `cd tmp` liikus olemasoleva kausta sisse ja prompt muutus kujule `~/tmp %`
5. `ls` näitas ainult tavalist nähtavat sisu, selles näites faili `words.txt`
6. `ls -a` näitas lisaks ka peidetud kirjeid `.` ja `..`
7. `ls -al` näitas sama sisu pika detailvaatena koos õiguste, omaniku, suuruse ja ajaga

Olulised tähelepanekud sellest samast pildist on:

- `.` tähendab praegust kausta
- `..` tähendab ülemkausta
- `ls` ja `ls -a` ei näita sama pilti

- `mkdir` võib ebaõnnestuda täiesti mõistlikul põhjusel, näiteks siis, kui kaust on juba olemas

`rm` kustutab vaikselt

`rm` on veel üks tähtis näide käsust, mis edukal juhul sageli midagi ei kuva.

See tähendab:

- `rm fail.txt` kustutab faili
- kui kõik õnnestub, siis uut teadet ei tule
- tavaliselt näed lihtsalt järgmist käsuviipa

Näide:

```
printf 'ajutine\n' > proov.txt
rm proov.txt
ls -l proov.txt
```

Siin:

- `rm proov.txt` võib töötada täiesti vaikselt
- alles järgmine käsk näitab, et faili enam ei ole

See on seotud varasema põhireegliga: kui käsul ei ole midagi näidata, siis ta ei näita midagi, isegi siis, kui ta töötas edukalt.

Miks `rm` ei kustuta kataloogi

Tavaline `rm` on mõeldud failide eemaldamiseks.

Kui proovid kustutada kataloogi nii:

```
mkdir testkaust
rm testkaust
```

siis saad tavaliselt veateate, sest kataloog ei ole tavaline fail.

See on hea kaitsekiht: süsteem ei lase kataloogi vaikimisi sama lihtsalt eemaldada nagu üksikut faili.

Kuidas kustutada tühi kataloog

Kõige ohutum viis on:

1. kustuta vajadusel failid eraldi
2. kustuta tühi kataloog käsuga `rmdir`

Näide:

```
mkdir -p naide/arhiiv
printf 'sisu\n' > naide/arhiiv/fail.txt
```

```
rm naide/arhiiv/fail.txt
rmdir naide/arhiiv
```

`rmdir` töötab ainult siis, kui kataloog on tühi. See on algajale väga hea, sest vähendab juhusliku liigse kustutamise riski.

`rm -r` ja `rm -rf`

Kui tahad kustutada kataloogi koos sisuga, kasutatakse rekursiivset kustutamist:

```
rm -r vana-kaust
```

Oluline loogika:

- `-r` tähendab, et käsk läheb kataloogi sisse ja eemaldab selle sisu rekursiivselt
- see puudutab korraga paljusid faile ja alamkatalooge

Veel agressiivsem kuju on:

```
rm -rf vana-kaust
```

Siin:

- `-r` kustutab rekursiivselt
- `-f` tähendab `force`, ehk ära küsi kinnitust ja ära peatu väiksemate hoiatuste juures

See on põhjus, miks `rm -rf` on ohtlik:

- see võib eemaldada väga palju korraga
- tavaliselt ei ole prügikasti ega tagasivõtmise nuppu
- vale tee või vale kataloog võib teha suure kahju

Kas siin on turvavõrk

Üldreegel on: käsureal ei ole vaikimisi Finderi või Windows Exploreri moodi prügikasti.

See tähendab:

- `rm` ei vii faili prügikasti
- `rm -r` ei vii kataloogi prügikasti
- kui kustutamine õnnestub, siis failid on tavakasutuse mõttes kohe läinud

Mõnes süsteemis võib olla alias nagu `rm -i`, mis küsib üle, aga sellele ei tasu kindlalt lootma jääda. Parem harjumus on:

- kontrolli enne `pwd`
- vaata üle `ls`
- alles siis kasuta `rm`
- eelista kataloogi eemaldamisel alguses `rmdir`, kui see on võimalik

Hea algaja rusikareegel on:

- üksik fail: `rm fail.txt`
- tühi kataloog: `rmdir kaust`
- kataloog koos sisuga: `rm -r kaust`
- `rm -rf` kasuta ainult siis, kui saad täpselt aru, miks seda vajad

`ls`, `ls -a` ja `ls -la`

See on üks esimesi kohti, kus algaja näeb, et terminal ja graafiline failivaade ei näita alati täpselt sama asja.

Põhireegel on lihtne:

- `ls` näitab tavalist kaustasisu
- `ls -a` näitab ka peidetud kirjeid
- `ls -la` näitab peidetud kirjeid ja lisab detailvaate

Oluline detail:

- `-a` tähendab `all`
- see näitab ka kirjeid `.` ja `..`
- `.` tähendab praegust kataloogi
- `..` tähendab ülemkataloogi

Sageli on mugavam kasutada ka:

- `ls -A`

See näitab peidetud faile, aga jätab `.` ja `..` välja.

Näide:

```
mkdir -p ~/tmp/peidetud-naide
cd ~/tmp/peidetud-naide
touch tavaline.txt .peidetud.txt
mkdir .seaded
ls
ls -a
ls -la
ls -A
```

Tüüpiline tulemus on selline:

- `ls` näitab ainult `tavaline.txt`
- `ls -a` näitab `. .. .peidetud.txt .seaded tavaline.txt`
- `ls -la` näitab sama detailse õiguste- ja omanikuväljaga

Selle pildi lugemisel tasub jälgida seda järjekorda:

1. `ls` näitab alguses ainult nähtavat faili `tere.txt`
2. `ls -a` lisab nähtavale ka `.` ja `..`
3. `ls -al` näitab sama sisu pika detailvaatena

```
~ %
~ % cd uuskaust/pildid
~/uuskaust/pildid % ls
tere.txt
~/uuskaust/pildid % ls -a
.         ..        tere.txt
~/uuskaust/pildid % ls -al
total 8
drwxr-xr-x  3 vilo  staff   96 Apr 15 07:01 .
drwxr-xr-x  4 vilo  staff  128 Apr 15 06:43 ..
-rw-r--r--  1 vilo  staff   20 Apr 15 06:57 tere.txt
~/uuskaust/pildid % echo "näiteks seaded" > .peidetud.txt
~/uuskaust/pildid % ls -al
total 16
drwxr-xr-x  4 vilo  staff  128 Apr 15 07:05 .
drwxr-xr-x  4 vilo  staff  128 Apr 15 06:43 ..
-rw-r--r--  1 vilo  staff   16 Apr 15 07:05 .peidetud.txt
-rw-r--r--  1 vilo  staff   20 Apr 15 06:57 tere.txt
~/uuskaust/pildid % ls
tere.txt
~/uuskaust/pildid % ls -a
.              ..             .peidetud.txt tere.txt
~/uuskaust/pildid %
```

Joonis 4: Terminali näide, kus kõigepealt võrreldakse käske `ls`, `ls -a` ja `ls -al`, seejärel luuakse peidetud fail `.peidetud.txt` ja vaadatakse sama kausta sisu uuesti.

4. käsk `echo "näiteks seaded" > .peidetud.txt` loob uue peidetud faili
5. järgmine `ls` ei näita seda endiselt, sest nimi algab punktiga
6. alles `ls -a` paljastab, et `.peidetud.txt` on päriselt olemas

See pilt õpetab hästi üht tähtsat asja: peidetud fail ei ole kadunud fail. Ta on olemas, aga tavaline `ls` ei näita teda vaikimisi.

Mis need punktiga algavad failid ja kataloogid on

Punktiga algavad nimed nagu:

- `.zshrc`
- `.ssh`
- `.git`
- `.config`

on tavaliselt peidetud faili- või katalooginimed.

See ei tähenda, et need oleksid "erilised failitüübid" või kuidagi paremini kaitstud. See on eelkõige kokkulepe:

- kui nimi algab punktiga, siis paljud tööriistad ei näita seda vaikimisi
- tavaliselt hoitakse seal seadeid, metaandmeid või kasutajakeskkonna konfiguratsiooni

Hea mõtteviis on:

- tavalised töõfailid ei alga punktiga
- seadistus- ja tööriistafailid algavad sageli punktiga

Kuidas neid shellis vaadata

Shellis on kõige tavalisemad võtted:

```
ls
ls -a
ls -la
ls -A
```

Kui tahad näha ainult punktiga algavaid nimesid käesolevas kaustas, siis üks lihtne võte on:

```
find . -maxdepth 1 -name '.*'
```

Kui tahad vaadata oma kodukataloogi peidetud seadistusfaile, siis on väga tavaline:

```
cd ~
ls -la
```

Just nii näed näiteks:

- `~/.zshrc`
- `~/.ssh`
- `~/.gitconfig`

Kuidas neid Finderis vaadata macOS-is

macOS Finder peidab punktiga algavad failid ja kaustad tavaliselt ära. See on ainult kuvamisreegel, mitte failide päris kadumine.

Kõige praktilisem töövoog on:

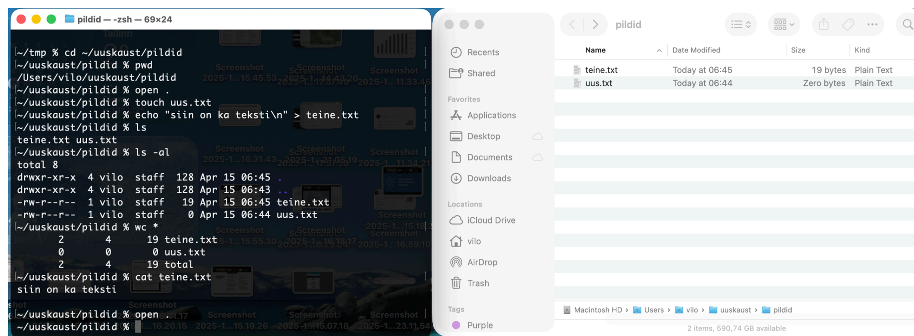
1. ava sama kaust Finderis
2. ava sama kaust Terminalis
3. tee Terminalis väike muudatus
4. vaata Finderist, mis muutus

Terminalist saad praeguse kausta Finderis avada nii:

```
open .
```

Selle pildi sees toimub samm-sammult järgmine:

1. `cd ~/uuskaust/pildid` liigub kausta, mida tahetakse korraga vaadata nii Finderis kui ka Terminalis
2. `pwd` kinnitab, et töökoht on õige
3. `open .` avab selle sama kausta Finderis



Joonis 5: Finder ja Terminal sama kausta peal: Terminalis liigutakse kausta, avatakse sama koht käsuga `open .`, luuakse tühi fail `uus.txt`, kirjutatakse faili `teine.txt` üks rida ja võrreldakse siis tulemust nii Finderis kui ka terminalis.

4. `touch uus.txt` loob tühja faili
5. `echo "siin on ka teksti\n" > teine.txt` loob teise faili ja kirjutab sinna ühe rea
6. `ls` ja `ls -al` näitavad terminalis, et mõlemad failid on olemas
7. Finderi aknas on näha seesama tulemus graafilises vaates

See on algajale väga hea harjutus, sest siin on korraga näha, et Terminal ja Finder ei tööta eri maailmades. Nad näitavad sama kausta kahe erineva kasutajaliidese kaudu.

Finderis saad peidetud failide kuvamist lülitada klahvikombinatsiooniga:

`Command-Shift-.`

See tähendab:

- kui peidetud failid olid peidus, siis need muutuvad nähtavaks
- kui need olid nähtavad, siis need peidetakse uuesti ära

See on väga hea algaja harjutus, sest siis näed otse, kuidas näiteks `.zshrc` või `.git` kaust päriselt olemas on, kuigi Finder ei näita neid alati vaikimisi.

Miks see oluline on

See teema on seotud paljude järgmiste peatükkidega:

- `~/.zshrc` ja `~/.bashrc` ilmuvad muutujate ja aliaste peatükis
- `~/.ssh` ilmub SSH peatükis
- `.git` ilmub Git-i peatükis

Kui punktiga algavate nimede loogika on selge, siis muutuvad ka need peatükid palju arusaadavamaks.

Faili sisu võrdlemine räsi abil

Mõnikord tahad teada, kas kaks faili on sisult täpselt samad. Selleks saab kasutada krüptoräsi.

Levinud näited:

- `sha256sum fail.txt`
- `shasum -a 256 fail.txt`
- `md5sum fail.txt` või `md5 fail.txt`, kui vajad ainult kiiret kontrolli

Kui kahe faili `SHA-256` räsi on sama, siis on need praktilises mõttes sama sisuga.

Näide:

```
printf 'tere\n' > a.txt
cp a.txt b.txt
sha256sum a.txt b.txt
```

Kui muudad ühe faili sisu, muutub ka räsi:

```
printf 'juurde\n' >> b.txt
sha256sum a.txt b.txt
```

Räsi kasutatakse sageli:

- failide võrdlemiseks
- allalaaditud faili tervikluse kontrolliks
- selleks, et näha, kas sisu on muutunud

Oluline märkus:

- `SHA-256` sobib hästi tervikluse kontrolliks
- `MD5` on vanem ja nõrgem, seega eelista `SHA-256`

Failide leidmine käsuga `find`

Kui `ls` näitab ainult käesoleva kausta sisu, siis `find` suudab otsida sügavamalt.

Näited:

```
find . -name '*.md'
find . -type f
find . -type d
```

```
find . -name '*.log'
```

Kui tulemused on õiged, siis alles seejärel:

```
find . -name '*.log' -delete
```

Viimane näide on võimas ja ohtlik. Enne `-delete` kasutamist tasub alati kõigepealt kontrollida, mida `find` päriselt leiab.

Kasulikud variandid:

- `find . -iname '*.jpg'` tõstutundetu nimeotsing
- `find . -maxdepth 2 -type f` piira sügavust
- `find . -size +10M` otsi suuri faile
- `find . -type f -mtime -7` otsi viimase 7 päeva jooksul muudetud faile

Veel üks väga praktiline lühikuju on:

```
ls -lt | head
```

See aitab kiiresti näha, mis siin kaustas viimati muutus.

Kui tahad suuremaid faile:

```
find . -type f -size +100M
```

Kui tahad hiljuti muudetud faile:

```
find . -type f -mtime -7
```

Need on väga head "mis siin üldse toimub?" tüüpi esimesed kontrollkäsud.

Minitest

1. Loo kaust `harjutus`.
2. Loo sinna fail `readme.txt`.
3. Tee failist koopia nimega `readme.bak`.
4. Nimeta koopia ümber nimeks `vana.txt`.
5. Loo kausta peidetud fail nimega `.salajane`.
6. Võrdle käske `ls` ja `ls -a`.
7. Arvuta kahe sama faili räsi ja võrdle tulemusi.
8. Leia `find` abil kõik `.txt` failid oma harjutuskaustast.

Teksti vaatamine ja liikumine

Selles peatükis vaatame käske nagu `cat`, `more`, `less`, `head` ja `tail`.

Loogika

Need käsud on omavahel seotud, sest nad aitavad sul enne muutmist kõigepealt sisu vaadata.

Praktiline mõtteviis on:

- lühikese faili jaoks kasuta `cat`
- pikema faili jaoks kasuta `less`
- kui fail on pikem kui korraga mugav vaadata, siis kasuta `head` või `tail`, et näha ainult algust või lõppu

See on seotud failide ja logidega töötamise loogikaga: enne mõista, siis muuda.

Kiirspikker

- `cat fail.txt` kuvab faili tervikuna
- `less fail.txt` avab faili mugavaks sirvimiseks
- `head fail.txt` näitab algust
- `tail fail.txt` näitab lõppu
- `tail -f logi.txt` jälgib faili muutumist
- `less` sees `/muster` otsib teksti
- `less` sees `78g` läheb reale 78
- `less` sees `25%` või `25p` läheb umbes veerandi peale faili sisse

Kõige tavalisemad valikud:

- `head -n 20` näita esimesed 20 rida
- `tail -n 20` näita viimased 20 rida
- `tail -f` jälgi faili juurde lisanduvat sisu

Käivita need käsud

```
seq 25 > numbrid.txt
head -n 7 numbrid.txt
tail -n 7 numbrid.txt
```

```
less numbrid.txt
```

`less` sees:

- `q` väljub
- `/tekst` otsib edasi
- `n` liigub järgmise vaste juurde
- `g` läheb faili algusse
- `G` läheb faili lõppu
- `78g` läheb reale 78
- `25%` või `25p` liigub 25% peale faili sisse

`less` sees saab hüpata rea või protsendi järgi

See on väga praktiline siis, kui fail on pikk ja sa ei taha ainult kerida, vaid minna kohe kindlasse kohta.

Näiteks:

```
seq 200 > numbrid.txt
less numbrid.txt
```

`less` sees võid kirjutada:

- `78g`, et minna reale 78
- `25%`, et minna umbes veerandi peale faili sisse
- `50%`, et minna faili keskele
- `G`, et minna faili lõppu

Loogika on:

- rea number + `g` tähendab "mine sellele reale"
- protsent + `%` või `p` tähendab "mine selle koha peale failis"

See on eriti kasulik logide, konfiguratsioonifailide ja suurte andmefailide puhul.

Millal mida kasutada

- `cat` lühikese faili jaoks
- `less` pika faili või logi jaoks
- `head` ja `tail` siis, kui fail on piisavalt pikk, et terve sisu korraga ei oleks mõistlik vaadata

Kõige sagedasem päriselu muster on:

```
tail -f app.log
```

või:

```
less /etc/passwd
```

`tail -f` logide vaatamiseks

`tail -f` on eraldi oluline juhtum, sest siin ei vaata sa ainult faili lõppu, vaid jälgid faili kasvu reaalajas.

See on seotud logide, serverite ja taustaprotsessidega:

- programm lisab faili uusi ridu
- `tail -f` näitab neid ridu kohe, kui need faili jõuavad
- vaatamine kestab seni, kuni selle katkestad

Kõige tavalisem kasutus on:

```
tail -f app.log
```

Peata jälgimine:

```
Ctrl-c
```

Praktiline harjutus on teha kaks terminaliakent:

Esimeses aknas:

```
touch app.log
tail -f app.log
```

Teises aknas:

```
printf 'server käivitus\n' >> app.log
printf 'viga: ühendus katkes\n' >> app.log
```

Siis näed kohe, kuidas `tail -f` sobib logide vaatamiseks paremini kui tavaline `cat` või ühekordne `tail -n 20`.

Minitest

1. Loo 25-realine fail `seq 25 > numbrid.txt`.
2. Vaata esimesed 7 rida käsuga `head -n 7 numbrid.txt`.
3. Vaata viimased 7 rida käsuga `tail -n 7 numbrid.txt`.
4. Ava fail `less` abil ja otsi üles number `17`.
5. Ava pikem fail `seq 200 > numbrid.txt`, sisene `less`-i ja proovi käske `78g` ning `25%`.
6. Proovi logi jälgimist käsuga `tail -f app.log` ja lisa teises terminalis faili paar rida juurde.

Failide vaatamine ja muutmine: cat, less, nano, vim

Selles peatükis keskendume tekstifailide avamisele, lugemisele ja kiirele muutmisele.

Loogika

Need tööriistad on seotud nii:

- `cat` ja `less` on peamiselt vaatamiseks
- `nano` ja `vim` on muutmiseks

Praktiliselt tähendab see, et alguses piisab täiesti sellest:

- vaata faili `less` abil
- tee kiire muudatus `nano` abil
- õpi `vim`-ist vähemalt kindlalt väljuma

Kiirspikker

- `cat fail.txt` kuvab faili
- `less fail.txt` sirvib faili
- `nano fail.txt` lihtne redaktor
- `vim fail.txt` võimas modaalne redaktor

Kõige sagedasemad `vim` käsud alguses:

- `Esc` väljub sisestusrežiimist
- `:q` väljub, kui midagi pole muutunud
- `:q!` väljub ilma salvestamata
- `:wq` salvestab ja väljub

Käivita need käsud

```
printf 'esimene\nteine\nkolmas\n' > naide.txt
cat naide.txt
```

```
less naide.txt
```

```
nano naide.txt
vim naide.txt
```

Kuidas `vim`-ist välja saada

Kõige klassikalisem:

1. vajuta `Esc`
2. kirjuta `:q` ja Enter

Kui fail on muudetud ja tahad ilma salvestamata väljuda:

```
:q!
```

Kui tahad salvestada ja väljuda:

```
:wq
```

`less` sees otsimine

- `/muster` otsib edasi
- `?muster` otsib tagasi
- `n` järgmine vaste
- `N` eelmine vaste

Kõige tavalisem töövoog

Kui pead muutma konfiguratsioonifaili, siis üks lihtne rada on:

```
less seadistus.conf
nano seadistus.conf
```

Kui töötad rohkem terminalis, liigud hiljem võib-olla `vim`-i peale.

Minitest

1. Loo tekstifail kolme reaks.
2. Ava see `nano` abil ja lisa üks rida.
3. Ava see `vim` abil ja välju failist.

Käskude kuju ja argumentide loogika

Paljud käsud näevad käsureal välja sarnased. Kui see põhimuster on selge, on hiljem palju lihtsam uusi käske õppida.

Loogika

Enamasti saab käsurea kirjutada nii:

`käsk [valikud] [argumendid]`

See tähendab:

- kõigepealt tuleb käsu nimi
- siis tulevad valikud ehk lipud
- siis tulevad argumendid ehk see, mille peal käsku kasutatakse

Näiteks:

```
ls -l
grep -n root fail.txt
cp fail.txt koopia.txt
```

1. Lühikesed ja pikad valikud

Valikud muudavad käsu käitumist.

Lühikesed valikud

```
ls -l
ls -a
ls -la
```

Siin:

- `-l` on üks lühike valik
- `-a` on teine lühike valik
- `-la` tähendab, et mõlemad pannakse kokku

Pikad valikud

Mõni käsk toetab pikemaid nimesid:

```
grep --help
grep --color=auto root fail.txt
```

Siin:

- `--help` on pikk valik ilma väärtuseta
- `--color=auto` on pikk valik koos väärtusega

Kõik käsud ei toeta samu kujusid. Mõni toetab `-h`, mõni `--help`, mõni mõlemat.

2. Argumendid

Argument on see, mille peal käsk töötab.

Näited:

```
ls /etc
cat fail.txt
cp vana.txt uus.txt
```

Siin on argumendid:

- `/etc`
- `fail.txt`
- `vana.txt` ja `uus.txt`

Praktiline rusikareegel on:

```
käsk [valikud] [argumendid]
```

See on kõige loetavam kuju ka siis, kui käsk muutub pikemaks.

3. Valikute ja argumentide järjekord

Mõne käsu puhul võib järjekord tunduda paindlik, aga alati ei tasu sellele loota.

Turvalisem on kirjutada nii:

```
grep -n root fail.txt
cp -R kaust koopia
tar -czf varu.tar.gz kaust/
```

Ehk:

1. käsu nimi
2. kõige tavalisemad valikud
3. sihtfailid või muud argumendid

4. Erimärk --

Kui faili nimi algab miinusega, võib käsk seda valikuna valesti tõlgendada.

Siis aitab `--`:

```
touch -- -imelik-fail
ls -- -imelik-fail
rm -- -imelik-fail
```

`--` tähendab siin: "siit edasi ära tõlgenda enam midagi valikuna".

5. Globbing ehk mustrid failinimedes

Shell oskab mõningaid märke tõlgendada mustritena.

Näited:

```
ls *.md
ls data-?.txt
ls pilt[12].png
```

Need tähendavad:

- `*` sobitab null või rohkem märki
- `?` sobitab täpselt ühe märgi
- `[]` sobitab ühe märgi etteantud hulgast

Oluline detail on see, et shell laiendab need mustrid enne, kui käsk ise käivitub.

Näiteks:

```
grep root *.txt
```

siin ei saa `grep` argumenti `*.txt`. Shell teeb sellest enne tegelike failinimede loendi.

6. Jutumärgid ja backslash

Kui failinimes on tühikud või erimärgid, tuleb nimi kaitsta.

Kõige tavalisemad võtted on:

```
echo '$HOME'
echo "$HOME"
echo fail\ nimega\ tühik.txt
```

Tähendus:

- `'...'` jätab teksti sõna-sõnalt
- `"..."` lubab näiteks muutuja asenduse
- `\` kaitseb ühte märki

Näited

```
mkdir "Minu Kaust"
cd "Minu Kaust"
printf 'tere\n' > "fail nimi.txt"
cat "fail nimi.txt"
```

ja:

```
touch Minu\ fail.txt
cat Minu\ fail.txt
```

Mõlemad töötavad. Pikemate nimede puhul on jutumärgid tavaliselt loetavamad.

7. Üksik- ja topeltjutumärgid ei ole sama asi

Need kaks on shellis erineva tähendusega.

```
nimi='Mari'
echo 'Tere $nimi'
echo "Tere $nimi"
```

Tulemus on põhimõtteliselt selline:

- `'Tere $nimi'` jätab teksti muutmata
- `"Tere $nimi"` asendab muutuja väärtusega

Praktiline reegel:

- kasuta `'...'`, kui tahad täiesti sõna-sõnalist teksti
- kasuta `"..."`, kui tahad säilitada ühe argumendi, aga lubada muutujate asendust

8. Failinimed tühikute ja erimärkidega

Kui failinimes on tühik, sulud, tärnid või muud erimärgid, siis shell võib nime valesti tükkideks jagada või mustrina tõlgendada.

Näited:

```
printf 'sisu\n' > "Minu fail.txt"
mv "Minu fail.txt" "Uus nimi.txt"
```

Kõige praktilisem soovitus alguses on:

1. eelista nimedes sidekriipse või alakriipse
2. kui nimes on tühik või erimärk, kasuta jutumärke

9. Kõige sagedamini korduvad lipud

Paljudes käskudes kohtad samu lühikesi märke, aga nende tähendus ei ole alati täpselt sama.

- `-h` võib tähendada abi või inimloetavat kuju
- `-v` tähendab sageli jutukamat väljundit
- `-r` või `-R` tähendab sageli rekursiivselt
- `-n` tähendab sageli arvu või rea numbrit

Näited:

```
head -n 5 fail.txt
grep -n root fail.txt
rm -r vana-kaust
```

Sama lipp ei tähenda kõigis käskudes sama asja. Seepärast tuleb iga käsu abi eraldi vaadata.

Minitest

1. Käivita `ls -la`.
2. Ava mõne käsu abi kujul `--help`.
3. Loo fail nimega `Minu fail.txt` ja kuva selle sisu.
4. Proovi käsku `ls *.md` mõnes kaustas, kus on mitu Markdown-faili.

5. Selgita ühe lausega, mida teeb `--`.

Sisend, väljund, torud ja suunamine

See peatükk seletab lahti märgid, mis panevad käsud omavahel koostööd tegema.

Loogika

Kõige tähtsam mõte on lihtne:

- käsk võib midagi ekraanile kirjutada
- selle väljundi võib suunata faili
- selle väljundi võib anda teisele käsule

Need ei ole siiski kõik sama asi. Märgid `>`, `>>`, `|`, `;`, `&&` ja `||` teevad eri tööd.

1. > kirjutab faili

Kui tahad käsu väljundi faili panna, kasuta märki `>`.

```
echo tere > sonad.txt
cat sonad.txt
```

Siin:

- `echo tere` toodab teksti
- `>` suunab selle faili
- `cat sonad.txt` näitab tulemust

Oluline reegel:

- `>` kirjutab faili uue sisu ja kirjutab vana sisu üle

2. >> lisab faili lõppu

Kui tahad olemasolevale failile juurde lisada, kasuta märki `>>`.

```
echo esimene > read.txt
echo teine >> read.txt
cat read.txt
```

Tulemus on:

```
esimene
teine
```

Siin ongi kõige tähtsam vahe:

- `>` kirjutab üle
- `>>` lisab lõppu

3. Toru `|`

Toru ei kirjuta väljundit faili. Ta saadab ühe käsu väljundi järgmise käsu sisendiks.

Näide:

```
printf 'üks\nkaks\nkolm\n' | wc -l
```

Siin:

- `printf` toodab kolm rida
- toru `|` saadab need edasi
- `wc -l` loeb kokku, mitu rida tuli

See on teistsugune loogika kui failis `>` hoidmine.

4. Järjestikused käsud: `;`

Märk `;` tähendab lihtsalt: käivita järgmine käsk pärast eelmist.

```
pwd ; ls ; date
```

Siin käivitatakse kolm käsku järjest. `;` ei anna andmeid ühest käsust teise edasi.

See on peamine vahe `;` ja `|` vahel:

- `;` käivitab käsud järjest
- `|` ühendab käsud voona

5. `&&` ja `||`

Need kaks märki lisavad käsujadale tingimuse.

`&&`

```
mkdir proov && cd proov
```

See tähendab: tee teine käsk ainult siis, kui esimene õnnestus.

`||`

```
grep 'midagi' puuduv.txt || echo 'otsing ebaõnnestus'
```

See tähendab: tee teine käsk siis, kui esimene ebaõnnestus.

Võrdlus

```
false ; echo 'see käivitus ikkagi'
false && echo 'seda ei näe'
false || echo 'varukäsk läks tööle'
```

Siin juhtub:

- `;` järel läheb järgmine käsk alati käima
- `&&` järel järgmine käsk ei käivitu, sest esimene ebaõnnestus
- `||` järel järgmine käsk käivitub just sellepärast, et esimene ebaõnnestus

6. Vead ja `2>`

Tavaline väljund ja veateated ei ole käsureal päris sama asi.

- tavaline väljund läheb tavaliselt `stdout` kaudu
- vead lähevad tavaliselt `stderr` kaudu

Kui tahad vead eraldi faili panna, kasuta:

```
ls puuduv_fail 2> vead.txt
cat vead.txt
```

Siin:

- `2>` tähendab veaväljundi suunamist
- tavaline väljund läheks endiselt ekraanile

7. `tee`

`tee` on kasulik siis, kui tahad väljundit korraga:

- näha ekraanil
- salvestada faili

Näide:

```
printf 'Tallinn\nTartu\nNarva\n' | tee linnad.txt
cat linnad.txt
```

Kui tahad faili lõppu lisada, kasuta:

```
printf 'Pärnu\n' | tee -a linnad.txt
```

8. `2>&1` ja muud keerulisemad kujud

Kui tahad ühendada veaväljundi tavalise väljundiga, kohtad kuju:

```
find . -name '*.md' > tulemused.txt 2>&1
```

See tähendab, et nii tavaline väljund kui ka vead lähevad samasse faili.

See ei ole enam kõige esimene kuju, mida pähe õppida, aga hea on teada, et selline võimalus on olemas.

9. Miks `sudo echo ... > fail` ei tööta nii, nagu algaja ootab

Paljud proovivad kuju:

```
sudo echo 'naide=1' > /etc/naide.conf
```

Aga siin teeb ümbersuunamise `>` sinu praegune shell, mitte `sudo`.

Sellepärast kasutatakse sageli hoopis sellist kuju:

```
echo 'naide=1' | sudo tee /etc/naide.conf
```

Peamine loogika on:

- `sudo echo ... > fail` ei anna root-õigust shelli suunamisele
- `sudo tee fail` avab faili protsessis, millel on vajalikud õigused

10. Kui väljund ei ilmu kohe

Mõni programm ei kirjuta iga rida kohe ekraanile või torusse edasi, vaid kogub väljundi vahepeal puhvrisse.

See tähendab:

- terminalis töötav programm võib näidata ridu kohe
- sama programm toru või faili kaudu võib väljundi edasi anda hiljem

Pythoni puhul kohtab seda sageli. Selleks on olemas näiteks:

- `python3 -u programm.py`
- `print(..., flush=True)`

See on juba natuke järgmise taseme teema, aga hea on teada, miks mõni toru "vaikib" kauem kui ootasid.

Minitest

1. Loo fail käsuga `echo tere > fail.txt`.
2. Lisa teine rida käsuga `echo maailm >> fail.txt`.
3. Näita faili sisu käsuga `cat`.
4. Ühenda kaks käsku toruga, näiteks `printf ... | wc -l`.
5. Võrdle käske `pwd ; ls` ja `pwd | ls`.
6. Proovi, kuidas `false && echo ok` erineb käsust `false || echo ok`.

Esimene tervikharjutus: 30 minutit

See peatükk ei ole enam täiesti esimene kokkupuude käsureaga. Mõte on teine: kui oled läbi vaadanud peatükid terminali põhimõtetest, failidest, abi leidmisest ja suunamisest, siis siin teed ühe lühikese tervikharjutuse algusest lõpuni läbi.

Loogika

Selle harjutuse eesmärk on siduda kokku mõned juba tuttavad mõtted:

- vaata enne, kus sa oled

- tööta eraldi harjutuskaustas
- kirjuta faili sisu väikeste sammudena
- kontrolli tulemust iga muudatuse järel

See on hea koht, kus harjutada käsurida ilma, et peaks veel korraga õppima uusi sümboleid või täiesti uusi käske.

Enne alustamist

Kui mõni käsk on meelest läinud, siis peata korraks töö ja vaata abi:

```
man pwd
man ls
man cat
```

Paljud käsud toetavad ka kujusid `--help` või `-h`, aga see ei ole kõigis süsteemides ühtlane. Kõige kindlam algus on tavaliselt `man`.

Kiirspikker

- `pwd` näitab praegust kausta
- `ls` näitab kausta sisu
- `mkdir` loob kausta
- `cd` liigub kausta sisse
- `echo ... > fail` kirjutab faili esimese rea
- `echo ... >> fail` lisab faili lõppu järgmise rea
- `cat fail` näitab faili sisu
- `wc -l fail` loeb ridu
- `cp allikas siht` teeb koopia

Harjutus

Tee see plokk rahulikult algusest lõpuni läbi:

```
pwd
ls
mkdir proov
cd proov
mkdir esimene-harjutus
cd esimene-harjutus
echo tere > sonad.txt
echo maailm >> sonad.txt
echo linux >> sonad.txt
ls
cat sonad.txt
wc -l sonad.txt
cp sonad.txt koopia.txt
```

```
ls
cat koopia.txt
```

Mida siin tehti

Harjutuses juhtus samm-sammult järgmine:

1. `pwd` ja `ls` kontrollisid alguskohta
2. `mkdir proov` ja `mkdir esimene-harjutus` lõid eraldi töökaustad
3. `cd` liikus õigesse kohta
4. `echo ... > sonad.txt` lõi faili ja kirjutas sinna esimese rea
5. `echo ... >> sonad.txt` lisas järgmised read olemasoleva faili lõppu
6. `cat sonad.txt` näitas faili sisu
7. `wc -l sonad.txt` luges kokku, mitu rida failis on
8. `cp sonad.txt koopia.txt` tegi failist koopia
9. viimane `cat koopia.txt` kinnitas, et koopia sisaldab sama teksti

Just selline tööviis on käsureal väga tavaline: tee väike samm ja kontrolli tulemust kohe.

`>` ja `>>` vahe

Selles harjutuses on kaks väga tähtsat märki:

- `>` kirjutab faili uue sisu ja kirjutab vana sisu üle
- `>>` lisab uue rea olemasoleva faili lõppu

Näiteks:

```
echo esimene > naide.txt
echo teine >> naide.txt
cat naide.txt
```

Tulemus on:

```
esimene
teine
```

Kui teha viimane rida kujul `echo teine > naide.txt`, siis vana sisu kirjutataks üle.

Väike turvamärkus

Terminalis ei tasu kunagi lihtsalt kopeerida ja käivitada käsku, mille mõtet sa ei mõista.

Eriti ettevaatlik tasub olla käskudega, mis:

- tõmbavad midagi veebist
- muudavad palju faile korraga
- käivitavad teise käsu automaatselt

Kui jääd kahtlema, peata ja loe enne abi või küsi üle. Käsurida on võimas just sellepärast, et ta teeb täpselt seda, mida sa käsid.

Kui käsk jääb "rippuma"

Kui mõni programm jääb pikalt tööle ja sa tahad selle peatada, siis esimene tavaline pääsetee on:

```
Ctrl-c
```

See ei ole iga juhtumi jaoks lahendus, aga alguses on see kõige tähtsam katkestusklahv.

Kui see tundus arusaadav

Pärast seda harjutust on hea jätkata selles järjekorras:

1. Teksti otsimine: grep ja sugulased
2. Teksti teisendamine: tr, cut, paste, column, strings
3. Vood ja tabelid: sort, uniq, wc, pr, join

Kui see tundus veel liiga kiire

Siis tasub minna tagasi ja lugeda aeglasemalt:

1. Terminali esimesed sammud
2. Abi leidmine: man, –help ja info
3. Kataloogid ja failid
4. Sisend, väljund, torud ja suunamine

Minitest

1. Tee kaust `teine-harjutus`.
2. Loo sinna fail `read.txt` kolme reaga.
3. Kontrolli käsuga `cat`, kas kõik read on olemas.
4. Kontrolli käsuga `wc -l`, kas failis on kolm rida.
5. Tee failist koopia nimega `read-koopia.txt`.
6. Muuda koopiat nii, et lisad sinna ühe rea juurde käsuga `>>`.

Linux, Unix, GNU, macOS, Windows ja shellid

Linuxi kasutamisel kohtab kiiresti mitut sarnast sõna: Linux, Unix, GNU, macOS, Windows, shell, `sh`, `bash`, `zsh`, PowerShell, WSL. Need ei tähenda päris sama asja.

Lühidalt

- Unix oli ajalooline operatsioonisüsteemide perekond ja mõtteviis.

- Linux on kernel ehk tuum, mille ümber ehitatakse süsteem.
- GNU on tööriistade kogum, mis annab paljud tuttavad käsud ja utiliidid.
- macOS on Unix-laadne süsteem, kuid kasutab mitmes kohas BSD- ja Apple'i tööriistu.
- Windows ei ole Unix-laadne, kuid selle saab WSL-i abil väga Linuxi moodi tööle panna.
- Shell on käsutõlk, mille kaudu kasutaja käske sisestab.

Loogika

Selle peatüki mõte on anda õiged mõisted enne, kui käsud lähevad detailseks. Nii on hiljem lihtsam aru saada, milline käitumine tuleb shellist, milline Linuxi süsteemist, milline GNU tööriistadest ja millised erinevused tulevad macOS-ist või Windowsist.

Et samad töövood toimiksid eri masinates võimalikult sarnaselt, on kõige olulisem eristada kolme kihti:

- operatsioonisüsteem
- käsureatööriistad
- shell

Kui need kihid on sarnased, siis on ka õpiku näited sarnasemad.

Kiirspikker

- `uname -a` näitab süsteemi infot
- `echo $SHELL` näitab sinu vaikimisi shelli
- `ps -p $$` näitab käimasolevat shelliprotsessi
- `bash --version` või `zsh --version` näitab shelli versiooni
- `command -v ls` näitab, kust käsk leitakse
- `sw_vers` näitab macOS-i versiooni
- `wsl -l -v` näitab Windowsi WSL-jaotusi ja nende versiooni

Kasuta siit ainult neid käske, mis sobivad sinu masinaga:

- `sw_vers` on macOS-i jaoks
- `wsl -l -v` on Windowsi ja WSL-i jaoks
- `uname -a`, `echo "$SHELL"` ja `ps -p $$` on Unix-laadsetes shellides kõige üldisemad näited

Käivita need käsud

```
uname -a
echo "$SHELL"
ps -p $$
```

```
bash --version
zsh --version
```

```
sw_vers
wsl -l -v
```

Miks see oluline on

Kui loed dokumentatsiooni või juhendeid, siis:

- mõni käitumine tuleb kernelist
- mõni käitumine tuleb shellist
- mõni käsk on GNU variant ja võib teistes Unix-laadsetes süsteemides erineda
- mõni erinevus tuleb sellest, kas töötad päris Linuxis, macOS-is või Windowsis

Levinud shellid

- `sh` on ajalooline ja üldine shelliliides
- `bash` on väga levinud GNU shell
- `zsh` on paindlik interaktiivne shell, mida kasutatakse palju ka macOS-is
- PowerShell on Windowsi käsukeskkond ja skriptikeel

Linux, Unix ja GNU erinevused

Praktikas öeldakse sageli "Linux", kuigi tegelik süsteem koosneb mitmest kihist:

- Linux annab kerneli
- GNU annab suure hulga käsureatööriistu
- distributsioon seob need tervikuks

Samas macOS on Unix-laadne, kuid paljud käsud ei ole seal GNU variandid. Näiteks `sed`, `grep`, `find` ja `tar` võivad käituda veidi teisiti kui Linuxis. Windows ei ole Unix-laadne, kuid WSL annab päris Linuxi kasutajaruumi, nii et enamik selle õpiku näiteid töötab seal otse.

macOS: miks see on sarnane

macOS tundub käsureal Linuxi moodi, sest:

- tal on Unix-laadne kasutajakeskkond
- seal on olemas `sh`, `bash`, `zsh`, `ssh`, `grep`, `sed`, `awk`
- failisüsteem, õigused ja torude loogika on tuttavad
- väga suur osa shelli- ja SSH-töövoost on sama

See on põhjus, miks suur osa siinsetest käsureatöövoogudest töötab macOS-is väikeste kohandustega.

macOS: mis on teisiti

macOS ei ole tavaliselt "GNU/Linux". Praktikas tähendab see:

- paketihaldur ei ole `apt` või `dnf`; kõige levinum lisapakettide tööriist on Homebrew ehk `brew`
- mitmed käsud on BSD variandid, mitte GNU variandid
- mõni lipp või vaikekäitumine erineb
- teenuste haldus ei käi `systemd` kaudu

Kõige sagedasem praktiline erinevus on see, et Linuxi juhendis toodud lipp ei pruugi macOS-is sama moodi töötada.

Kuidas saada macOS võimalikult sarnaseks Linuxiga

Algaja jaoks ei ole tavaliselt vaja macOS-i vägisi Linuxi moodi ümber ehitada.

Enamasti piisab sellest:

1. kasuta olemasolevat `zsh`-i või `bash`-i
2. õpi selgeks põhilised käsud ja nende loogika
3. paigalda Homebrew ainult siis, kui sul on päriselt vaja lisatarkvara

Mida macOS-is ei tasu vägisi samaks teha

Kõike ei ole mõtet Linuxi moodi suruda.

- Homebrew ei ole macOS-is vaikimisi sees; see on eraldi paigaldatav lisatööriist
- `brew` on macOS-is loomulik paketihaldur siis, kui sul on vaja lisatarkvara
- Finder ja rakenduste käivitamine jäävad Apple'i loogika järgi
- teenuste haldus ja süsteemikonfiguratsioon ei ole üks ühele Linuxiga

Hea eesmärk ei ole "teha macOS-ist Linux", vaid teha shelli- ja arendustöö piisavalt sarnaseks.

Kui sul tekib hiljem mõni konkreetne ühilduvusprobleem, siis lahenda see eraldi. Alguses ei ole vaja paigaldada GNU variante lihtsalt harjumuse pärast.

Windows: milline tee valida

Windowsis on kolm peamist teed:

- `WSL2`: parim valik, kui tahad selle õpiku käske kasutada võimalikult päris Linuxi moodi
- `PowerShell`: parim valik Windowsi enda halduseks
- `Git Bash`: kerge ja mugav, aga mitte täielik Linux

Kui eesmärk on "see õpik töötaks samal moel", siis soovitus on väga selge: kasuta `WSL2`.

`WSL` tähendab `Windows Subsystem for Linux`. See on Windowsi võimalus käivitada päris Linuxi kasutajaruumi omaette keskkonnas. Praktikas tähendab see, et saad Windowsi sees avada Ubuntu või muu Linuxi keskkonna ja kasutada seal tavapäraseid Linuxi käske, pakette ja shelli.

`WSL2` on uuem ja tavalisem variant. Õpiku mõttes on ta hea sellepärast, et käsud käituvad seal palju rohkem päris Linuxi moodi kui PowerShellis või vanas `cmd.exe`-s.

Kuidas saada Windows võimalikult sarnaseks Linuxiga

Kõige mõistlikum töövoog on:

1. paigalda Windows Terminal
2. paigalda WSL2
3. vali näiteks Ubuntu
4. tee käsureatöö WSL-i sees, mitte tavalises `cmd.exe`-s

Alustuseks piisab sageli sellest:

```
wsl --install
```

Pärast paigaldust kontrolli:

```
wsl -l -v
```

Ja siis mine Linuxi sisse:

```
wsl
```

Seal sees hakkavad juba tööle tavapärased Linuxi käsud:

```
uname -a
echo "$SHELL"
sudo apt update
```

Kuidas Windowsis shelliga mõelda

Kui töötad WSL-is, siis loogika on:

- Windows on host-süsteem
- WSL annab Linuxi kasutajaruumi
- shell, paketid ja käsud töötavad Linuxi loogika järgi

Kui töötad PowerShellis, siis loogika on teine:

- toru ei kanna ainult teksti, vaid objekte
- käsunimed ja lühendid on teised
- paljud selle õpiku näited ei ole üks ühele kopeeritavad

See tähendab, et "sama moodi toimima" saad Windowsis kõige paremini WSL-i abil, mitte PowerShelli Linuxiks painutades.

Praktilised soovitused Windowsi jaoks

- hoia Linuxi projektid võimalusel WSL-i kodukataloogis nagu `~/projekt`, mitte alati `/mnt/c/...`

- kasuta Windows Terminali, mitte vana konsooli
- kasuta VS Code Remote WSL töövoogu, kui arendad Windowsist Linuxi tööriistadega
- kasuta PowerShelli siis, kui haldad Windowsi ennast

Mis jääb Windowsis ikkagi teistsuguseks

- failiteed ja kettatähed on teistsugused
- `CRLF` ja `LF` võivad tekitada segadust
- `.exe`, `.bat` ja Windowsi õiguste loogika ei ole sama mis Unixis
- mõni tööriist töötab WSL-is paremini kui otse Windowsi failisüsteemis

Minitest

1. Uuri välja, millist shelli sa kasutad.
2. Võrdle käske `echo $SHELL` ja `ps -p $$`.
3. Kontrolli, kas sinu süsteemis on olemas `bash`, `zsh` ja `sh`.
4. Kui oled macOS-is, kontrolli käsuga `sw_vers`, millist süsteemi kasutad.
5. Kui oled Windowsis, kontrolli käsuga `wsl -l -v`, kas WSL on olemas.
6. Kirjelda ühe lausega, miks `WSL2` sobib Linuxi-laadse käsureatöö jaoks paremini kui ainult PowerShell.

Failisüsteemi kaart

Selles peatükis vaatame, millised kaustad Unix-laadse süsteemi juures kõige sagedamini ette tulevad ja mida nad üldjoontes tähendavad.

Loogika

Algaja jaoks on väga tavaline küsimus:

- mis vahe on `/` ja `~` vahel
- miks mõni fail on minu kodukataloogis, aga mõni `/etc` all
- miks süsteemifailidega ei tasu suvaliselt katsetada

Failisüsteemi kaart aitab siduda üksikuid käske suurema pildiga. Kui tead, mis tüüpi asjad mingis kaustas tavaliselt elavad, on ka veaotsing ja navigeerimine palju lihtsam.

Kiirspikker

- `/` on kogu failipuu juur
- `~` tähendab sinu kodukataloogi
- `/home` on Linuxis kasutajate kodukataloogide tavaline vanemkaust
- `/Users` on macOS-is kasutajate kodukataloogide tavaline vanemkaust
- `/etc` sisaldab palju süsteemi seadistusfaile
- `/usr` sisaldab palju programme ja teeke

- `/var` sisaldab muutuvat sisu nagu logid ja vaheandmed
- `/tmp` on ajutiste failide koht

Käivita need käsud

```
pwd
echo "$HOME"
cd /
ls
ls /etc | head
ls /usr | head
ls /var | head
ls -ld /tmp
```

Kui kasutad macOS-i, siis vaata ka:

```
ls /Users
```

/ ehk juur

Kõige ülemine kaust on `/`.

See ei tähenda "minu kodukataloog", vaid kogu failipuu algust. Kui kirjutad:

```
cd /
```

siis liigud süsteemi juure juurde, mitte oma isiklikku töökataloogi.

Oluline vahe:

- `cd /` viib süsteemi juure
- `cd ~` viib sinu kodukataloogi
- `cd ..` liigub ühe taseme võrra üles

Kodukataloog: ~

Kodukataloog on koht, kus tavaline kasutaja enamasti töötab.

Linuxis on see sageli midagi sellist:

```
/home/vilo
```

macOS-is sageli midagi sellist:

```
/Users/vilo
```

Sümbol `~` tähendab lühidalt sinu kodukataloogi. Näiteks:

- `~/Downloads`
- `~/.ssh`
- `~/proov`

See on põhjus, miks esimesed harjutused tasub teha just kodukataloogi all.

/etc

`/etc` sisaldab palju süsteemi seadistusi.

Sealt võib leida näiteks:

- teenuste seadistusfaile
- võrgu seadeid
- kasutajate ja gruppide infot

See ei ole hea koht algajale juhuslikeks katsetusteks. Selles kaustas muudatusi tehes tasub alati täpselt teada, mida muudad ja miks.

/usr

`/usr` sisaldab palju programme, käske, teeke ja dokumentatsiooni.

Praktiliselt võid sellest mõelda nii:

- siin on palju "süsteemi poolt pakutud tööriistu"
- sina kasutad neid sageli, aga ei muuda neid otse käsitsi

Sageli näed seal kaustu nagu:

- `/usr/bin`
- `/usr/lib`
- `/usr/share`

/var

`/var` on mõeldud muutuvate andmete jaoks.

Seal võivad olla näiteks:

- logifailid
- vahemälud
- spool-id
- teenuste töö käigus tekkivad andmed

Kui otsid, miks mingi teenus ei tööta või kuhu ruum kadus, jõuad üsna tihti just `/var` alla.

/tmp

`/tmp` on ajutiste failide koht.

See tähendab tavaliselt:

- siia pannakse lühiajaliselt vahefaile
- süsteem või programmid võivad selle sisu hiljem kustutada
- siia ei tasu panna faile, mida tahad kindlasti alles hoida

Kui tahad lihtsalt kiiresti midagi testida, võib `/tmp` olla kasulik töökoht. Kui tahad, et fail kindlasti alles jääks, kasuta pigem oma kodukataloogi.

Linux ja macOS ei ole siin täiesti samad

Raamatu loogika on Unix-laadne, aga detailides on vahe:

- Linuxis on kasutajate kodud tihti `/home`
- macOS-is on kasutajate kodud tihti `/Users`
- macOS-is on osa süsteemikaustu kaitstumad ja neid ei ole mõistlik käsitsi muuta

Seetõttu tasub mõelda mitte ainult teepärale endale, vaid ka selle rollile.

Rusikareegel

Kui mõtled "kus ma tohiksin vabalt katsetada?", siis tavaliselt:

- hea koht on sinu kodukataloog
- ettevaatlik koht on projektikaust, kus on päris töö
- halb koht juhukatsetusteks on süsteemikaust nagu `/etc` või `/usr`

Minitest

1. Seleta oma sõnadega, mis vahe on `cd /` ja `cd ~` vahel.
2. Uuri välja, mis on sinu kodukataloogi tegelik tee.
3. Vaata, kas sinu süsteemis on kasutajate kodud pigem `/home` või `/Users` all.
4. Pane kirja, milline kaust sobib ajutisteks failideks ja milline süsteemi seadistusteks.

Kettaruum ja süsteemi maht

Selles peatükis vaatame, kuidas kontrollida, kas kettal on ruumi ja millised kaustad selle ruumi ära kasutavad.

Loogika

Algajal tuleb väga kiiresti ette küsimus:

- kuhu ruum kadus
- kui suur mingi kaust on
- kas kettal on üldse veel vaba ruumi

Siin on oluline eristada kahte eri vaadet:

- faili- või kettasüsteemi üldseis
- konkreetse kausta või puu suurus

`df` ja `du` lahendavad need kaks eri küsimust.

Kiirspikker

- `df -h` näitab failisüsteemide kasutust
- `df -h .` näitab selle koha failisüsteemi, kus sa parajasti oled
- `du -sh .` näitab praeguse kausta kogusuurust
- `du -sh *` näitab alamkaustade ja failide suurusi
- `du -sh * | sort -h` sorteerib väiksemast suuremani
- `du -a . | sort -nr | less` näitab suurimaid kirjeid detailsemalt

Käivita need käsud

```
df -h
df -h .
du -sh .
du -sh * 2>/dev/null
du -sh * 2>/dev/null | sort -h
```

Kui tahad näha palju detailsemalt, kuhu ruum kaob, siis väga praktiline on:

```
du -a . 2>/dev/null | sort -nr | less
```

Kui tahad vaadata oma kodukataloogi tüüpilisi "ruumisööjaid", siis proovi:

```
cd "$HOME"
du -sh Downloads Documents Desktop 2>/dev/null
```

`df`: kui palju ruumi failisüsteemil on

`df` vastab küsimusele:

"Kui täis see kettasüsteem või mount point on?"

Näide:

```
df -h
```

Lipp `-h` tähendab inimesele loetavat vormi, näiteks:

- `K`
- `M`
- `G`
- `T`

Kui tahad teada just selle koha tausta, kus sa töötad, siis on väga praktiline:

```
df -h .
```

See on hea nipp, sest mitme mount point'i puhul ei huvita sind alati kogu masin, vaid just see failisüsteem, mille peal su töökaust asub.

`du`: kui suur see kaust ise on

`du` vastab küsimusele:

"Kui palju ruumi see kaust või failipuu kasutab?"

Näited:

```
du -sh .
du -sh *
```

Siin tähendab:

- `-s` summary ehk kogu kokkuvõte
- `-h` inimesele loetav suurus

See kombinatsioon on üks praktilisemaid ruumikontrolle kogu käsureal.

Kui tahad ainult ühte kogusummat, siis sobib hästi:

```
du -sh .
```

Kui tahad näha ka faili- ja alamkaustade taset detailsemalt, siis:

```
du -a .
```

Siin tähendab:

- `-a` näita mitte ainult kaustu, vaid ka üksikfaile

Just see teeb `du`-st väga praktilise veaotsingutööriista siis, kui ruum on "kuskile ära kadunud".

`df` ja `du` ei vasta samale küsimusele

Oluline vahe:

- `df` räägib failisüsteemi tasemest
- `du` räägib faili või kausta tasemest

Näiteks:

- `df -h` võib näidata, et kettal on alles ainult 5% vaba ruumi
- `du -sh Downloads` võib näidata, et suure osa sellest võtab `Downloads`

Kõige tavalisem töövoog

Kui ruum tundub otsas olevat, siis liigu nii:

1. vaata `df -h`, kas probleem on päriselt ruumis
2. mine kausta, kus arvad suure sisu olevat
3. käivita `du -sh *`
4. sorteeri tulemus, et näha suurimaid kohti

Näide:

```
cd "$HOME"
du -sh * 2>/dev/null | sort -h
```

Sellest on tihti kohe näha, kas ruumi söövad näiteks:

- allalaadimised
- vana projektikaust
- andmefailid
- buildi väljundid

Kui sellest ei piisa ja tahad minna sügavamale, siis järgmine väga tavaline päriselu käik on:

```
du -a . 2>/dev/null | sort -nr | less
```

Selle loogika on:

- `du -a .` käib kogu puu läbi ja annab iga faili või kausta suuruse
- `sort -nr` paneb suurimad kirjed ette
- `less` lubab tulemust rahulikult sirvida ja otsida

See on üks kõige praktilisemaid "mis siin kõige rohkem ruumi võtab?" käsujadasid kogu Unix-laadses käsureas.

`du -a | sort -nr | less` päris töös

See kuju väärib eraldi väljaütlemist, sest seda kasutatakse palju rohkem kui alguses arvata võiks.

Näide:

```
cd "$HOME"
du -a . 2>/dev/null | sort -nr | less
```

Mida siin vaadata:

- kõige ülemised read on tavaliselt suurimad ruumisööjad
- kui näed mõnd kausta, mine sinna sisse ja korda sama käsku kitsamas kohas
- `less` sees saad otsida näiteks `/Downloads`, `/node_modules`, `/.git`, `/.venv`

See on hea iteratiivne tööviis:

1. alusta suuremast kohast
2. leia suurim alamkaust või fail
3. mine sinna sisse
4. korda sama analüüsi

Inimloetav vs toores numbriline sort

Kõige universaalsem kuju on:

```
du -a . 2>/dev/null | sort -nr | less
```

Seda on hea kasutada, sest:

- `sort -n` töötab numbrite peal selgelt
- `du` annab suurused ühes sisemises ühikus
- tulemus on hästi võrreldav

Mõnes keskkonnas meeldib inimestele rohkem ka inimloetav kuju:

```
du -ah . 2>/dev/null | sort -hr | less
```

Aga see eeldab, et sinu `sort` toetab `-h` lippu. Selle õpiku põhikuju on seetõttu pigem `du -a . | sort -nr | less`.

Peidetud kaustad võivad olla kõige suuremad

Väga sageli söövad ruumi just peidetud kaustad, mida tavaline `ls` kohe silma ette ei too, näiteks:

- `.git`
- `.venv`
- `.cache`
- `.npm`

Selle jaoks on väga praktiline:

```
du -sh .[!.]* * 2>/dev/null | sort -h
```

See näitab koos:

- peidetud kirjed
- tavalised kirjed

Kui ruum "justkui kadus ära", siis see on üks esimesi käske, mida tasub proovida.

Ettevaatus

Kõiki suuri käske ei tasu pimesi käivitada suvalises kohas.

Näiteks:

```
du -sh /
```

võib olla aeglane ja mitte eriti informatiivne, eriti kui sul ei ole õigusi kõiki kaustu lugeda.

Praktilisem on alustada kitsamalt:

- `du -sh .`
- `du -sh *`
- `du -sh "$HOME"/*`

Kas failide arv ja suurus on sama asi?

Ei ole.

- üks väga suur fail võib võtta rohkem ruumi kui tuhat väikest
- samas tuhanded väikesed failid võivad teha töö aeglaseks või segaseks

Kui tahad lisaks suurusele aru saada ka failide hulgast, siis saad vaadata näiteks:

```
find . -type f | wc -l
```

See ei mõõda ruumi, aga aitab näha, kas probleem võib olla ka väga suures failide arvus.

Minitest

1. Uuri välja, kui palju vaba ruumi on failisüsteemil, kus asub sinu praegune töökaust.
2. Vaata, kui suur on sinu praegune kaust kokku.
3. Sorteeri praeguse kausta alamkaustad suuruse järgi.
4. Pane ühe lausega kirja, mis vahe on `df -h` ja `du -sh .` vahel.

Õigused, omanikud ja täitmisbitid

Unix-laadsetes süsteemides on igal failil omanik, grupp ja õigused.

Loogika

Õigused määravad, kes võib faili lugeda, muuta või käivitada. See on seotud kasutajate, gruppide, `sudo` ja shelliskriptidega, sest kõik need teemad sõltuvad õiguste korrektsest mõistmisest.

Kiirspikker

- `ls -l` vaata õigusi
- `chmod +x fail` tee fail käivitatavaks
- `chmod 644 fail` tavaline tekstifail
- `chmod 755 fail` tavaline käivitatav fail
- `chown kasutaja:grupp fail` muuda omanikku

Õiguste vaatamine

```
ls -l
```

Näites:

```
-rw-r--r-- 1 mari users 120 Apr 12 10:00 naide.txt
```

See rida kirjeldab:

- faili tüüpi
- omaniku õigusi
- grupi õigusi
- teiste kasutajate õigusi

Õiguste muutmine

```
chmod u+x skript.sh
chmod 644 naide.txt
```

Kõige tavalisemad lipud

- `ls -l` kuva õigused detailvaates
- `chmod +x` lisa täitmisõigus
- `chmod 644` sea tavalise tekstifaili õigused
- `chmod 755` sea tavalise käivitatava faili õigused
- `chown kasutaja:grupp` muuda omanikku ja gruppi

Omaniku muutmine

```
sudo chown kasutaja:grupp fail.txt
```

Käivitatavaks tegemine

`chmod +x` ja shebang-rida käivad sageli koos.

- shebang nagu `#!/bin/sh` või `#!/usr/bin/env python3` ütleb, millise interpretaatoriga faili käivitada
- täitmisõigus ütleb, et faili tohib käivitada käsuga `./fail`
- kui käivitad faili kujul `./fail`, siis süsteem vaatab kõigepealt faili algust ja otsib sealt, millega seda tõlgendada

Kui üks neist puudub, siis võib fail küll olemas olla, aga ta ei käivitu ootuspäraselt.

Käivita need käsud

```
printf '#!/bin/sh\necho tere\n' > tere.sh
chmod +x tere.sh
./tere.sh
```

```
printf '#!/usr/bin/env perl\nprint \"tere\\n\";\n' > tere.pl
chmod +x tere.pl
./tere.pl
```

Mida tähendab käivitatav fail

Täidetavaks tegemine ei muuda faili maagiliselt programmiks. Tavaliselt on vaja:

1. õiget shebang-rida
2. täitmisõigust
3. olemasolevat interpretaatorit või binaari

Kui fail algab näiteks nii:

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

siis süsteem proovib selle käivitada Perliga. Kui Perl puudub või tee on vale, siis ei piisa ainult täitmisõigusest.

Minitest

1. Tee fail `proov.sh`, mis väljastab ühe rea.
2. Anna talle täitmisõigus.
3. Käivita fail nii `sh proov.sh` kui `./proov.sh`.

Kasutajad, grupid ja sudo

Selles peatükis räägime kasutajatest, gruppidest, `sudo` kasutamisest ja riskidest.

Loogika

Unix-laadsed süsteemid eeldavad, et igal tegevusel on tegija. Seetõttu on oluline aru saada, millise kasutajana sa töötad, millistesse gruppidesse kuulud ja millal on vaja kõrgemaid õigusi.

Kiirspikker

- `whoami` näitab aktiivset kasutajat
- `id` näitab kasutajat ja gruppe
- `groups` näitab gruppe
- `sudo käsk` käivitab käsu kõrgemate õigustega
- `su - kasutaja` vahetab kasutajat

Käivita need käsud

```
whoami
id
groups
```

```
sudo -l
```

Miks ettevaatlik olla

`sudo` annab suure võimu. Vale käsk kõrgemate õigustega võib:

- kustutada süsteemifaile

- muuta õigusi valesti
- paigaldada või eemaldada tarkvara

Seega tasub enne `sudo` käivitamist mõelda, kas seda on päriselt vaja.

Mis on `root`

`root` on Unix-laadsete süsteemide eriline administraatori kasutaja.

Oluline loogika:

- tavakasutaja töötab piiratud õigustega
- `root` võib peaaegu kõike
- `sudo` annab tavakasutajale võimaluse käivitada mõni üksik käsk ajutiselt kõrgemate õigustega

Seega ei ole `root` ja `sudo` päris sama asi:

- `root` on kasutaja
- `sudo` on tööriist, millega mõni käsk käivitatakse kõrgemate õigustega

Kuidas `root` promptis ära tunda

Sageli on `root`-i promptis lõpus märk `#`, samas kui tavakasutajal on sageli `$` või `%`.

Näited:

```
vilo@macbook proov % whoami
vilo
```

```
root@server:/etc# whoami
root
```

See ei ole küll absoluutne reegel, kuid väga levinud rusikareegel on:

- `$` või `%` tähendab tavakasutajat
- `#` tähendab, et tasub olla eriti ettevaatlik

Kui on kahtlus, kontrolli alati:

```
whoami
id
```

`sudo käsk` vs `sudo -i` vs `su -`

Kõige ohutum tavakasutus on enamasti:

```
sudo käsk
```

Näiteks:

```
sudo apt update
sudo systemctl restart ssh
```

See on hea, sest kõrgemad õigused kehtivad ainult sellele ühele käsule.

On olemas ka kujud:

```
sudo -i
su -
```

Need annavad sulle root-shelli või vahetavad kasutajat. See tähendab, et järgmised käsud töötavad juba kõrgemate õigustega seni, kuni sellest shellist väljud.

Algajale on see riskantsem, sest:

- prompt võib muutuda
- iga järgmine käsk võib teha suurema mõjuga muudatuse
- on lihtne unustada, et oled enam mitte tavakasutaja

Seepärast on hea algreegel:

- eelista `sudo käsk`
- väldi püsivat root-shelli, kui selleks ei ole selget põhjust

Mida `root`-ina mitte teha

Hea ettevaatusreegel on: ära tee harjutusi, failide sorteerimist ega igapäevast tekstitööd root-kasutajana.

Väldi eriti:

- `rm -rf` käske root'ina
- failide juhuslikku `chown` või `chmod` muutmist süsteemikaustades
- harjumust "kui ei tööta, siis pane `sudo` ette"

Parem mõtteviis on:

1. proovi käsku tavakasutajana
2. loe veateadet
3. mõtle, kas probleem on õigustes või hoopis milleski muus
4. kasuta `sudo` ainult siis, kui tead, miks seda vaja on

Väike praktiline kontroll

Kui tahad näha, millal oled tavakasutaja ja millal kõrgemates õigustes, siis need käsud on kõige kasulikumad:

```
whoami
id
sudo -l
```

Need annavad rohkem kindlust kui ainult prompti kuju vaatamine.

Minitest

1. Vaata oma kasutaja gruppe.
2. Uuri, milliseid käske tohib sinu kasutaja `sudo` abil käivitada.
3. Selgita oma sõnadega, miks `sudo rm -rf ...` on ohtlik.
4. Selgita ühe lausega, mis vahe on `root` kasutajal ja käsul `sudo`.

Muutujad, keskkond, PATH ja aliased

Selles peatükis vaatame shelli muutujaid, keskkonnamuutujaid, `PATH`-i, alias'eid ja prompti seadistamist.

Kiirspikker

- `nimi=väärtus` loob shellimuutuja
- `export NIMI=väärtus` loob keskkonnamuutuja
- `echo "$PATH"` näitab otsinguteed
- `alias ll='ls -la'` loob aliase
- `alias h='history | tail -n 20'` loob ajaloo lühialiase
- `env` näitab keskkonda

Levinud kontrollkäsud

- `echo "$PATH"` kuva otsingutee
- `command -v python3` leia, kust konkreetne käsk leitakse
- `type cd` näita, mis liiki käsk see on
- `type -a grep` näita kõik samanimelised vasted
- `alias ll='ls -lah'` loo lühinimi detailsele `ls`-ile
- `alias h='history | tail -n 20'` loo lühinimi ajaloo vaatamiseks
- `source ~/.zshrc` loe shelli seadistusfail uuesti sisse
- `env | grep NIMI` kontrolli, kas keskkonnamuutuja on olemas

Loogika

Selles peatükis on kolm eri asja, mida ei tohi segamini ajada:

1. shellimuutujad
2. keskkonnamuutujad
3. alias'ed

Nende rollid on erinevad:

- muutuja hoiab väärtust
- keskkonnamuutuja antakse edasi alamprotsessidele
- alias asendab lühinime teise käsuga

Selles peatükis lisandub veel üks väga tähtis vahe:

- osa “käske” on shelli enda sees
- osa on välised programmid failisüsteemis

Käivita need käsud

```
nimi='Mari'
echo "$nimi"
export PROOV='tere'
env | grep PROOV
```

```
echo "$PATH"
command -v python3
type cd
alias ll='ls -la'
ll
```

Shellimuutuja ja keskkonnamuutuja

Shellimuutuja elab tavaliselt ainult jooksvas shellis. Keskkonnamuutuja eksporditakse edasi alamprotsessidele.

Näide:

```
nimi='Mari'
export PROJEKT='linux-opik'
sh -c 'echo "$nimi / $PROJEKT"'
```

Siin alam-shell näeb `PROJEKT` muutujat, aga mitte tingimata muutujat `nimi`.

See on kõige olulisem vahe, mida alguses meeles pidada.

Mis asi “käsk” üldse on

Terminalis kirjutatud nimi ei tähenda alati üht ja sama tüüpi asja.

Praktikas võib see olla näiteks:

- alias
- shelli funktsioon
- shelli sisseehitatud käsk
- shelli reserveeritud sõna
- väline programm kettal

Näited:

- `cd` on tavaliselt shelli sisseehitatud käsk
- `if`, `for`, `do`, `done` on shelli reserveeritud sõnad
- `grep`, `sed`, `awk`, `python3` on tavaliselt välised programmid
- `ll` võib olla alias

See on tähtis, sest:

- sisseehitatud käske ei otsita alati `PATH`-ist
- reserveeritud sõnad ei ole üldse "programmid"
- alias võib sama nimega välise käsu varjutada

Kuidas kontrollida, mis tüüpi käsk sul ees on

Kõige praktilisem käsk selle jaoks on:

```
type cd
type if
type grep
type ll
```

Tüüpilised tulemused võivad olla:

```
cd is a shell builtin
if is a shell keyword
grep is /usr/bin/grep
ll is an alias for ls -lah
```

Kui tahad lihtsalt teada, mida shell jooksutaks, on hea ka:

```
command -v grep
command -v cd
command -v ll
```

Praktiline rusikareegel:

- `type` on parem õppimiseks, sest ta ütleb ka käsu liigi
- `command -v` on hea skriptides ja lühemaks kontrolliks

Kui tahad näha kõiki samanimelisi vasteid, kasuta:

```
type -a python3
type -a grep
```

See on eriti kasulik siis, kui süsteemis on mitu sama nimega käsku.

`PATH` loogika

Kui kirjutad terminali:

```
python3
```

siis shell ei tea automaatselt, kus see fail asub. Ta otsib seda kataloogidest, mis on kirjas muutujas `PATH`.

Seetõttu on need kaks käsku väga praktilised:

```
echo "$PATH"
command -v python3
```

Kui käsk ei leidu, on probleem sageli just selles:

- programm pole paigaldatud
- selle asukoht pole `PATH`-is

Kuidas sama nimega käske otsitakse

Kui nimi ei ole alias, funktsioon, sisseehitatud käsk ega reserveeritud sõna, siis hakkab shell otsima väliseid programme mööda `PATH`-i.

Oluline loogika:

- `PATH` on kataloogide nimekiri
- neid vaadatakse vasakult paremale
- esimene sobiv fail võidab

Näide:

```
echo "$PATH"
```

võib anda midagi sellist:

```
/Users/vilo/bin:/usr/local/bin:/usr/bin:/bin
```

Kui samanimeline programm on:

- `/Users/vilo/bin/grep`
- `/usr/bin/grep`

siis leitakse esimesena `/Users/vilo/bin/grep`, sest see kataloog on `PATH`-is eespool.

See on põhjus, miks `PATH` järjekord loeb.

Kuidas panna oma `bin` ette või taha

Väga tavalised kohad enda tööriistade jaoks on:

- `~/bin`
- `~/.local/bin`

Kui tahad, et sealt otsitaks esimesena, lisa see `PATH`-i ette:

```
export PATH="$HOME/bin:$PATH"
```

või:

```
export PATH="$HOME/.local/bin:$PATH"
```

Kui tahad, et sealt otsitaks alles hiljem, lisa see lõppu:

```
export PATH="$PATH:$HOME/bin"
```

See tähendab:

- ette lisamine annab sinu tööriistadele eelisõiguse
- lõppu lisamine jätab süsteemi vaikevahendid ettepoole

Hea praktiline soovitus on:

- kui tead, et tahad omaenda skripte eelistada, lisa `~/bin` ettepoole
- kui tahad olla ettevaatlikum, lisa see lõppu

Näide oma käsu lisamisest `PATH`-i

```
mkdir -p "$HOME/bin"
cat > "$HOME/bin/tere" <<'EOF'
#!/bin/sh
echo "Tere oma bin-kataloogist"
EOF
chmod +x "$HOME/bin/tere"
export PATH="$HOME/bin:$PATH"
command -v tere
tere
```

Siin juhtub:

- lood oma väikese käsu
- teed selle käivitatavaks
- lisad `~/bin` otsingutee ette
- shell leiab nüüd käsu nime järgi üles

Kui tahad seda püsivaks teha, lisa sama `export PATH=...` rida:

- `~/.zshrc`, kui kasutad `zsh`
- `~/.bashrc`, kui kasutad `bash`

`.zshrc`, `.zprofile`, `.bashrc` ja `.bash_profile`

See on üks suuremaid segaduskohti shelli seadistamisel.

Praktiline jaotus on tavaliselt selline:

- `~/.zshrc` sinna pane alias'ed, prompt ja muud interaktiivse shelli mugavused
- `~/.zprofile` sinna pane login-shelli algseadistused, näiteks osa `PATH`-ist
- `~/.bashrc` sinna pane alias'ed ja interaktiivse bash'i seadistused
- `~/.bash_profile` seda loeb bash login-shellis; tihti pannakse see omakorda `~/.bashrc` sisse lugema

Hea rusikareegel:

- alias'ed ja prompt lähevad enamasti `~/.zshrc` või `~/.bashrc`
- `PATH` ja muud sessiooni algseadistused lähevad sageli `~/.zprofile` või `~/.bash_profile`

Näiteks zsh puhul:

```
cat >> ~/.zprofile <<'EOF'
export PATH="$HOME/bin:$PATH"
EOF
```

```
cat >> ~/.zshrc <<'EOF'
alias ll='ls -lah'
alias h='history | tail -n 20'
EOF
```

Kui mõni alias või `PATH`-i muudatus "mõnikord töötab ja mõnikord mitte", siis põhjus on väga sageli just selles, et rida sattus valesse startup-faili.

Miks `which` võib olla petlik

Mõnes süsteemis kasutatakse ka käsku:

```
which grep
```

Aga õppimise mõttes on parem eelistada:

```
type grep
command -v grep
```

Põhjus on lihtne:

- `which` keskendub rohkem välistele käskudele
- `type` ja `command -v` räägivad paremini shelli enda vaatenurgast
- aliaste ja sisseehitatud käskude puhul on `type` tavaliselt õpetlikum

Värvid või mitte

Mõned keskkonnad kasutavad värve vaikimisi, teised mitte. Näiteks:

```
ls --color=auto
grep --color=auto root /etc/passwd
```

Värvid võivad aidata, kuid skriptides ei tasu neid alati eeldada.

Alias'ed

Alias on lühike mugav nimi sagedase käsu jaoks.

Mõistlikud alias'ed on:

```
alias ll='ls -lah'
alias la='ls -A'
alias l='ls -CF'
alias h='history | tail -n 20'
alias gs='git status'
alias gp='git pull'
alias v='vim'
```

Need on head just sellepärast, et:

- nad lühendavad sageli kasutatavaid käske
- nad ei varja liiga palju loogikat
- neid on lihtne meeles pidada

Alias `h` on praktiline just sellepärast, et ta näitab kiirelt viimaseid ajalookirjeid ilma, et peaks kogu ajalugu läbi kerima.

Väldi alias'eid, mis muudavad ohtlikke käske liiga "maagiliseks".

Kuhu alias'ed panna

Tavaliselt:

- `~/.zshrc`, kui kasutad `zsh`
- `~/.bashrc`, kui kasutad `bash`

Siin tähendab `~` sinu kodukataloogi lühikuju.

Näide:

```
cat >> ~/.zshrc <<'EOF'
alias ll='ls -lah'
alias la='ls -A'
alias h='history | tail -n 20'
alias gs='git status'
alias gp='git pull'
EOF
```

Pärast faili muutmist lae seadistus uuesti sisse:

```
source ~/.zshrc
```

või:

```
source ~/.bashrc
```

Ajalugu shellis

Paljud shellid lubavad ajalugu mugavalt kasutada ja isegi seadistada.

Varases algusosas piisab täiesti käsust `history`. Alles pärast torude ja filtrite selgeks saamist muutub loomulikuks ka lühem vaade, kus ajaloost näidatakse ainult viimaseid ridu.

Näiteks:

```
history
history | tail -n 20
echo $HISTFILE
echo $HISTSIZE
```

Kui tahad ajaloo jaoks lühikest mugavat alias't, siis üks hea ja arusaadav variant on:

```
alias h='history | tail -n 20'
```

See on hea just sellepärast, et:

- `history` näitab shelli käskude ajalugu
- `tail -n 20` lõikab sellest välja viimased 20 rida
- sama kuju on algajale kergem lugeda kui shellispetsiifiline `fc`

Mõnes shellis kohtab ka kujusid nagu `history 20`, `history -20` või `fc -l -20`, kuid neid ei tasu õpiku põhikuju näitena eelistada, sest need ei käitu kõikjal ühtemoodi.

Mõnes shellis saab seadistada:

- mitu käsku meelde jäetakse
- kas duplikaadid eemaldatakse
- kas ajalugu kirjutatakse faili kohe või alles shelli sulgemisel

See on eriti kasulik `bash`-i ja `zsh`-i kasutamisel.

Prompti eri kujud

Prompt on tekst, mida näed enne käsu sisestamist. See ei ole käsu osa, vaid shelli kasutajaliidese osa.

Praktikas võib prompt näidata:

- kasutajanime
- masina nime
- praegust kataloogi
- aktiivset virtuaalkeskkonda
- seda, kas oled tavakasutaja või `root`

Sama asukoht võib eri promptidega välja näha näiteks nii:

```
$ pwd
/Users/vilo/proov
```

```
% pwd
/Users/vilo/proov
```

```
vilo@macbook proov % pwd
/Users/vilo/proov
```

```
vilo@server:~/proov$ pwd
/home/vilo/proov
```

```
(.venv) vilo@macbook proov % pwd
/Users/vilo/proov
```

Oluline reegel on:

- prompti kuju on mugavus
- `pwd` on kindel kontroll

Ajutised prompti näited bashi jaoks

Neid tasub alguses proovida ajutiselt jooksvas shellis, mitte kohe config-faili kirjutada.

Väga lihtne prompt:

```
export PS1='$ '
```

Kasutaja, host ja täielik tee:

```
export PS1='\u@\h \w \$ '
```

Ainult viimase kataloogi nimi:

```
export PS1='\u@\h \W \$ '
```

Koos kellaajaga:

```
export PS1='[\A] \u@\h \W \$ '
```

Siin:

- `\u` on kasutajanimi
- `\h` on hosti lühinimi
- `\w` on täielikum tee
- `\W` on ainult viimase kataloogi nimi

Ajutised prompti näited zsh jaoks

`zsh` kasutab promptis veidi teisi lühikoode.

Väga lihtne prompt:

```
export PROMPT='%# '
```

Kasutaja, host ja kodukataloogi suhteline tee:

```
export PROMPT='%n@%m %~ %# '
```

Ainult viimase kataloogi nimi:

```
export PROMPT='%n@%m %1~ %# '
```

Virtuaalkeskkonna ja git-haru asemel me siin veel midagi automaatselt juurde ei lisa, aga just sellise loogika peale need keerukamad promptid ehitataksegi.

Siin:

- `%n` on kasutajanimi
- `%m` on hosti lühinimi
- `%~` on tee kodukataloogi suhtes

- `%1~` on ainult viimane olulisem teelõik
- `%#` annab tavakasutajale `%` ja root'ile `#`

Värvilised promptid

Värv on ainult visuaalne abi. See ei muuda käsu sisu, aga võib aidata kiiremini aru saada, kus sa oled.

Lihtne värviline prompt `zsh` jaoks:

```
export PROMPT='%F{blue}%n@%m%f %F{green}%1~%f %# '
```

Lihtne värviline prompt `bash` jaoks:

```
export PS1='\[\e[34m\]\u@\h\[\e[0m\] \[\e[32m\]\W\[\e[0m\] \$ '
```

Kui tahad lihtsalt katsetada:

1. käivita üks prompti näide
2. tee `pwd`
3. liigu teise kausta
4. võrdle, mida prompt näitab ja mida `pwd` kinnitab

See on parim viis prompti loogikast aru saada.

Mõned praktilised keskkonnamuutujad

Need tulevad väga sageli ette:

- `PATH` kust käske otsida
- `HOME` kasutaja kodukataloog
- `EDITOR` vaikimisi tekstiredaktor
- `SHELL` kasutatav shell

Näited:

```
echo "$HOME"
echo "$SHELL"
export EDITOR=vim
```

Minitest

1. Loo alias `la='ls -la'`.
2. Loo alias `h='history | tail -n 20'`.
3. Vaata oma `PATH` muutujat.
4. Ekspordi muutuja `DEMO=1` ja kontrolli seda käsuga `env`.
5. Vaata, kuhu sinu shell käsuajalugu salvestab.
6. Lisa alias `ll='ls -lah'` oma shelli config-faili ja lae see uuesti sisse.

Paketihaldus: apt, dnf, pacman, brew

Linuxis ja macOS-is paigaldatakse tarkvara enamasti paketihalduri abil.

Loogika

Paketihaldur aitab tarkvara paigaldada nii, et sõltuvused ja versioonid püsiksid hallatavad.

See peatükk on seotud:

- süsteemi seadistamisega
- `venv`, `pip` ja `npm` teemadega
- arenduskeskkonna ülesseadmisega

Kiirspikker

- `apt install pakett` paigaldab paketi Debiani või Ubuntu süsteemis
- `dnf install pakett` paigaldab paketi Fedoras
- `pacman -S pakett` paigaldab paketi Arch Linuxis
- `brew install pakett` paigaldab paketi macOS-is või Homebrew Linuxis
- `python3 -m pip install pakett` paigaldab Pythoni paketi
- `npm install` paigaldab projekti JavaScripti sõltuvused

Levinud paketihaldurid

- `apt` Debianis ja Ubuntus
- `dnf` Fedoras
- `pacman` Arch Linuxis
- `brew` macOS-is ja mõnikord ka Linuxis

Süsteemi paketihaldur vs keele paketihaldur

Oluline on eristada kaht taset:

- süsteemi paketihaldur paigaldab tööriistu ja teeke operatsioonisüsteemi tasemel
- keele paketihaldur paigaldab sõltuvusi konkreetse programmeerimiskeele ökosüsteemis

Näited:

- `apt`, `dnf`, `pacman`, `brew` on süsteemi- või kasutajataseme paketihaldurid
- `pip` haldab Pythoni pakette
- `npm` haldab JavaScripti ja Node.js pakette

See tähendab, et `pip install requests` ei ole sama asi mis `apt install python3-requests`, kuigi mõlemad võivad puudutada Pythonit.

Praktiline Homebrew baas macOS-is

Kui teed selle õpiku tööriistad macOS-is kiiresti valmis, siis üks mõistlik algus on:

```
brew install python3 pandoc node sqlite jq ripgrep tmux gh
brew install --cask docker-desktop basictex
```

See annab:

- `python3`, `pip`, `venv`
- `pandoc` Markdowni konverteerimiseks
- `node` ja `npm`
- `sqlite3`
- `jq`, `ripgrep`, `tmux`, `gh`
- väiksema LaTeX-i baaskomplekti ja Dockeri

Kui tahad LaTeX-i võimalikult täielikku komplekti, siis võid `basictex` asemel kasutada:

```
brew install --cask mactex-no-gui
```

Praktikas ei paigaldata `basictex` ja `mactex-no-gui` cask'e tavaliselt koos. `basictex` on väiksem, `mactex-no-gui` on suurem ja täielikum.

Sama loogika on automatiseeritud ka skriptis:

```
./scripts/setup-mac.sh
```

Käivita need käsud

```
sudo apt update
sudo apt install htop
```

```
sudo dnf install htop
```

```
sudo pacman -S htop
```

```
brew install htop
```

```
python3 -m pip install requests
python3 -m pip install --user pipx
```

```
npm install
npm install lodash
```

Kõige tavalisemad lipud

- `apt install` paigaldab paketi
- `apt remove` eemaldab paketi
- `apt search` otsib pakette nime järgi
- `dnf install` paigaldab paketi Fedoras
- `pacman -S` paigaldab paketi Arch Linuxis

- `brew install` paigaldab paketi Homebrew kaudu
- `python3 -m pip install` paigaldab Pythoni paketi
- `npm install` paigaldab projekti sõltuvused

Soovitus

Märgi alati selgelt, millise distributsiooni käske parajasti näidatakse, sest paketihaldus ei ole kõigis süsteemides ühesugune.

Tavalised tegevused

```
sudo apt search ripgrep
sudo dnf search ripgrep
brew search ripgrep
```

```
sudo apt remove htop
sudo dnf remove htop
brew uninstall htop
```

```
python3 -m pip list
python3 -m pip show requests
```

```
npm list
npm run dev
```

pip kohta oluline märkus

Pythoni puhul on turvalisem kasutada kujut:

```
python3 -m pip install ...
```

mitte lihtsalt:

```
pip install ...
```

Nii on selgem, millise Pythoni tõlgendiga pakette paigaldatakse.

`npm` kohta oluline märkus

`npm` töötab tihti projekti sees koos failiga `package.json`.

- `npm install` paigaldab projekti sõltuvused
- `npm run nimi` käivitab projekti skripti
- globaalseid paigaldusi tasub teha ettevaatusega ja ainult siis, kui selleks on konkreetne põhjus

Kontrollkäigud pärast paigaldust

```
python3 --version
python3 -m pip --version
```

```
node --version
npm --version
pandoc --version
sqlite3 --version
```

Minitest

1. Uuri, milline paketihaldur sinu masinas on.
2. Otsi selle abil paketti `ripgrep` või `htop`.
3. Vaata, kuidas kuvatakse info ühe konkreetse paketi kohta.
4. Kontrolli, kas sinu masinas on olemas `python3`, `pip` ja `npm`.
5. Kui kasutad macOS-i, võrdle oma masinat siin soovitatud `brew` baaskomplektiga.

Lihtne veaotsing käsureal

Selles peatükis vaatame, mida kontrollida siis, kui käsk ei tööta nii nagu ootasid.

Loogika

Algajale on tihti raske aru saada, mida üldse kontrollida, kui ekraanile ilmub veateade. Väga sageli ei olegi vaja "suurt häkki", vaid rahulikku kontrolljärjekorda.

Hea käsurea veaotsing tähendab tavaliselt:

1. loe veateade lõpuni läbi
2. kontrolli, kus sa parajasti oled
3. kontrolli, kas käsk või fail on üldse olemas
4. kontrolli õigusi
5. kontrolli, kas probleem on shellis, paketis või teekonnas

Kiirspikker

- `pwd` näitab, kus sa oled
- `ls -lah` näitab, mis siin olemas on
- `command -v käsk` näitab, kas käsk on leitav
- `echo "$SHELL"` näitab praegust shelli
- `ls -l fail` aitab vaadata õigusi
- `käsk --help` või `man käsk` aitab kinnitada õiget süntaksit

Käivita need käsud

```
pwd
ls -lah
command -v bash
```

```
command -v rg
echo "$SHELL"
```

Kui mõni skript ei käivitu, siis kontrolli nii:

```
ls -l skript.sh
bash skript.sh
```

Kontrolljärjekord

Kui käsk ei tööta, siis alusta sellest:

1. mis täpselt oli käsk
2. mis oli täpne veateade
3. kas oled õiges kaustas
4. kas fail või käsk, millele viitad, on päriselt olemas
5. kas sul on vajalikud õigused

See kõlab lihtsana, aga suur osa vigu kukub just siia.

"command not found"

See tähendab tavaliselt üht neist:

- kirjutasid käsu nime valesti
- käsk ei ole paigaldatud
- käsk ei ole sinu `PATH`-is

Praktilised kontrollid:

```
command -v rg
command -v python3
echo "$PATH"
```

Kui `command -v` ei leia midagi, siis on järgmine küsimus tavaliselt:

- kas tööriist on puudu
- või kas ta on olemas, aga sinu shell ei leia seda

"No such file or directory"

See tähendab enamasti:

- oled vales kaustas
- failinimi on vale
- kasutad vale suhtelist või absoluutset teed

Kontrolli:

```
pwd
ls
ls kahtlane-fail
```

Kui fail peaks justkui olemas olema, aga käsk ikka ei leia seda, siis vaata, kas:

- suur- ja väiketähed on õiged
- kasutad `./fail` või `../fail` õigesti
- fail asub hoopis mõnes muus kaustas

"Permission denied"

See tähendab enamasti:

- sul ei ole õigust faili lugeda, kirjutada või käivitada
- skriptil puudub täitmisõigus
- üritad teha midagi süsteemikaustas ilma vajalike õigusteta

Kontrolli:

```
ls -l fail
ls -ld kaust
```

Kui probleem on skripti käivitamises, siis on väga tavaline põhjus lihtsalt see, et täitmisõigus puudub.

Vale shell või vale süntaks

Vahel ei ole probleem failis ega õigustes, vaid selles, et käsku tõlgendab teine shell, kui sa arvasid.

Kontrolli:

```
echo "$SHELL"
bash --version
zsh --version
```

Kui skript kasutab Bashi süntaksit, siis tasub vaadata, kas tal on korralik shebang, näiteks:

```
#!/usr/bin/env bash
```

Ja testi vajadusel nii:

```
bash skript.sh
```

Puudu olev pakett

Kui tööriista ei leita ja nimi on õige, siis võib lahendus olla lihtsalt paigaldus.

Näited:

```
sudo apt install ripgrep
sudo dnf install ripgrep
brew install ripgrep
```

Siin aitab sind peatükk Paketihaldus: apt, dnf, pacman, brew.

Hea veaotsingu küsimused

Kui tahad end ise kiiresti edasi aidata, siis küsi:

- kas ma olen õiges kaustas
- kas see fail on päriselt olemas
- kas see käsk on masinas olemas
- kas mul on vajalikud õigused
- kas ma kasutan õiget shelli

Kui neile küsimustele vastad, on suur osa "müstilistest" vigadest juba pooleldi lahendatud.

Minitest

1. Kontrolli, kas sinu masinas leitakse käsk `python3` käsuga `command -v`.
2. Tee meelega üks vale failinimi ja vaata, millise vea saad.
3. Seleta oma sõnadega, mida tähendab "command not found".
4. Seleta oma sõnadega, mida tähendab "permission denied".

Võrgu põhitööriistad

Selles peatükis vaatame väikest võrgu baasi, mis aitab kiiresti aru saada, kas probleem on käsus, failis või võrgus.

Loogika

Võrguprobleemi puhul tasub küsimus jagada väikesteks osadeks:

1. kas nimi või host üldse vastab
2. kas HTTP-teenus vastab
3. millised võrguliidesed masinas on
4. kas mõni port üldse kuulab

Need küsimused ei lahenda kõiki võrguvigu, aga annavad väga kiiresti esimese pildi.

Kiirspikker

- `ping` kontrollib, kas host vastab ICMP-le
- `curl -I` küsib veebiserverilt ainult päised
- `ip a` näitab Linuxis võrguliideseid
- `ifconfig` on levinud alternatiiv macOS-is ja BSD-s
- `ss -ltn` näitab Linuxis kuulavaid TCP-porte
- `lsof -iTCP -sTCP:LISTEN -n -P` on macOS-is kasulik kuulavate portide vaade

Käivita need käsud

Linuxis:

```
ping -c 4 example.com
curl -I https://example.com/
ip a
ss -ltn
```

macOS-is:

```
ping -c 4 example.com
curl -I https://example.com/
ifconfig
lsof -iTCP -sTCP:LISTEN -n -P
```

`ping`

`ping` on kiire kontrollküsimus:

"Kas see nimi või host üldse vastab?"

Näide:

```
ping -c 4 example.com
```

Oluline märkus:

- kõik hostid ei vasta `ping`-ile
- kui `ping` ei vasta, ei tähenda see alati, et veeb või teenus oleks maas

Seetõttu ei tasu `ping`-i võtta viimase tõena, vaid esimese vihjena.

`curl -I`

Kui sind huvitab veebiteenuse eluolu, siis `curl -I` on tihti kasulikum kui `ping`.

Näide:

```
curl -I https://example.com/
```

See näitab:

- kas HTTP või HTTPS vastab
- mis olekukood tuleb
- kas server teeb ümbersuunamise

Kui `curl -I` töötab, aga `ping` mitte, siis on teenus sageli ikkagi täiesti elus.

`ip a` ja `ifconfig`

Need käsud aitavad vaadata:

- millised liidesed masinas on

- kas mõnel liidesel on aadress
- kas masin paistab olevat üldse võrku ühendatud

Linuxis on tavalisem:

```
ip a
```

macOS-is ja mõnes vanemas süsteemis:

```
ifconfig
```

Siin ei pea algaja esialgu kõike mõistma. Esimene kasulik küsimus on lihtsalt:

- kas näen mõnd aktiivset liidest
- kas seal on IP-aadress

`ss -ltn` ja kuulavad pordid

Kui probleem on selles, et "teenus ei vasta", siis järgmine küsimus on:

"Kas see teenus üldse kuulab pordi peal?"

Linuxis:

```
ss -ltn
```

macOS-is praktiline vaste:

```
lsof -iTCP -sTCP:LISTEN -n -P
```

See aitab näha, kas mõni protsess kuulab näiteks porte nagu:

- `22` SSH jaoks
- `80` HTTP jaoks
- `443` HTTPS jaoks
- mõni rakenduse enda port

Väike kontrolljärjekord

Kui võrgus midagi ei tööta, siis liigu näiteks nii:

1. proovi `curl -I`, kas teenus vastab
2. proovi `ping`, kas nimi elab
3. vaata `ip a` või `ifconfig`, kas masinal on üldse mõistlik liides
4. vaata `ss -ltn` või alternatiiviga, kas kohalik teenus kuulab

See aitab eristada:

- nimeprobleemi
- ühenduseprobleemi
- kohaliku teenuseprobleemi

Minitest

1. Tee `curl -I` päring aadressile `https://example.com/`.
2. Vaata, milline käsk näitab sinu süsteemis võrguliideseid.
3. Vaata, kas sinu masinas on mõni kuulav TCP-port.
4. Pane ühe lausega kirja, miks `ping` ei anna alati täielikku vastust võrgu toimimise kohta.

Failide kopeerimine ja sünkroonimine

Selles peatükis võrdleme käske `cp`, `scp`, `rsync`, `wget` ja `curl`.

Loogika

Neid käske ühendab üks küsimus:

- kas liigutad faile samas masinas
- teise masinasse üle SSH
- või tõmbad midagi veebist

Just selle järgi tasub valida tööriist:

- `cp` kohalik koopia
- `scp` kiire kopeerimine üle SSH
- `rsync` nutikas sünkroniseerimine
- `curl` ja `wget` veebist toomine

Kiirspikker

- `cp` kopeerib lokaalseid faile
- `scp` kopeerib üle SSH
- `rsync` sünkroonib nutikalt
- `wget` laeb alla URL-ist
- `curl` teeb HTTP-päringuid ja allalaadimisi

Kõige sagedamini kasutatud lipud:

- `cp -R` kopeeri kataloog rekursiivselt
- `cp -a` GNU/Linuxis säilitab metaandmeid nii hästi kui võimalik
- `scp -r` kopeeri kataloog üle võrgu
- `scp -p` säilita faili ajad ja õigused nii hästi kui võimalik
- `rsync -a` säilita struktuur ja metaandmed
- `rsync -v` näita, mida tehakse
- `rsync -n` tee dry-run
- `curl -O` salvesta serveri failinimega
- `curl -L` järgi ümbersuunamisi

Käivita need käsud

```
cp fail.txt koopia.txt
cp -R kaust kaust-koopia

scp fail.txt kasutaja@server:/tmp/
rsync -av kaust/ kasutaja@server:/tmp/kaust/

wget https://example.com/fail.txt
curl -O https://example.com/fail.txt

curl -L -O https://example.com/arhiiv.tar.gz
curl -I https://example.com/
```

Millal mida kasutada

- `cp` kui allikas ja sihtkoht on samas masinas
- `scp` kui tahad lihtsalt faili üle SSH saata
- `rsync` kui sisu muutub ja tahad korduvat sünkroniseerimist
- `curl` või `wget` kui allikas on veebis

Praktiliselt:

- üks kiire koopia: `cp`
- üks kiire ülekanne serverisse: `scp`
- korduv sünkroonimine või varukoopia: `rsync`
- URL-ist faili tõmbamine: `curl -L -O`

`cp`, `scp`, `rsync` omavahel

Need kolm näevad sarnased välja, aga loogika on erinev.

`cp`:

```
cp fail.txt koopia.txt
cp -R kaust kaust-koopia
```

`scp`:

```
scp fail.txt kasutaja@server:/tmp/
scp -r kaust kasutaja@server:/tmp/
```

`rsync`:

```
rsync -av kaust/ kasutaja@server:/tmp/kaust/
rsync -avn kaust/ kasutaja@server:/tmp/kaust/
```

`rsync` on eriti tähtis just sellepärast, et:

- ta saadab ainult muutused
- ta sobib korduvaks tööks
- `-n` abil saab enne kontrollida, mida ta teeks

Kataloogipuu loogika

Kui kopeerid tervet kaustapuud, siis tasub alati läbi mõelda neli küsimust:

1. kas allikas ja sihtkoht on samas masinas või üle võrgu
2. kas teed ühekordset koopiat või korduvat sünkroonimist
3. kas tahad säilitada õigused, ajatemplid ja lingid
4. kas tahad pärast kontrollida, et tulemus sai õige

Just siin tuleb vahe eriti selgelt välja:

- `cp -R` teeb lihtsa kohaliku koopia
- `scp -r` saadab puu üle SSH, aga ei ole kõige mugavam korduva töö jaoks
- `rsync -av` sobib kõige paremini korduvaks sünkroonimiseks

Oluline detail `rsync` juures:

- `rsync -av kaust/ siht/` tähendab tavaliselt "kopeeri kausta sisu"
- `rsync -av kaust siht/` tähendab sagedamini "kopeeri kaust ise sihtkausta sisse"

See kaldkriipsu detail on väike, aga muudab tulemust palju.

`rsync` ja lõpu kaldkriips

See on üks kõige olulisemaid `rsync`-i detaile.

Näide:

```
mkdir -p ~/tmp/rsync-naide/allikas/alam
mkdir -p ~/tmp/rsync-naide/siht
printf 'tere\n' > ~/tmp/rsync-naide/allikas/alam/fail.txt
```

Nüüd võrdle kahte käsku:

```
rsync -avn ~/tmp/rsync-naide/allikas ~/tmp/rsync-naide/siht/
rsync -avn ~/tmp/rsync-naide/allikas/ ~/tmp/rsync-naide/siht/
```

Loogika:

- ilma lõpu kaldkriipsuta kopeeritakse tavaliselt kaust `allikas` ise sihtkoha sisse
- lõpu kaldkriipsuga kopeeritakse kausta `allikas` sisu sihtkohta

Just sellepärast tasub enne päris sünkroonimist teha:

```
rsync -avn allikas/ siht/
```

`-n` ehk `--dry-run` aitab enne näha, mida käsk teeks.

Metaandmed: õigused, omanikud, ajatemplid

Suure puu puhul ei ole tähtis ainult faili sisu. Sageli on tähtsad ka:

- faili õigused
- omanik ja grupp
- ajatemplid
- sümboolsed lingid

Rusikareeglid:

- `cp -R` keskendub eelkõige sisule ja struktuurile
- `cp -a` GNU/Linuxis püüab säilitada metaandmeid võimalikult terviklikult
- `scp -p` säilitab ajatemplid ja õigused paremini kui paljas `scp`
- `rsync -a` on tavaliselt kõige mõistlikum valik, kui metaandmed loevad

Omaniku kohta tasub meeles pidada:

- tavaline kasutaja ei saa üldjuhul taastada suvalise teise kasutaja omandit
- kaugserveris sõltub lõplik omanik sageli sellest, mis kasutajana sa sisse logisid
- seetõttu võib "sisu sama, aga omanik teine" olla täiesti ootuspärane tulemus

`curl` ja `wget`

Mõlemad oskavad faile alla laadida, kuid rõhuasetus on veidi erinev:

- `wget` on klassikaline allalaadija
- `curl` on üldisem HTTP-klient ja sobib hästi ka API-dega rääkimiseks

Kui tahad veebilehte tekstina lugeda, linke kokku koguda või teha väikest crawl'i, siis vaata edasi peatükki Veebist sisu toomine ja tekstivaade: curl, wget, lynx.

Levinud `curl` võtmed:

- `-O` salvesta serveri failinimega
- `-o fail` salvesta kindla nimega
- `-L` järgi ümbersuunamisi
- `-I` küsi ainult päised

Näited:

```
curl -o naide.html https://example.com/
curl -L -O https://example.com/fail.txt
curl -I https://example.com/
```

Need käsud on väga kasulikud ka selleks, et kontrollida, kas URL üldse vastab ootuspäraselt.

Kõige tavalisemad päriselu näited

Kopeeri projektikaust varuks:

```
rsync -av projekt/ projekt-varu/
```

Saada üks fail serverisse:

```
scp backup.sql kasutaja@server:/tmp/
```

Tõmba arhiiv veebist:

```
curl -L -O https://example.com/arhiiv.tar.gz
```

Kontrolli enne päris sünkroniseerimist, mida `rsync` teeks:

```
rsync -avn projekt/ server:/srv/projekt/
```

Kui teed suure või tundliku sünkroonimise, siis `--dry-run` võiks olla peaaegu automaatne esimene samm.

Minitest

1. Kopeeri üks fail uue nime alla.
2. Tee kaustast rekursiivne koopia.
3. Uuri `rsync --help` abil, mida teeb võti `-a`.
4. Tee `curl -I` abil päring mõnele veebiaadressile ja vaata päiseid.

Kauglogimine ja SSH

Selles peatükis käsitleme `ssh`, võtmeid, agenti ja turvalisi ühendusi teistesse masinatesse.

Loogika

SSH lahendab kolm väga sagedast probleemi:

1. logi turvaliselt teise masinasse sisse
2. käivita seal käske
3. liiguta faile sama turvalise kanali kaudu

Sellepärast on SSH seotud ka käskudega `scp`, `rsync` ja Git.

Kiirspikker

- `ssh kasutaja@host` logib kaugmasinasse
- `ssh -p 2222 kasutaja@host` kasutab teist porti
- `ssh-keygen` loob võtmed
- `ssh-copy-id kasutaja@host` kopeerib avaliku võtme serverisse
- `scp` ja `rsync` kasutavad sageli sama SSH-kanalit

Kõige tavalisemad lipud, mida tasub päriselt meeles pidada:

- `-p` port
- `-i` konkreetne võti
- `-v` veaotsingu jaoks jutukam väljund

Käivita need käsud

```
ssh kasutaja@server.example.org
ssh -p 2222 kasutaja@server.example.org
```

```
ssh -i ~/.ssh/id_ed25519 kasutaja@server.example.org
ssh kasutaja@server.example.org 'hostname && uptime'
```

Võtmete põhimõte

- privaatvõti jääb sinu arvutisse
- avalik võti kopeeritakse serverisse
- server lubab sisse, kui võtmepaar sobib

See on põhjus, miks võtmetega autentimine on nii tavaline:

- mugavam kui iga kord parooli sisestada
- tavaliselt turvalisem
- vajalik paljudes automaatsetes töövoogudes

`~/.ssh/config`

Kui ühendud tihti samade masinatega, tasub kasutada config-faili.

Siin tähendab `~` sinu kodukataloogi lühikuju.

Näide:

```
Host opik-server
    HostName server.example.org
    User vilo
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
```

Pärast seda saab ühendada lihtsalt nii:

```
ssh opik-server
scp fail.txt opik-server:/tmp/
rsync -av kaust/ opik-server:/tmp/kaust/
```

See on hea näide sellest, kuidas SSH seob omavahel sisselogimise ja failide kopeerimise.

Kõige tavalisem töövoog

1. loo võtmepaar `ssh-keygen`
2. kopeeri avalik võti serverisse
3. testi sisselogimist
4. lisa vajadusel `~/.ssh/config`

Võtmete loomine samm-sammult

See osa ei ole enam lihtsalt kiirspikker, vaid väike eraldi töövoog.

Kõige tavalisem turvaline algus on:

```
ssh-keygen -t ed25519 -C 'minu-voti-kommentaar'
ssh-add ~/.ssh/id_ed25519
```

Selle taga on järgmine loogika:

1. `ssh-keygen` loob võtmepaari
2. privaatvõti jääb sinu arvutisse
3. avalik võti viiakse serverisse
4. `ssh-add` lisab võtme agendile, et peaksid harvemini parooli või paroolifraasi sisestama

Avalikku ja privaatset võtit ei tohi segi ajada:

- privaatvõtit ei saadeta teistele
- avaliku võtme võib kopeerida serverisse

Kui sinu süsteemis on olemas `ssh-copy-id`, siis on avaliku võtme serverisse lisamine sageli kõige lihtsam:

```
ssh-copy-id kasutaja@server.example.org
```

Kui seda käsku ei ole, siis tuleb avalik võti lisada serveri faili `~/.ssh/authorized_keys` mõnel muul viisil.

Mis jääb hilisemaks

SSH-l on ka keerukamaid võimalusi, näiteks port forwarding ja agent forwarding. Need on kasulikud, aga alguses ei ole nad “esimesed lipud, mis tuleb pähe õppida”.

Veaotsing

Kui ühendus ei tööta, siis kõige kasulikum esimene käsk on sageli:

```
ssh -v kasutaja@server.example.org
```

See aitab näha:

- millist võtit prooviti
- kas port on õige
- kas autentimine kukkus läbi või ühendust ei saadud üldse

Minitest

1. Vaata, kas sul on kaust `~/.ssh`.
2. Uuri, millised võtmed sul seal juba olemas on.

3. Pane kirja, mis vahe on avalikul ja privaatsel võtmel.
4. Kirjuta näidis `~/.ssh/config` ühe kujuteldava serveri jaoks.

Veebist sisu toomine ja tekstivaade: curl, wget, lynx

Selles peatükis vaatame, kuidas veebist sisu alla tuua, HTML-i tekstiks muuta ja vajadusel teha väike kontrollitud crawl.

Loogika

Veebist sisu toomisel tasub kõigepealt eristada kolme eri ülesannet:

1. too üks vastus või fail
2. laadi alla terve lehtede puu või jätka katkestatud tõmmet
3. vaata HTML-i loetava tekstina või kogu sealt lingid kokku

Just selle järgi tasub tööriist valida:

- `curl` ühe päringu, päiste, API või toor-HTML jaoks
- `wget` allalaadimise, jätkamise ja crawl'i jaoks
- `lynx` HTML-i tekstivaate ja linkide loendi jaoks

Sõna "scrape" tähendab siin lihtsalt seda, et võtad veebist sisu ja töötled seda edasi käsureal. Kui ametlik API või andmefail on olemas, siis eelista seda peaaegu alati HTML-i kraapimisele.

Enne kui kraabid

Enne automaatset allalaadimist tasub alati kontrollida mõnda lihtsat asja:

- kas saidil on olemas API või andmeeksport
- kas `robots.txt` või kasutustingimused lubavad seda tööd
- kas saad alustada väikese prooviga, mitte kohe terve domeeniga
- kas lisad päringute vahele pausi, kui teed korduvaid tõmbeid
- kas suudad hiljem tõestada, kust andmed tulid ja millal sa need tõid

Hea reegel on: alusta ühe URL-iga, kontrolli tulemust ja alles siis mõtle suurema crawl'i peale.

Kiirspikker

- `curl -I URL` küsib ainult päised
- `curl -L -o fail.html URL` salvestab vastuse faili ja järgib ümbersuunamisi
- `curl -sL URL | lynx -stdin -dump` muudab veebilehe tekstiks
- `wget URL` laeb URL-i vaikimisi faili
- `wget -c URL` jätkab katkestatud allalaadimist

- `wget --recursive --level=1 --no-parent URL` teeb väikese piiratud crawl'i
- `lynx URL` avab tekstilise veebivaate interaktiivselt
- `lynx -dump URL` trükib lehe teksti koos viidetega välja
- `lynx -dump -listonly URL` trükib välja ainult linkide nimekirja

Kui tööriist puudub

`curl` on sageli juba olemas, kuid `wget` ja `lynx` ei pruugi igas süsteemis vaikimisi paigaldatud olla.

```
sudo apt install wget lynx
sudo dnf install wget lynx
brew install wget lynx
```

Käivita need käsud

See plokk näitab ühe väikese ohutu töövoo:

- vaata, kas URL üldse vastab
- salvesta leht faili
- muuda HTML tekstiks
- kogu linkide nimekiri eraldi välja

```
mkdir -p veeb-naide
cd veeb-naide
curl -I https://example.com/
curl -L -o naide-curl.html https://example.com/
wget -O naide-wget.html https://example.com/
lynx -dump naide-curl.html | sed -n '1,20p'
lynx -dump -listonly https://example.com/
ls -lh
```

Kui tahad veebist tulnud HTML-i kohe torusse panna, siis tee nii:

```
curl -sL https://example.com/ | lynx -stdin -dump | sed -n '1,20p'
```

`curl`: üks vastus korraga

`curl` on hea siis, kui tahad täpselt kontrollida, mis päring tehakse ja mida vastuseks saadakse.

Olulised lipud alguses:

- `-I` ainult päised
- `-L` järgi ümbersuunamisi
- `-o fail` salvesta kindla nimega
- `-O` salvesta serveri pakutud nimega
- `-s` vaikne reziim

- `-S` näita viga ka siis, kui kasutad `-s`

Näited:

```
curl -I https://example.com/
curl -L -o esileht.html https://example.com/
curl -sL https://example.com/ | grep -i '<title'
curl -sL https://example.com/ | lynx -stdin -dump
```

Praktiline mõte on siin lihtne:

- `curl` kirjutab vaikimisi väljundi ekraanile
- seetõttu sobib ta hästi torude ja filtritega
- `curl` ise ei tee "veebipuud", vaid ühe või mõne konkreetse päringu

`wget`: allalaadija ja crawler

`wget` on mugav siis, kui tahad, et fail päriselt kettale jääks, või kui tahad teha piiratud rekursiivset allalaadimist.

Olulised lipud alguses:

- `-O fail` salvesta kindla nimega
- `-c` jätka katkestatud tõmmet
- `-P kaust` salvesta kindlasse kausta
- `--recursive` rekursiivne allalaadimine
- `--level=1` või `-l 1` piira sügavust
- `--no-parent` ära mine ülemkataloogidesse
- `--page-requisites` tõmba lehe toimimiseks vajalikud failid
- `--convert-links` muuda lingid lokaalses koopias sobivaks
- `--adjust-extension` pane HTML-failidele sobiv laiend
- `--wait=1` ja `--random-wait` tee viisakam crawl

Näited:

```
wget https://example.com/
wget -O esileht.html https://example.com/
wget -c -O esileht-koopia.html https://example.com/
```

Väike piiratud crawl:

```
wget --recursive --level=1 --no-parent https://example.com/
```

Kui tahad teha lokaalse koopiana väikest dokumentatsioonipuud, siis mall on näiteks selline:

```
wget \
  --mirror \
  --convert-links \
  --adjust-extension \
  --page-requisites \
  --no-parent \
```

```
  --wait=1 \
  --random-wait \
  https://docs.example.org/juhend/
```

Seda viimast käsku ei tasu kunagi pimesi suvalise suure saidi juurel käivitada. Kõigepealt kontrolli alati:

- kui suur ala sul tegelikult vaja on
- kas `--no-parent` ja õige alg-URL piiravad töö piisavalt kitsaks
- kas serverile on mõistlik teha nii palju päringuid

`lynx`: HTML tekstiks ja lingid välja

`lynx` on kasulik siis, kui veebileht on vaja teha kiiresti loetavaks tekstiks.

See on hea tööriist näiteks siis, kui:

- tahad lehte lugeda terminalist
- tahad HTML-ist saada lihtsa tekstivaate
- tahad linkide nimekirja eraldi kätte
- tahad torust tuleva HTML-i kiiresti puhastada enne `grep`-i või `sed`-i

Kõige tavalisemad töökujud:

```
lynx https://example.com/
lynx -dump https://example.com/
lynx -dump -listonly https://example.com/
```

Kui sul on HTML juba failis, siis saad sama teha lokaalselt:

```
lynx -dump naide-curl.html
lynx -dump -listonly naide-curl.html
```

Vahe on lihtne:

- `lynx URL` on interaktiivne tekstibrauser
- `lynx -dump URL` prindib loetava tekstivaate ekraanile
- `lynx -dump -listonly URL` prindib ainult lingid

Väike scrape-töövoog

Kui eesmärk ei ole "terve sait alla", vaid "too üks leht ja töötle seda", siis üks praktiline töövoog on:

```
curl -L -o leht.html https://example.com/
lynx -dump leht.html > leht.txt
grep -n 'Example' leht.txt
lynx -dump -listonly leht.html > lingid.txt
wc -l lingid.txt
```

See töövoog on hea, sest iga samm on kontrollitav:

1. tõmbad algse HTML-i faili
2. teed sellest inimesele loetava tekstiversiooni
3. otsid tekstist mustrit
4. võtad lingid eraldi nimekirjana välja

See seob hästi kokku ka peatükid `grep`, `sed`, `sort` ja `awk`.

Mida tasub meeles pidada

- `curl` sobib paremini üksikute päringute ja torude jaoks
- `wget` sobib paremini allalaadimise ja crawl'i jaoks
- `lynx` ei ole "crawler", vaid tekstivaate ja linkide tööriist
- HTML-i scrape on habras, sest lehe struktuur võib muutuda
- JavaScriptiga ehitatud sait ei pruugi anda `curl`-ile või `wget`-ile sama pilti, mida näeb brauser

Kui näed, et info tuleb lehele alles JavaScripti abil, siis puhas `curl` või `wget` ei pruugi sulle tegelikku sisu anda. Siis tasub otsida:

- ametlikku API-t
- JSON-vastust brauseri võrgupaneelist
- eksportfaili nagu CSV või JSON

Minitest

1. Tee `curl -I` päring aadressile `https://example.com/`.
2. Salvesta sama leht korraga nii `curl`-i kui `wget`-iga.
3. Muuda üks HTML-fail `lynx -dump` abil tekstiks.
4. Trüki `lynx -dump -listonly` abil välja ainult lingid.
5. Seleta ühe lausega, millal kasutaksid `curl`, millal `wget` ja millal `lynx`.

Arhiivid ja pakkimine

Selles peatükis vaatame, kuidas faile kokku pakkida, lahti pakkida ja transportimiseks või varunduseks arhiveerida.

Loogika

Arhiveerimine koondab failid, pakkimine teeb need väiksemaks. See on seotud failide liigutamise, allalaadimise ja varukoopiatega.

Kiirspikker

- `tar -cf fail.tar kaust/` loo arhiiv
- `tar -xf fail.tar` paki lahti
- `tar -tf fail.tar` vaata arhiivi sisu
- `tar -czf fail.tar.gz kaust/` loo gzip-pakitud arhiiv

- `tar -czf fail.tgz kaust/` sama loogika lühema laiendiga
- `tar -xzf fail.tar.gz` paki gzip-arhiiv lahti
- `tar -cJf fail.tar.xz kaust/` loo xz-pakitud arhiiv
- `zip -r fail.zip kaust/` loo zip-arhiiv
- `unzip fail.zip` paki zip lahti

Tähtsamad võtmed

- `c` create ehk loo arhiiv
- `x` extract ehk paki lahti
- `t` table ehk näita sisu
- `f` file ehk järgmine argument on arhiivifaili nimi
- `z` kasuta `gzip` pakkimist
- `J` kasuta `xz` pakkimist
- `v` verbose ehk näita töö käigus rohkem infot

`tar` põhimõte

`tar` ise on ajalooliselt arhiveerija. Pakkimine lisatakse sageli eraldi:

- `tar` ainult koondab failid
- `gzip` teeb faili väiksemaks
- `xz` pakib tihedamalt, aga võib olla aeglasem

Sellepärast on need kujundid sisuliselt järgmised:

- `tar -cf` loo arhiiv ilma pakkimata
- `tar -czf` loo `gzip`-ga pakitud arhiiv
- `tar -xzf` paki `gzip`-ga pakitud arhiiv lahti
- `tar -cJf` loo `xz`-ga pakitud arhiiv

`.tar.gz` ja `.tgz` tähendavad tavaliselt sama asja. `.tgz` on lihtsalt lühem failinimi.

Käivita need käsud

```
mkdir -p ~/tmp/arh/kaust
printf 'tere\n' > ~/tmp/arh/kaust/tere.txt
tar -cf ~/tmp/arh/proov.tar -C ~/tmp/arh kaust
tar -tf ~/tmp/arh/proov.tar

tar -czf ~/tmp/arh/proov.tar.gz -C ~/tmp/arh kaust
tar -xzf ~/tmp/arh/proov.tar.gz -C ~/tmp/arh

zip -r ~/tmp/arh/proov.zip ~/tmp/arh/kaust
unzip ~/tmp/arh/proov.zip -d ~/tmp/arh/unzipped
```

Millal mida kasutada

- `tar.gz` on väga levinud Linuxi maailmas
- `tar.xz` sobib siis, kui tihendusaste on tähtis
- `zip` on mugav, kui faile jagatakse erinevate süsteemide vahel

Metaandmed ja puustruktuur

Arhiiviformaadi valikul ei loe ainult tihendusaste. Loeb ka see, mida arhiiv peab kaasa võtma.

- `tar` sobib hästi Unix-laadse puustruktuuri jaoks
- `tar` säilitab failipuud, õigused, ajatemplid ja lingid paremini kui `zip`
- `zip` on sageli mugavam jagamiseks eri süsteemide vahel
- `zip` ei ole tavaliselt parim valik siis, kui Unix-i õigused ja omanikud on olulised

Praktiline mõtteviis:

- Linuxi või serveri varukoopia: eelista sageli `tar.gz` või `tar.xz`
- laiemaks jagamiseks: `zip`

Kasulikud võtted

- `tar -tf fail.tar.gz` näitab sisu ilma lahti pakkimata
- `tar -tzf fail.tar.gz | less` laseb suure arhiivi sisu sirvida
- `tar -xzf fail.tar.gz -C sihtkaust` pakib lahti kindlasse kohta
- `tar -czf backup-$(date +%F).tar.gz kaust/` teeb kuupäevaga varukoopia
- `unzip -l fail.zip` näitab zip-arhiivi sisu ilma lahti pakkimata

Kui arhiiv on suurem, siis on väga praktiline:

```
tar -tzf fail.tar.gz | less
```

See aitab enne lahtipakkimist näha:

- mis kaustad seal sees üldse on
- kas arhiivis on ootuspärane juurkaust
- kas failinimed paistavad mõistlikud

Minitest

1. Loo väike testkaust kahe failiga.
2. Tee sellest `tar.gz` arhiiv.
3. Vaata arhiivi sisu ilma seda lahti pakkimata.
4. Paki arhiiv lahti teise kausta.

Tervete kataloogipuude haldus ja jagamine

See peatükk koondab üheks töövooks käsud `cp`, `rsync`, `scp`, `tar`, `zip` ja Git-i loogika. Mõte ei ole korrata kõiki üksikkäske, vaid anda tervikpilt: mida teha siis, kui sul on vaja hallata või jagada tervet projektipuud.

Loogika

Kui liigud üksikfailidest edasi tervete kaustapuudeni, siis küsimus ei ole enam ainult "kuidas kopeerida üks fail", vaid:

1. kas tahan teha kohaliku koopia
2. kas tahan saata puu teise masinasse
3. kas tahan teha ühest hetkest arhiivi
4. kas tahan jagada seda nii, et hiljem saaks muudatusi jälgida
5. kuidas kontrollida, et tulemus sai õige

Just siin lähevad tööriistad oma tugevuste järgi lahku.

Millal millist tööriista kasutada

- `cp -R` sobib, kui tahad samas masinas teha lihtsa koopia
- `rsync -av` sobib, kui tahad korduvat sünkroonimist või varundust
- `scp -r` sobib, kui tahad kiiresti terve puu SSH kaudu teise masinasse saata
- `tar -czf` või `tar -cJf` sobib, kui tahad ühest hetkest ühte arhiivifaili
- `zip -r` sobib, kui jagad sisu inimestega, kes töötavad eri süsteemides
- Git sobib, kui tahad mitte ainult jagada tulemust, vaid ka jälgida muudatusi ajas

Kohalik koopia: `cp`

Kui vajad ühekordset koopiat samas masinas, siis alusta lihtsast käsust:

```
cp -R projekt projekt-koopia
```

See on hea valik siis, kui:

- tahad kiiresti proovida midagi teises koopias
- tahad teha enne suuremat muutust kohaliku varu
- allikas ja sihtkoht on samal masinal

Kui metaandmed loevad, siis GNU/Linuxis kasutatakse sageli:

```
cp -a projekt projekt-koopia
```

Seda tasub võtta kui "säilita nii palju kui võimalik". Eri süsteemid ei käitu siin alati täpselt ühtemoodi.

Korduv sünkroonimine: `rsync`

Kui sama puud liigutatakse korduvalt, siis `rsync` on tavaliselt parem kui `cp` või `scp`.

```
rsync -av projekt/ projekt-varu/
```

Selle käsu tugevused:

- saadab uuesti peamiselt muutunud sisu
- säilitab struktuuri ja metaandmeid paremini
- sobib nii lokaalseks kui kaugsünkroonimiseks

Oluline kaldkriipsu loogika:

- `projekt/` tähendab enamasti "selle kausta sisu"
- `projekt` tähendab sagedamini "see kaust tervikuna"

Enne suuremat kopeerimist on väga mõistlik teha dry-run:

```
rsync -avn projekt/ projekt-varu/
```

Kui siht on serveris:

```
rsync -av projekt/ kasutaja@server:/srv/projekt/
```

Kiire ülekanne teise masinasse: `scp`

`scp` on hea siis, kui tahad lihtsalt midagi kiiresti teise masinasse saata:

```
scp -r projekt kasutaja@server:/tmp/
```

See on praktiline, kui:

- vajad ühekordset üleslaadimist
- sul ei ole vaja keerukamat sünkroonimisloogikat
- SSH on juba seadistatud

Kui tahad võimalikult palju säilitada ajatemplite ja õiguste kohta, kasuta sageli:

```
scp -rp projekt kasutaja@server:/tmp/
```

Üks fail kogu puust: `tar`, `tgz`, `zip`

Kui tahad tervest puust teha ühe jagatava faili, siis sobib arhiiv:

```
tar -czf projekt-2026-04-13.tgz projekt/
tar -tf projekt-2026-04-13.tgz
```

See on hea valik siis, kui:

- tahad saata ühe faili
- tahad võtta kindla hetke snapshot'i
- tahad arhiivi enne lahti pakkimata kontrollida

Kui adressaadil on tõenäoliselt Linux või macOS, on `tar.gz` või `.tgz` sageli loomulik valik.

Kui adressaadid on väga eri keskkondades, on `zip` mugav:

```
zip -r projekt.zip projekt/
unzip -l projekt.zip
```

Mis saab õigustest ja omanikest

Suure puu halduses on oluline eristada nelja asja:

- faili sisu
- failipuu struktuur
- õigused
- omanikud ja grupid

Rusikareeglid:

- `cp -R` ja `scp -r` lahendavad eelkõige sisu ja struktuuri
- `rsync -a` ja `tar` hoiavad Unix-laadset metaandmete pilti paremini koos
- `zip` sobib rohkem jagamiseks kui täpseks Unix-i säilitamiseks
- Git ei talleta tavaliselt omanikku, gruppi ega ajatemplite ajalugu

Git talletab hästi:

- faili sisu
- kataloogistruktuuri
- tekstimuudatuste ajaloo
- täidetavusbitte olulisemates juhtudes

Git ei ole varukoopia-arhiiv igas mõttes. See ei asenda `tar`-i ega `rsync`-i, kui tähtis on kogu failisüsteemi metaandmete võimalikult täpne ülekandmine.

Jagamine: arhiiv või GitHub

Kui eesmärk on lihtsalt "saada see tervik teisele inimesele", siis mõtle nii:

- üks ühekordne hetkeseis: `tar.gz` või `zip`
- korduv uuendamine serverisse: `rsync`
- ühine arendus ja muudatuste ajalugu: Git + GitHub

Seega:

- andmepuu või snapshot: arhiiv
- deploy või varu: `rsync`
- koostöö ja versioonihaldus: Git

Kuidas kontrollida, et said õige asja

Suure puu juures on kontroll sama tähtis kui kopeerimine ise.

Kõige praktilisemad kontrollid:

- vaata failide arvu ja suurust
- loe arhiivi sisu ilma lahti pakkimata
- tee `rsync`-iga dry-run
- võrdle vähemalt mõne võtmefaili räsi

Näited:

```
du -sh projekt projekt-koopia
find projekt | wc -l
find projekt-koopia | wc -l
```

```
tar -tf projekt-2026-04-13.tgz | head
unzip -l projekt.zip
```

```
shasum -a 256 projekt/README.md projekt-koopia/README.md
```

Linuxis on sama loogika sageli kujul:

```
sha256sum projekt/README.md projekt-koopia/README.md
```

Soovitatud töövood

1. Enne riskantset muutust

```
rsync -av projekt/ projekt-varu/
```

2. Saada tervik serverisse

```
rsync -avn projekt/ kasutaja@server:/srv/projekt/
rsync -av projekt/ kasutaja@server:/srv/projekt/
```

3. Tee jagatav snapshot

```
tar -czf projekt-$(date +%F).tgz projekt/
tar -tf projekt-$(date +%F).tgz | head
```

4. Jaga koostööks

```
git status
git add .
git commit -m 'Valmista projekt jagamiseks'
git push
```

Viimase töövoo detailid tulevad eraldi Git-i peatükis.

Minitest

1. Tee ühest testkaustast kohalik koopia käsuga `cp -R`.
2. Tee samast kaustast `rsync -avn` dry-run teise kausta.

3. Paki sama kaust `.tgz` faili ja kuva selle sisu käsuga `tar -tf`.
4. Mõtle ühe näite põhjal, kas sinu eesmärk on snapshot, sünkroonimine või koostöö ajalooga.

Protsessid, tööd ja signaalid

Selles peatükis vaatame, kuidas jälgida töötavaid protsesse, saata neile signaale ning kasutada shelli tööde juhtimist.

Loogika

Siin on oluline eristada kolme asja:

1. protsess
2. shelli töö
3. signaal

Need on seotud, aga mitte samad:

- protsess on käivitatud programm
- töö on shelli vaates hallatav käsk või käsujada
- signaal on viis protsessile juhtsõnum saata

Kiirspikker

- `ps` näitab protsesse
- `ps aux` või `ps -ef` näitab rohkem infot
- `top` või `htop` näitab protsesse reaalajas
- `kill PID` saadab protsessile signaali
- `kill -9 PID` lõpetab protsessi jõuga
- `käsk &` käivitab töö taustal
- `jobs` näitab shelli taustatöid
- `bg` jätkab tööd taustal
- `fg` toob töö esiplaanile
- `wait` ootab taustatööd ära
- `Ctrl-c` katkestab käsu
- `Ctrl-z` peatab käsu ajutiselt

Kõige tavalisemad tegevused päriselus on:

- vaata, mis jookseb
- peata või lõpeta kinni jäänud käsk
- saada pikk töö taustale

Protsesside vaatamine

```
ps
ps aux
```

```
ps -ef
```

```
top
htop
```

`top` ja `htop` näitavad:

- protsessi ID-d
- kasutatavat mälu
- protsessori koormust
- käimasolevaid käske

Kõige kasulikumad variandid alguses on tavaliselt:

```
ps aux | grep python
top
htop
```

ehk kas otsid konkreetset protsessi või vaatad tervikut reaalajas.

Kui tahad väga kiiresti näha suurimaid protsessiressursside kasutajaid, siis need on head 1-linerid:

```
ps aux | sort -nrk 3 | head
ps aux | sort -nrk 4 | head
```

Siin:

- veerg `3` on tavaliselt CPU kasutus
- veerg `4` on tavaliselt mälu kasutus

See on väga praktiline, kui küsid:

- mis praegu protsessorit sööb
- mis võtab kõige rohkem mälu

Protsessi lõpetamine

Igal protsessil on tavaliselt PID ehk protsessi number.

```
kill 12345
kill -15 12345
kill -9 12345
```

Tavapraktika:

- proovi esmalt tavalist `kill` või `kill -15`
- kasuta `kill -9` ainult siis, kui protsess ei allu viisakamale lõpetamisele

See loogika on oluline, sest:

- `SIGTERM` annab programmile võimaluse ise viisakalt lõpetada
- `SIGKILL` katkestab ta jõuga

Seepärast ei tasu `kill -9` teha automaatselt esimeseks valikuks.

Tööd shellis: `jobs`, `fg`, `bg`

Shell oskab hallata käske ka töödena.

Näide:

```
sleep 300
```

vajuta seejärel `Ctrl-z`, et töö peatada, ja siis:

```
jobs
bg
jobs
fg
```

Tähendus:

- `Ctrl-z` peatab töö ajutiselt
- `jobs` näitab shelli teadaolevaid töid
- `bg` jätkab peatatud tööd taustal
- `fg` toob töö tagasi esiplaanile

Taustal saab töö käivitada ka kohe:

```
sleep 300 &
jobs
```

Kõige tavalisem lühike töövoog on:

1. käivita käsk
2. saad aru, et see võtab kaua aega
3. vajuta `Ctrl-z`
4. tee `bg`
5. vaata `jobs`

See on põhjus, miks `jobs`, `bg` ja `fg` on seotud terminali ja shelli peatükkidega.

Järjest või korraga

Protsesside ja tööde juures tekib väga sageli küsimus: kas käsud käivad üksteise järel või samal ajal?

Kõige lihtsam rusikareegel on:

- `käsk1 ; käsk2` tähendab: tee järjest
- `käsk &` tähendab: saada käsk taustale ja jätka kohe järgmisega

Järjestikune näide:

```
date '+%H:%M:%S'
sleep 3
echo 'kolm sekundit hiljem'
sleep 1
echo 'veel üks sekund hiljem'
```

Siin teine `sleep` ei alga enne, kui esimene on lõpetanud.

Taustaga näide:

```
sh -c 'sleep 3; echo "pikk töö valmis"' &
sh -c 'sleep 1; echo "lühike töö valmis"' &
jobs
wait
```

Siin käivad kaks tööd korraga. Kuigi "pikk töö" käivitati enne, võib "lyhike töö" lõpetada varem.

See ongi üks tähtis erinevus:

- järjestikuses jadas määrab järjekorra shell
- taustatööde puhul võivad lõpetamisajad olla teistsugused kui käivitamisjärjekord

Signaalid lühidalt

Signaal on lühike juhtsõnum protsessile.

Levinud näited:

- `SIGTERM` palub protsessil viisakalt lõpetada
- `SIGKILL` lõpetab protsessi jõuga
- `SIGINT` tekib sageli `Ctrl-c` vajutamisel
- `SIGTSTP` tekib sageli `Ctrl-z` vajutamisel

Kõige tavalisemad näited

Leia mõni protsess:

```
ps aux | grep ssh
```

Katkesta esiplaanil töötav käsk:

- vajuta `Ctrl-c`

Peata ja saada töö taustale:

```
sleep 300
```

siis `Ctrl-z`, edasi:

```
bg
jobs
```

Päris näide: taustale saadetud pikk töö

Kui tahad õppida `jobs`, `bg` ja `fg` loogikat, siis `sleep` on endiselt üks parimaid harjutusi.

```
sleep 300
```

Seejärel:

- vajuta `Ctrl-z`
- käivita `jobs`
- käivita `bg`
- käivita uuesti `jobs`

Siin näed väga selgelt:

- kuidas esiplaanil olev käsk peatatakse
- kuidas shell muudab selle tööks
- kuidas sama töö jätkub taustal

Ja lõpetuseks:

```
fg
```

või teises variandis:

```
kill %1
```

See on hea “päris shelli” näide, sest siin ei pea PID-i kohe käsitsi teadma.

Päris näide: kaks peatatud tööd ja `%1`, `%2`

Kui tahad aru saada, mida tähendavad `fg %1` ja `bg %2`, siis tee teadlikult kaks peatatud tööd.

Kõigepealt:

```
sleep 120
```

vajuta `Ctrl-z`.

Siis:

```
sleep 240
```

vajuta jälle `Ctrl-z`.

Nüüd:

```
jobs
bg %1
jobs
fg %2
```

Loogika on:

- `%1` tähendab töö number 1 shelli `jobs` nimekirjas
- `%2` tähendab töö number 2
- `bg %1` jätkab esimese töö taustal
- `fg %2` toob teise töö ette

Kui tood töö 2 ette ja peatad selle uuesti `Ctrl-z` abil, siis võid sama loogikat jätkata ka nii:

```
bg %2
jobs
```

See on hea viis harjutada, et töö numbrid tulevad `jobs` väljundist, mitte "kõhutunde järgi".

See ei ole sama asi mis PID:

- `%1` on shelli töö number
- `12345` tüüpi number on protsessi PID

Kui tahad taustal jooksva töö lõpetada shelli töö numbri järgi, siis võid teha ka:

```
kill %1
```

See on tihti mugavam kui hakata PID-i käsitsi otsima.

Päris näide: hiljem käivitatud töö võib enne lõpetada

See on väga hea harjutus mõistmaks, mida tähendab "korraga":

```
sh -c 'sleep 5; echo "viie sekundi töö lõpetas"' &
sh -c 'sleep 2; echo "kahe sekundi töö lõpetas"' &
jobs
wait
```

Siin käivitati viie sekundi töö enne, aga kahe sekundi töö lõpetab varem.

See aitab hästi eristada:

- käivitamisjärjekorda
- tegelikku lõpetamisjärjekorda

Just taustatööde juures ei ole need alati samad.

Mis juhtub, kui terminali aken läheb kinni

Siin tuleb veel üks oluline vahe:

- `jobs`, `bg`, `fg` töötavad sinu praeguse shelli sees
- kui see shell lõpeb, võivad ka tema taustatööd lõppeda

See tähendab, et pelgalt `&` ei ole sama asi mis "töö jääb kindlasti elama".

Kui tead, et terminal võib sulguda, on tavaliselt kolm praktilist varianti:

- kasuta `tmux`-i või `screen`-i
- kasuta `nohup`
- mõnes shellis kasuta `disown`

Kui töö on oluline ja pikk, siis `tmux` on enamasti kõige mõistlikum valik.

`nohup` ja `disown`

`nohup` tähendab umbes "ära katkesta seda tööd lihtsalt sellepärast, et sessioon lõpeb".

Näide:

```
nohup sh -c 'sleep 10; echo "valmis"' > nohup-naide.log 2>&1 &
jobs
```

Siin:

- `nohup` aitab tööl jääda ellu ka siis, kui sessioon kaob
- väljund suunatakse faili, sest terminali ei pruugi enam olemas olla

`disown` on teistsugune tööriist:

- käivitad töö kõigepealt tavaliselt
- siis eemaldad selle shelli tööde nimekirjast

Näiteks:

```
sleep 120 &
jobs
disown %1
jobs
```

Pärast `disown`-i ei halda shell seda tööd enam samamoodi töö numbriga `%1`.

Hea rusikareegel:

- kui vajad püsivat sessiooni, eelista `tmux`-i
- kui vajad lihtsalt "ära tapa seda tööd terminali sulgemisel", siis `nohup` võib aidata
- `disown` on kasulik pigem teadlikule shellikasutajale, mitte esimese valikuna algajale

Päris näide: käivita väike server taustal

Veel praktilisem näide on käivitada väike kohalik teenus:

```
python3 -m http.server 8000 &
jobs
ps aux | grep '[h]ttp.server'
```

Selle töövoo loogika:

- `&` saadab käsu kohe taustale
- `jobs` näitab shelli teadaolevat tööd
- `ps aux | grep '[h]ttp.server'` näitab päris protsessi süsteemi vaates

Nii saad väga hästi aru, et:

- shelli töö
- süsteemi protsess

on seotud, aga mitte täpselt sama asi.

Kui tahad selle töö lõpetada:

```
kill %1
```

või leia PID ja kasuta `kill PID`.

Päris näide: vaata, mis protsess kasutab protsessorit

Kui jooksutad mõnda aktiivset käsku, siis on hea vaadata seda ka `top` abil:

```
top
```

See ei ole "copy-paste tulemus", vaid jälgimise tööriist:

- leia oma protsess
- vaata PID-i
- vaata CPU või mälu kasutust

Kui see saab selgeks, siis muutuvad ka `kill`, `ps` ja logid palju mõtestatumaks.

Minitest

1. Käivita `sleep 120`.
2. Peata see `Ctrl-z` abil.
3. Vaata tööd käsuga `jobs`.
4. Jätka tööd taustal käsuga `bg`.
5. Too see tagasi esiplaanile käsuga `fg`.
6. Käivita kaks taustatööd kujul `sh -c 'sleep 5; echo ...' &` ja `sh -c 'sleep 2; echo ...' &`.
7. Kasuta `wait`, et oodata mõlemad ära.
8. Käivita `python3 -m http.server 8000 &` ja leia see protsess käsuga `ps aux | grep`.

Logid ja teenused

Selles peatükis vaatame, kust otsida logisid ja kuidas vaadata teenuste seisundit, eriti Linuxi süsteemides.

Loogika

Kui mõni teenus ei tööta, siis väga sageli on kaks kõige tähtsamat küsimust:

1. kas teenus üldse töötab

2. mida logid selle kohta ütlevad

Just siin kohtuvad kaks tööriistamaailma:

- teenuse seisund
- logifailid või journal

Kiirspikker

- `systemctl status nimi` näitab teenuse seisu
- `journalctl -u nimi` näitab selle teenuse journal'i
- `journalctl -n 50` näitab viimaseid kirjeid
- `tail -f fail.log` jälgib logifaili reaalajas
- `/var/log` sisaldab paljusid logifaile

Käivita need käsud

Linuxis:

```
systemctl status ssh
journalctl -u ssh -n 50
journalctl -n 50
ls /var/log | head
```

Kui sul ei ole systemd-d või teenuse nimi on teine, siis vaata vähemalt logifaile:

```
tail -n 50 /var/log/syslog
tail -n 50 /var/log/messages
```

Kui tahad viimaseid ridu rahulikult sirvida, siis väga praktiline on:

```
tail -n 50 /var/log/syslog | less
```

Teenus ja protsess ei ole päris sama asi

Teenuse puhul mõtle nii:

- teenus on süsteemi hallatav töö
- protsess on selle töö jooksev eksemplar

See tähendab, et vahel on kasulik vaadata nii teenust kui protsessi, aga alguses tasub teenuse puhul alustada just:

```
systemctl status nimi
```

`systemctl status`

Näide:

```
systemctl status ssh
```

See aitab näha:

- kas teenus on aktiivne
- kas ta käivitub süsteemi stardis
- kas viimastes teadetes paistab mõni viga

Teenuse nimi võib süsteemiti erineda. Näiteks:

- mõnes süsteemis on nimi `ssh`
- mõnes `sshd`

Kui üks ei tööta, proovi teist.

`journalctl`

Kui teenus ei tööta, siis järgmine väga praktiline samm on:

```
journalctl -u ssh -n 50
```

See näitab viimaseid kirjeid just selle teenuse kohta.

Kasulikud variandid:

```
journalctl -u ssh --since today
journalctl -u ssh -f
```

Siin tähendab:

- `--since today` näita tänaseid kirjeid
- `-f` jälgi juurde tulevaid logisid

Logifailid kaustas `/var/log`

Mitte kõik süsteemid ei kasuta journal'it samal viisil. Väga tihti jõuad ka tavaliste logifailideni.

Näited:

- `/var/log/syslog`
- `/var/log/messages`
- teenuse enda logikaust

Logide vaatamiseks on praktilised:

```
tail -n 50 /var/log/syslog
tail -f /var/log/syslog
```

Kui üks fail puudub, proovi teist. Logide nimed ei ole kõigis distributsioonides samad.

macOS-i märkus

macOS-is ei ole süsteemi teenuste maailm päris sama mis systemd-ga Linuxis.

Seal kohtad sagedamini:

- `launchd`
- käske nagu `log show`

Selle raamatu peatükk on teadlikult rohkem Linuxi poole kaldu, sest `systemctl` ja `journalctl` on just seal kõige kesksemad.

Praktiline kontrolljärjekord

Kui teenus ei tööta, siis alusta nii:

1. `systemctl status nimi`
2. `journalctl -u nimi -n 50`
3. vaata, kas logifailis on sama vea jälg
4. kontrolli vajadusel ka võrgupeatükist, kas teenus kuulab õigel pordil

See seob hästi kokku peatükid Võrgu põhitööriistad ja Protsessid, tööd ja signaalid.

Päris näide ilma systemd-ta: näidisfail `app.log`

Kui sul parasjagu ei ole käepärast Linuxi teenust koos `systemctl`-iga, saad sama mõtte treenimiseks kasutada näidisfaili `app.log`:

```
cp data/app.log app.log
tail -n 20 app.log
```

See annab kohe viimased kirjed kätte.

Kui tahad näha ainult vead:

```
grep ' ERROR ' app.log | tail -n 10
```

Kui tahad viimased kirjed enne rahulikult läbi sirvida, siis:

```
tail -n 50 app.log | less
```

Kui tahad vaadata ainult andmebaasi mooduli vigu:

```
grep 'module=db' app.log | grep ' ERROR '
```

See on väga päris töövoog:

- esimene filter valib mooduli
- teine filter valib vea

Päris näide: jälgi logi reaalajas

Ühes terminalis:

```
cp data/app.log app.log
tail -f app.log
```

Teises terminalis lisa paar rida:

```
cat >> app.log <<'EOF'
2026-04-13T21:40:00 ERROR host=tallinn-app module=api user=vilo message="manual test error"
2026-04-13T21:40:02 WARN host=tallinn-app module=api user=vilo message="manual test warning'
EOF
```

See on väga hea harjutus, sest siis näed oma silmaga:

- kuidas logi kasvab
- miks `tail -f` on kasulik
- kuidas logid, protsessid ja teenused päriselus kokku käivad

Päris näide: too logi veebist oma hostilt

Kui paned õpiku andmefailid veebiserverisse, siis võid kasutada ka sellist töövoogu:

```
BASE_URL="https://sinu-domeen/~vilo/linux"
curl -L "$BASE_URL/data/app.log" -o app.log
grep ' ERROR ' app.log | tail -n 10
```

See näitab hästi, kuidas logi:

- tuuakse alla
- salvestatakse faili
- filtreeritakse edasi käsureal

Minitest

1. Vaata mõne tuntud teenuse olekut käsuga `systemctl status`.
2. Vaata sama teenuse viimaseid logikirjeid.
3. Uuri, millised logifailid sinu süsteemis `/var/log` all olemas on.
4. Pane ühe lausega kirja, mis vahe on teenuse seisul ja logidel.
5. Filtreeri `data/app.log` failist välja ainult `ERROR` read.

Püsivad terminalisessioonid: tmux ja screen

Selles peatükis vaatame, kuidas hoida käsurea sessioon elus ka siis, kui SSH-ühendus katkeb või paned sülearvuti kinni.

Loogika

Kaugmasinates ja pikkade tööde puhul on väga tavaline probleem:

- ühendus katkeb
- terminal aken pannakse kinni
- töö jääb pooleli

`tmux` ja `screen` lahendavad selle nii, et shell jääb serveris tööle ka siis, kui sina vahepeal lahkud.

Kiirspikker

- `tmux new -s nimi` alustab uut sessiooni
- `tmux ls` näitab sessioone
- `tmux attach -t nimi` ühendub sessiooniga tagasi
- `screen -S nimi` alustab uut screen-sessiooni
- `screen -ls` näitab olemasolevaid sessioone
- `screen -r nimi` ühendub tagasi

Käivita need käsud

Kui kasutad `tmux`-i:

```
tmux new -s opik
tmux ls
tmux attach -t opik
```

Kui kasutad `screen`-i:

```
screen -S opik
screen -ls
screen -r opik
```

`tmux`

`tmux` on tänapäeval sageli esimene valik, sest ta on paindlik ja hästi levinud.

Tüüpiline töövoog:

1. logi serverisse
2. käivita `tmux new -s nimi`
3. tee oma töö selles sessioonis
4. eemaldu sessioonist, aga ära tapa seda
5. tule hiljem tagasi käsuga `tmux attach -t nimi`

Oluline klahvikombinatsioon:

- `Ctrl-b d` eraldab sind sessioonist, aga jätab selle tööle

See tähendab, et sinu käivitatud protsessid võivad edasi joosta ka siis, kui ühendus katkeb.

`screen`

`screen` on vanem, aga endiselt täiesti kasulik tööriist.

Tema loogika on sama:

- loo sessioon
- tee töö sees
- eemaldu sessioonist

- naase hiljem

Oluline klahvikombinatsioon:

- `Ctrl-a d` eraldab sessioonist

Kumba valida?

Praktiline rusikareegel:

- kui masinas on olemas `tmux`, kasuta enamasti seda
- kui vanemas süsteemis on ainult `screen`, kasuta `screen`-i

Oluline on mitte tööriista nimi, vaid harjumus teha pikad tööd püsivas sessioonis.

Millal see eriti kasulik on

- pikk `rsync`
- pikk build
- andmetöötlus
- logide jälgimine
- serveris töötamine ebastabiilse võrgu pealt

See seostub hästi peatükkidega Kauglogimine ja SSH ja Protsessid, tööd ja signaalid.

`tmux` vs `nohup` vs `disown`

Need tööriistad lahendavad sarnast, aga mitte sama probleemi.

- `tmux` hoiab alles terve shelli sessiooni
- `nohup` aitab ühel käsul jääda ellu ka siis, kui ühendus katkeb
- `disown` eemaldab töö shelli tööde nimekirjast

Praktiline rusikareegel:

- kui tahad hiljem sama shelli juurde tagasi tulla, kasuta `tmux`-i
- kui tahad lihtsalt ühe pika käsu käima jätta, võib aidata `nohup`
- kui juba töötav taustatöö tuleb shellist "lahti siduda", võib abiks olla `disown`

Kui pead valima ühe harjumuse, siis vali `tmux`.

Minitest

1. Uuri, kas sinu masinas on olemas `tmux` või `screen`.
2. Käivita üks sessioon.
3. Eemaldu sellest ilma sessiooni lõpetamata.
4. Ühendu sessiooniga tagasi.
5. Seleta ühe lausega, miks see on kasulik just kaugmasinas.

Graafilised rakendused kaugmasinast

Selles peatükis vaatame, millal kasutada X11-edastust, millal veebiliidest ja mis on praktilised piirangud.

Loogika

Kaugelt graafilise rakenduse kasutamiseks on mitu rada, aga need ei ole võrdselt mugavad. Enamasti tasub eelistada veebiliidest või Remote SSH tüüpi lahendust, ja X11 forwarding jätta erijuhtudeks.

Kiirspikker

- `ssh -X kasutaja@server` proovib X11-edastust üle SSH
- `ssh -L 8888:localhost:8888 kasutaja@server` suunab kaugpordi lokaalsesse masinasse
- veebiliides brauseris on sageli kõige mugavam tee
- Remote SSH arenduseks väldib toorest GUI-edastust

Peamised variandid

- X11 forwarding üle SSH
- veebiliides brauseris
- kaug-töölaua lahendus
- IDE enda Remote SSH tugi

Käivita need käsud

```
ssh -X kasutaja@server
```

Veel üks väga tavaline näide veebiliidese jaoks:

```
ssh -L 8888:localhost:8888 kasutaja@server
```

Pärast seda saab tihti brauseris avada aadressi `http://localhost:8888`.

X11 forwarding

See võib töötada lihtsate X-rakendustega, kuid:

- on sageli aeglane
- vajab kohalikku X-serverit
- ei sobi alati moodsatele GUI-rakendustele

Veebiliides

Sageli on praktilisem kasutada teenuseid, mis töötavad brauseris:

- Jupyter

- veebipõhine adminliides
- kaugserveris jooksev rakendus HTTP kaudu

Minitest

1. Uuri, kas sinu masinal on X11 klient saadaval.
2. Pane kirja üks juhtum, kus veebiliides on mõistlikum kui X11.
3. Selgita, miks Remote SSH võib olla arenduses mugavam kui toores X11 forwarding.

Teksti otsimine: grep ja sugulased

Selles peatükis õpime otsima teksti tööriistadega `grep`, `egrep`, `fgrep` ja moodsamate vastetega.

Loogika

`grep` on seotud torude, failide ja logidega, sest ta võtab ridu sisse ja valib neist välja need, mis sobivad mustriga.

See tähendab, et `grep`-i kasutatakse väga tihti koos:

- failidega
- torudega
- logide ja konfiguratsioonidega

Kõige olulisem mõte on: `grep` ei "tee teksti targemaks", vaid filtreerib ridu.

Kiirspikker

- `grep 'muster' fail.txt` otsib mustrit
- `grep -n` näitab reanumbreid
- `grep -i` eirab tõstutundlikkust
- `grep -r` otsib rekursiivselt
- `grep -v` pöörab vaste ümber

Kõige sagedasemad valikud alguses:

- `-n` reanumbrid
- `-i` tõstutundetu otsing
- `-r` rekursiivne otsing
- `-v` näita mittevastavaid ridu
- `-E` laiendatud regulaaravaldis
- `-F` otsi fikseeritud sõnet, mitte regexit

Käivita need käsud

```
printf 'kass\nkoer\nKass\n' > loomad.txt
grep 'kass' loomad.txt
grep -i 'kass' loomad.txt
grep -n 'koer' loomad.txt
```

```
grep -r 'TODO' .
grep -v '^#' seadistus.conf
```

`grep`, `egrep`, `fgrep`

Ajalooliselt:

- `grep` otsib tavalise mustri järgi
- `egrep` tähistas laiendatud regulaaravaldisi
- `fgrep` tähistas fikseeritud teksti

Tänapäeval kasutatakse sageli:

```
grep -E 'muster'
grep -F 'sone'
```

Praktiline reegel:

- kui otsid lihtsalt täpset teksti, siis `grep -F`
- kui otsid mustrit, siis `grep` või `grep -E`

Kiired töökujud

- `grep 'muster' fail.txt` otsib ühest failist sobivad read
- `grep -n 'muster' fail.txt` lisab väljundisse reanumbrid
- `grep -i 'muster' fail.txt` eirab tõstutundlikkust
- `grep -r 'muster' .` otsib rekursiivselt praegusest kaustast
- `grep -F 'sõne' fail.txt` otsib täpset sõnet ilma regexita
- `grep -R 'muster' .` otsib rekursiivselt ka siis, kui all on alamkaustad

Väga praktiline 1-liner on:

```
grep -R 'TODO' .
```

See on sageli üks kiiremaid viise aru saada:

- kus projektis mingi sõna või märksõna esineb
- kas mõni seadistus, URL või funktsiooninimi on üldse olemas

Päris näide: suur sõnaloend ja mustriotsing

Siin on hea näha, kuidas `grep` on seotud ka teksti puhastamise ja torudega. Mõte ei ole lihtsalt "otsi faili seest", vaid:

1. too andmed alla

2. tee need ühtlaseks
3. otsi huvitavat mustrit

Kui sul on veebis oma andmekaust, on mugav kasutada baas-URL-i:

```
BASE_URL="https://sinu-domeen/~vilo/linux"
curl -L "$BASE_URL/data/words.txt" -o words.txt
```

Kui tahad teha samu katseid ilma veebita, sobib hästi ka repo lokaalne fail:

```
cp data/generated-words.txt words.txt
```

Kui fail on juba kohalikus kaustas olemas, võid alustada otse sellest.

Järgmine samm on teha sõnad väikesteks tähtedeks ja jätta alles ainult read, mis koosnevad tähtedest või numbritest:

```
tr '[:upper:]' '[:lower:]' < words.txt | grep -E '^[[:alnum:]]+$' > words-clean.txt
```

Siin:

- `tr '[:upper:]' '[:lower:]'` teeb sõnad väikesteks tähtedeks
- `grep -E '^[[:alnum:]]+$'` jätab alles ainult need read, kus terve rida koosneb tähtedest või numbritest
- tulemus kirjutatakse faili `words-clean.txt`

Nüüd saab teha huvitavama otsingu:

```
grep -x '..a..t..l.' words-clean.txt
```

Oluline loogika:

- `-x` tähendab, et muster peab katma kogu rea
- `.` tähendab "üks suvaline märk"
- muster `..a..t..l.` otsib 10-märgilisi ridu, kus:
- kolmas märk on `a`
- kuues märk on `t`
- üheksas märk on `l`

Kui vasteid on liiga palju, lisa näiteks:

```
grep -x '..a..t..l.' words-clean.txt | head
```

Või loenda vasteid:

```
grep -x '..a..t..l.' words-clean.txt | wc -l
```

See on hea näide, sest siin saavad kokku:

- `curl` või `wget`
- torud
- `tr`
- `grep -E`
- `grep -x`

See on juba päris töövoog, mitte ainult üksik käsunäide.

Päris terminali transkript

Allpool on tahtlikult veidi "päris elu moodi" terminali näide, mitte täiesti steriilne retsept. See on kasulik, sest näitab ka olukorda, kus esimene otsing ei anna midagi ja alles järgmine samm teeb tulemuse nähtavaks.

```
vilo@Mac tmp % wget https://raw.githubusercontent.com/dwyl/english-words/refs/heads/master/w
...
'words.txt' saved

vilo@Mac tmp % cat words.txt | grep -x 'a..y..l.e'

vilo@Mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a..y..l.e'
abbyville

vilo@Mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a.t.l'
antal
aptal
artal
artel
astel
attal
axtel

vilo@Mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a.t.l' | grep -o .
a
n
t
a
l
a
p
t
a
l
a
r
t
a
l
a
r
t
e
l
a
```

```
s
t
e
l
a
t
t
a
l
a
x
t
e
l

vilo@Mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a.t.l' | grep -o . | sort | uniq -c
  11 a
   8 t
   7 l
   3 e
   2 r
   1 x
   1 s
   1 p
   1 n
```

Selle töövoo loogika on:

- esimene `grep -x 'a..y..l.e'` ei leidnud midagi, sest failis oli vaste suure algustähega
- `tr 'A-Z' 'a-z'` muutis sisendi väiketäheliseks
- pärast seda leidus vaste `abbyville`
- muster `a.t.l` leidis mitu 5-tähelist sõna
- `grep -o .` lõhkus iga vaste üksikuteks märkideks
- `sort | uniq -c | sort -nr` näitas, milliseid tähti nendes vastetes esineb kõige rohkem

See näide on hea ka sellepärast, et ta ühendab mitu peatükki:

- `grep` otsib mustri järgi
- `tr` muudab teksti kuju
- `sort` ja `uniq -c` koondavad tulemuse statistikaks

Praktiline märkus: kui tahad õpetada lihtsalt mustriotsingut, siis võib toru `cat words.txt | ...` asemel kirjutada lühemalt:

```
tr 'A-Z' 'a-z' < words.txt | grep -x 'a..y..l.e'
```

Aga transkriptis on pikem kuju täiesti sobiv, sest ta näitab torude loogikat väga

nähtavalt.

Fun fact: tagasiviited ja korduv muster

GNU `grep` toetab ka tagasiviiteid. See tähendab, et saad öelda: "otsi midagi, kus seesama eelnevalt leitud tükk kordub uuesti".

Näide:

```
vilo@Mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -E '(..)\1\1+'
a.a.a.
aaaaaa
k.k.k.
larararia
logogogue
ratatat
ratatats
ratatat-tat
```

Selle mustri loogika on:

- `(..)` võtab kaks suvalist märki ja jätab need meelde
- `\1` tähendab "sama kahe märgi paar uuesti"
- teine `\1+` tähendab, et see sama paar kordub veel vähemalt ühe korra

Seega otsitakse ridadest kohta, kus mingi kahe märgi paar kordub vähemalt kolm korda järjestikku.

Näited:

- `aaaaaa` sobib, sest `aa` kordub kolm korda
- `a.a.a.` sobib, sest `a.` kordub kolm korda
- `ratatat` sobib, sest reas leidub alammuster `atatat`, kus `at` kordub kolm korda

See viimane on eriti õpetlik, sest siin tuleb välja veel üks oluline asi:

- ilma `-x`-ta otsib `grep` vastet rea seest
- `-x`-ga peab kogu rida mustriga sobima

See tähendab, et:

```
cat words.txt | tr 'A-Z' 'a-z' | grep -E '(..)\1\1+'
```

otsib rea seest sobivaid kohti, aga:

```
cat words.txt | tr 'A-Z' 'a-z' | grep -x -E '(..)\1\1+'
```

nõuab, et terve rida koosneks sellisest korduvast mustrist.

Praktiline märkus: see on hea "regexi mängumaa" näide, aga täiesti igapäevaseks tekstifiltreerimiseks ei ole tagasiviited alati parim tööriist. GNU grep

dokumentatsioon hoiatab, et tagasiviited võivad olla aeglasemad ja mõnikord probleemsemad kui lihtsamad mustrid.

Minitest

1. Loo fail, kus on viis sõna eri ridadel.
2. Otsi üht sõna tõstutundlikult ja siis tõstutundetult.
3. Otsi rekursiivselt sõna `TODO` mõnes projektikaustas.
4. Võta suuremast sõnaloendist ainult väiketähelised alfanumbrilised read ja otsi neist mustriga `grep -x`.
5. Proovi GNU `grep`-iga mustrit `( .. )\1\1+` ilma tühikuteta ja selgita, miks `ratatat` sobib ilma `-x`-ta.

Teksti teisendamine: tr, cut, paste, column, strings

Selles peatükis õpime väikeste käsureatööriistadega teksti ümber kujundama.

Loogika

Need käsud sobivad siis, kui tahad tekstivoo kuju kiiresti muuta ilma pikema skriptita. Need on seotud torude ja otsingupeatükkidega, sest neid kasutatakse sageli kohe pärast `grep`-i või enne `sort`-i.

Kiirspikker

- `tr` asendab või eemaldab märke
- `cut` võtab välja veerge või välju
- `paste` kleebib ridu kõrvuti
- `column` vormib tabeli
- `strings` kuvab binaarfailist loetavad tekstijupid

Käivita need käsud

```
echo 'tere maailm' | tr '[:lower:]' '[:upper:]'
echo 'a,b,c' | tr ',' '\n'
```

```
printf 'nimi:vanus:linn\nMari:20:Tartu\n' > andmed.txt
cut -d ':' -f 1 andmed.txt
cut -d ':' -f 1,3 andmed.txt
```

```
printf 'nimi vanus\nMari 20\nJaan 21\n' | column -t
```

Millal need kasulikud on

- `tr` sobib lihtsaks märgivahetuseks

- `cut` sobib lihtsa eraldajaga väljade võtmiseks
- `column` teeb käsuväljundi loetavamaks
- `strings` aitab uurida tundmatuid binaarfaile

Päris näide: teksti puhastamine ja veergudeks tegemine

See peatükk muutub palju selgemaks siis, kui teha üks päris töövoog näidisandmefailiga.

Alusta nii:

```
cp data/sample-text.txt tekst.txt
head -n 3 tekst.txt
```

Kui tahad tekstist teha sõnade voo, sobib hästi:

```
tr ' ' '\n' < tekst.txt | head -n 20
```

Siin:

- `tr ' ' '\n'` muudab tühikud reavahetusteks
- üks mitmesõnaline rida laguneb sõnade reaks

Kui tahad kõik sõnad suurtähtedeks muuta:

```
tr '[:lower:]' '[:upper:]' < tekst.txt | head -n 3
```

See on hea näide, sest siin ei muudeta "mõtet", vaid ainult märkide kuju.

Päris näide: logirea tükeldamine

Fail `data/app.log` sobib hästi `cut`-i näitamiseks.

```
head -n 5 data/app.log
cut -d ' ' -f 1-4 data/app.log | head -n 5
```

Selle näite loogika:

- eraldajaks on tühik
- `-f 1-4` võtab välja esimesed neli välja
- näed kiiresti aega, logitaset, hosti ja moodulit

Kui tahad ainult logitasemeid:

```
cut -d ' ' -f 2 data/app.log | head -n 10
```

See on hiljem väga kasulik koos `sort` ja `uniq -c`-ga.

Päris näide: tee väljund loetavaks `column` abil

Kui tahad näidata logi lühikokkuvõtet veergudena, saab teha väikese vahefaili:

```
cut -d ' ' -f 1-4 data/app.log | head -n 10 | tr '=' ' ' | column -t
```

Siin toimub korraga mitu asja:

- `cut` võtab logi alguse
- `tr '=' ' '` teeb võtme-väärtuse osad loetavamaks
- `column -t` joondab väljundi veergudesse

See on hea koht, kus terminal hakkab välja nägema juba nagu tabel.

Päris näide: `strings`

`strings` on kõige õpetlikum siis, kui sisend ei ole tavaline tekstifail.

Lihtne näide süsteemi pealt:

```
strings /bin/ls | head -n 20
```

Siin:

- `/bin/ls` on binaarfail
- `strings` üritab sealt leida loetavaid tekstijuppe

See on hea meeldetuletus, et mitte kõik failid ei ole "lihtsalt tekst", isegi kui neist saab mõnikord teksti välja koukida.

UTF-8, täpitähed ja reavahetused

Tekstitöötluses on veel kaks praktilist detaili, mis tulevad väga kiiresti ette:

- kodeering, tavaliselt `UTF-8`
- reavahetuse kuju, tavaliselt `LF` või `CRLF`

Eesti tekstiga on see eriti tähtis, sest `ÕÄÖÜ` ei ole ASCII märgid.

Näide:

```
printf 'Õun\nÄmber\nÖö\nÜks\n' > tahed.txt
cat tahed.txt
```

Kui fail on UTF-8 kujul, näed täpitähti õigesti.

CRLF vs LF

Linuxis ja macOS-is on tavaline reavahetus `LF`. Windowsi failides kohtab tihti `CRLF`.

Näide:

```
printf 'üks\r\nkaks\r\n' > crlf.txt
cat crlf.txt
tr -d '\r' < crlf.txt > lf.txt
```

Siin:

- `\r\n` teeb Windowsi moodi reavahetuse

- `tr -d '\r'` eemaldab carriage return märgid
- tulemuseks saad puhta `LF`-iga faili

Kui mõni tööriist käitub "imelikult", siis põhjus võib olla just reavahetustes, mitte käsus endas.

Minitest

1. Muuda tekst käsuga `tr` suurtähtedeks.
2. Lõika kooloniga eraldatud failist välja teine väli.
3. Vorminda väike tabel `column -t` abil.
4. Võta `data/app.log` failist välja ainult logitase käsuga `cut`.
5. Tee ühest väikesest väljundist veeruline vaade `column -t` abil.

Vood ja tabelid: sort, uniq, wc, pr, join

Selles peatükis vaatame, kuidas ridu loendada, sortida, rühmitada ja tabelilaadseks vormida.

Loogika

Kui sul on palju ridu, aitab see peatükk neist kokkuvõtte teha. Tüüpiline töövoog on: sorteeri, koonda, loenda.

Kiirspikker

- `sort` sorteerib ridu
- `uniq` eemaldab järjestikused duplikaadid
- `uniq -c` loendab järjestikuseid duplikaate
- `wc -l` loendab ridu
- `wc -w` loendab sõnu
- `join` ühendab kahest failist ühise väljaga read
- `pr` vormib väljundi printimiseks või veergudesse

Käivita need käsud

```
printf 'pirn\noun\npirn\nploom\n' > viljad.txt
sort viljad.txt
sort viljad.txt | uniq
sort viljad.txt | uniq -c

echo 'üks kaks kolm neli' | wc -w
printf 'a\nb\nc\n' | wc -l

printf '1 Mari\n2 Jaan\n' > nimed.txt
printf '1 Tartu\n2 Tallinn\n' > linnad.txt
```

```
join nimed.txt linnad.txt
pr -2 viljad.txt
```

Sõnade lugemine ja tõstu muutmine

```
echo 'Tere tere maailm' | tr '[:upper:]' '[:lower:]' | tr ' ' '\n' | sort | uniq -c
```

See on klassikaline Unix-laadne voog: teisenda, jaga ridadeks, sorteeri, loenda.

`join` eeldab tavaliselt, et mõlemad sisendfailid on ühise välja järgi sorditud. `pr` on kasulik siis, kui tahad väljundit kiirelt veergudesse või printimiseks vormida.

Päris näide: kõige sagedamad sõnad

Kui tahad näha, miks `sort | uniq -c | sort -nr` on nii klassikaline, siis kasuta näidisandmefaili:

```
cp data/sample-words.txt sonad.txt
sort sonad.txt | uniq -c | sort -nr | head -n 15
```

Selle töövoo loogika on:

- `sort` toob samad sõnad järjestikku
- `uniq -c` loendab järjestikused kordused
- `sort -nr` paneb suurimad loendused ette

Just see on üks Unix-laadse tekstitöötluse põhivõtteid.

Päris näide: logitasemete kokkuvõte

Fail `data/app.log` sobib hästi väikese logianalüüsi jaoks.

```
cut -d ' ' -f 2 data/app.log | sort | uniq -c | sort -nr
```

Siin:

- `cut -d ' ' -f 2` võtab välja logitaseme
- `sort | uniq -c` loendab tasemed kokku
- tulemuseks saad näiteks `INFO`, `WARN`, `ERROR` sagedused

See on hea näide, sest siin kohtuvad tekstifilter, väljavõte ja koondamine.

Päris näide: palju ridu ja palju sõnu

Kui tahad kiiresti aru saada, kui suur üks tekstifail on:

```
wc -l data/sample-text.txt
wc -w data/sample-text.txt
```

See annab kaks eri mõõdet:

- mitu rida
- mitu sõna

Mõlemad on praktilised, aga nad ei tähenda sama asja.

Päris näide: `join`

`join` tundub alguses natuke kuiv, aga ta on väga hea väikeste tabelite ühendamiseks.

```
printf '1 Tallinn\n2 Tartu\n3 Narva\n' > linnad.txt
printf '1 Harjumaa\n2 Tartumaa\n3 Ida-Virumaa\n' > maakonnad.txt
join linnad.txt maakonnad.txt
```

Siin:

- mõlemal failil on esimene väli ühine võti
- `join` ühendab sama võtmega read kokku

Oluline detail:

- sisendfailid peavad tavaliselt võtme järgi sorditud olema

Päris näide: `pr`

Kui tahad kiirelt sõnaloendit veergudesse panna:

```
head -n 20 data/sample-words.txt | pr -4 -t
```

Siin:

- `-4` teeb neli veergu
- `-t` jätab päise ja jaluse ära

See ei ole tänapäeval kõige sagedasem tööriist, aga väga õpetlik lühikese käsurea vormindusvõttena.

Veel üks väga praktiline näide on nummerdatud loendi panemine mitmesse veergu:

```
seq -w 0 99 | pr -5 -t
```

Siin:

- `seq -w 0 99` teeb loendi `00` kuni `99`
- `pr -5 -t` jagab selle viide veergu

Kui tahad suurema hulga numbreid panna ühele `pr` loogilisele lehele, siis saab mängida lehepikkusega:

```
seq -w 0 9999 | pr -8 -t -l 1250
```

Selle loogika on:

- `seq -w 0 9999` teeb numbrid `0000` kuni `9999`
- `-8` teeb kaheksa veergu
- `-t` eemaldab päise ja jaluse

- `-l 1250` ütleb, et ühe `pr` lehe kõrgus on 1250 rida

Oluline täpsustus:

- see tähendab "üks `pr` leht"
- see ei tähenda automaatselt "üks päris A4 paberileht"

Päris printimisel sõltub tulemus veel fontidest, paberi suurusest ja sellest, kas prindid terminalist, PDF-ist või mõnest muust keskkonnast.

Kui tahad lihtsalt rahulikult mitut veergu eelvaadata, siis väiksem näide on tavaliselt parem:

```
seq -w 0 199 | pr -8 -t | less
```

Locale ja sortimine

`sort` ei tööta alati kõigis keskkondades täpselt ühtemoodi, sest tulemus sõltub ka locale'ist.

See on eriti nähtav täpitähtede puhul.

Näide:

```
printf 'Õun\nÄmber\nÖö\nUdu\n' > tahed.txt
sort tahed.txt
LC_ALL=C sort tahed.txt
```

Siin võib juhtuda, et:

- tavaline `sort` kasutab sinu keskkonna locale'it
- `LC_ALL=C sort ...` sorteerib lihtsama baitide loogika järgi

See tähendab, et `sort` tulemus ei ole alati "absoluutne tõde", vaid sõltub keskkonnast.

Kui töötled eestikeelset teksti, siis tasub seda meeles pidada.

Minitest

1. Loenda faili read.
2. Sorteeri sõnaloend tähestiku järgi.
3. Loenda, mitu korda iga sõna esineb.
4. Proovi `join` abil ühendada kaks väikest faili ühise esimese välja järgi.
5. Tee `data/app.log` failist logitasemete sagedustabel.

sed, awk ja perl praktiliselt

Selles peatükis teeme sissejuhatuse voopõhisesse tekstimuutmisse, väljade töötlemisse ja `perl`-i one-lineritesse.

Loogika

`sed`, `awk` ja `perl` on kasulikud siis, kui lihtsast filtreerimisest enam ei piisa, aga eraldi programmi kirjutamine oleks liiga palju. Need on seotud tekstivoo peatükkidega, sest töötavad peamiselt ridade, väljade ja mustrite peal.

Hea tööjaotus on sageli selline:

- `sed` lihtsateks asendusteks
- `awk` väljade ja veergude töötlemiseks
- `perl` siis, kui vaja on tugevamat regulaaravaldiste loogikat või natuke rikkamat ühe rea programmi

Kiirspikker

- `sed 's/vana/uus/'` asendab esimese vaste real
- `sed 's/vana/uus/g'` asendab kõik vasted real
- `awk '{print $1}'` trükib esimese välja
- `awk -F: '{print $1}'` kasutab koolonit eraldajana
- `perl -pe 's/vana/uus/g'` käib read läbi ja prindib tulemuse välja
- `perl -ne 'print if /muster/'` käib read läbi, aga prindib ainult siis, kui tingimus sobib
- `perl -e '...'` käivitab antud Perl-koodi otse käsurealt

Käivita need käsud

```
echo 'kass koer kass' | sed 's/kass/rebane/'
echo 'kass koer kass' | sed 's/kass/rebane/g'
```

Selle pildi loogika on järgmine:

1. `echo "Tere tere vana kere" > tere.txt` loob algse faili
2. `cp tere.txt kere.txt` teeb samast sisust koopia
3. `sed 's/vana/uus/' tere.txt > mere.txt` loeb faili `tere.txt`, asendab esimese vaste ja kirjutab tulemuse uude faili
4. `head *` näitab kolme faili algust kõrvuti

Oluline tähelepanek on see, et `sed` ei muuda siin algset faili kohapeal. Tulemuse saab uude faili suunata märgiga `>`.

Üks asendus või kõik asendused

`sed`-i üks kõige tähtsamaid erinevusi on see, kas tehakse üks asendus või kõik asendused real.

Selle pildi sees on näha kaks väga erinevat tulemust:

1. `sed 's/ere/ERE/' tere.1.txt > tere.2.txt` muudab ainult rea esimese vaste

```
~/uuskaust/pildid % ls
~/uuskaust/pildid % echo "Tere tere vana kere" > tere.txt
~/uuskaust/pildid % cp tere.txt kere.txt
~/uuskaust/pildid % sed 's/vana/uus/' tere.txt > mere.txt
~/uuskaust/pildid % head *
==> kere.txt <==
Tere tere vana kere

==> mere.txt <==
Tere tere uus kere

==> tere.txt <==
Tere tere vana kere
~/uuskaust/pildid %
```

Joonis 6: Terminali näide, kus fail `tere.txt` kopeeritakse failiks `kere.txt`, seejärel tehakse käsuga `sed 's/vana/uus/'` uus fail `mere.txt`, ja `head *` abil võrreldakse kõigi kolme faili algust.

```
~/uuskaust/pildid % echo "Tere tere vana kere" > tere.1.txt
~/uuskaust/pildid % sed 's/ere/ERE/' tere.1.txt > tere.2.txt
~/uuskaust/pildid % sed 's/ere/ERE/g' tere.1.txt > tere.3.txt
~/uuskaust/pildid % ls
tere.1.txt tere.2.txt tere.3.txt
~/uuskaust/pildid % head *
==> tere.1.txt <==
Tere tere vana kere

==> tere.2.txt <==
TERE tere vana kere

==> tere.3.txt <==
TERE tERE vana kERE
~/uuskaust/pildid %
```

Joonis 7: Terminali näide, kus failis `tere.1.txt` tehakse kõigepealt käsk `sed 's/ere/ERE/'`, mis muudab ainult esimese vaste, ja seejärel `sed 's/ere/ERE/g'`, mis muudab kõik vasted.

2. `sed 's/ere/ERE/g' tere.1.txt > tere.3.txt` muudab kõik vasted samal real
3. `head *` näitab pärast kõiki kolme faili kõrvuti, nii et vahe on kohe nähtav

Rusikareegel on lihtne:

- ilma `g`-ta muudetakse tavaliselt ainult esimene vaste real
- `g` tähendab `global`, ehk kõik vasted sellel real

```
printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1}'
printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1, $2}'
```

```
echo 'kass koer kass' | perl -pe 's/kass/rebane/g'
printf 'Mari\nJaan\n' | perl -ne 'print if /Ja/'
printf 'Mari:20\nJaan:21\n' | perl -F: -lane 'print $F[0]'
```

`perl` one-linerite loogika

Kui kirjutad:

```
perl -pe 's/kass/rebane/g'
```

siis:

- `-e` tähendab, et kood tuleb käsurealt
- `-p` tähendab, et Perl loeb sisendi rida-realt läbi ja prindib iga rea vaikimisi välja

See teeb `perl -pe` kuju väga heaks voopõhisteks asendusteks.

Kui kirjutad:

```
perl -ne 'print if /Ja/'
```

siis:

- `-n` tähendab, et Perl käib read küll läbi, aga ei prindi neid automaatselt
- sina otsustad ise, millal `print` teha

See kuju on kasulik siis, kui tahad teha väikest tingimusloogikat.

Fun fact: algarvud regexiga

See järgmine näide ei ole kõige praktilisem viis algarvude leidmiseks, aga ta on väga hea näide sellest, kui veidralt võimsad võivad `perl`-i one-linerid olla:

```
seq 2 200 | perl -nlE 'say if ("a" x $_) !~ /^(aa+)\1+$/'
```

See kuju on siin meelega võimalikult lihtne. Kuna alustame vahemikku `2`-st, ei pea me eraldi `0` ja `1` juhtumeid regexis välja filtreerima.

Mõte on selline:

- iga arv teisendatakse ajutiselt ühe tähe korduseks, näiteks `7` muutub kujule `aaaaaaa`
- regex proovib leida, kas see rida koosneb mingist väiksemast plokist, mida saab mitu korda korrata
- kui saab, siis arv ei ole algarv
- kui ei saa, siis prinditakse arv välja

See tähendab sisuliselt: "prindi ainult need arvud, mida ei saa kirjutada kujul `m * n`, kus mõlemad on suuremad kui `1`."

Praktilises skriptis oleks tavaliselt mõistlikum kasutada tavapärast jaguvuskontrolli, aga ühe rea triki või loengu-näite jaoks on see väga meeldejääv.

Millal mida kasutada

- `sed` sobib lihtsaks voopõhiseks asenduseks
- `awk` sobib väljade ja ridade töötlemiseks
- `perl` sobib keerukamaks mustriloogikaks, mitmeks asenduseks või natuke rikkamaks ühe rea programmiks

Näiteks:

- kui tahad lihtsalt sõna välja vahetada, siis `sed` on sageli kõige loetavam
- kui tahad võtta teisest veerust väärtuse, siis `awk` on sageli kõige loomulikum
- kui tahad korraga filtreerida, asendada ja kasutada tugevamaid regex'e, siis `perl` võib olla kõige mugavam

Kui töö kasvab keerukaks, võib mõnikord olla selgem kasutada Pythonit. Aga väikeste ühekordsete ülesannete jaoks on `sed`, `awk` ja `perl` väga tugevad.

Minitest

1. Asenda reas üks sõna teisega.
2. Võta välja kooloniga eraldatud faili esimene väli.
3. Prindi `awk` abil ainult teine veerg.
4. Filtreeri `perl -ne` abil välja ainult need read, kus on kindel muster.
5. Tee `perl -pe` abil globaalne asendus kogu sisendi ulatuses.

`find` ja `xargs` ohutumalt

Selles peatükis vaatame, kuidas kasutada `find`-i ja `xargs`-i nii, et tühikud ja keerulised failinimed ei lõhuks tulemust ära.

Loogika

`find` leiab teed.

`xargs` annab need teed järgmisele käsule edasi.

Probleem tekib siis, kui failinimedes on:

- tühikud
- tabulaatorid
- reavahetused

Kui kasutada naiivset kuju, siis võib üks failinimi laguneda mitmeks tükiks. Sellepärast on oluline paar:

- `find ... -print0`
- `xargs -0`

Kiirspikker

- `find . -type f` leiab failid
- `find . -name '*.txt'` leiab nime järgi
- `find ... -print0` väljastab nullmärgiga eraldatud tulemused
- `xargs -0` loeb nullmärgiga eraldatud sisendi õigesti sisse
- `find ... -exec käsk {} +` on sageli lihtne alternatiiv `xargs`-ile

Käivita need käsud

```
find . -type f -name '*.txt'
find . -type f -name '*.txt' -print0 | xargs -0 wc -l
find . -type f -name '*.log' -exec ls -lh {} +
```

Kui tahad näha just keeruliste nimede loogikat, tee väike näide:

```
mkdir -p find-naide
cd find-naide
touch 'üks fail.txt' 'teine fail.txt'
find . -type f -name '*.txt' -print0 | xargs -0 ls -l
```

Miks `-print0` ja `-0` tähtsad on

Vaikimisi eraldatakse käsurea tekst sageli tühikute ja reavahetuste järgi. See tähendab, et fail nimega:

```
minu fail.txt
```

võib naiivses töövoos käituda nagu kaks eri sõna.

Selle vältimiseks:

- `find -print0` eraldab tulemused nullmärgiga
- `xargs -0` loeb just seda vormi

See on praktiline “ohutu vaikevariant”, kui liigud `find`-ist järgmise käsuni.

`xargs` või `-exec`?

Mõlemad on kasulikud.

Näide `xargs`-iga:

```
find . -type f -name '*.txt' -print0 | xargs -0 wc -l
```

Näide `-exec`-iga:

```
find . -type f -name '*.txt' -exec wc -l {} +
```

Rusikareegel:

- kui tahad lihtsat ja ohutut kuju, siis `-exec ... +` on sageli väga hea
- kui tahad teadlikult ehitada torupõhist töövoogu, siis `-print0 | xargs -0` on tugev valik

`xargs` vs `while read`

On veel üks väga kasulik kuju:

```
find data -type f -name '*.txt' -print0 |
while IFS= read -r -d '' fail; do
  wc -l "$fail"
done
```

Selle loogika on:

- `find -print0` annab failinimed ohutult edasi
- `read -r -d ''` loeb ühe nullmärgiga eraldatud faili korraga
- tsüklis saad iga faili kohta teha rohkem kui ühe sammu

`xargs` on väga hea siis, kui:

- tahad ühe käsu kiiresti paljudele failidele anda

`while read` on eriti hea siis, kui:

- tahad iga faili kohta teha väikese loogika
- vajad tsüklit, tingimust või mitut käsku järjest

Näiteks:

```
find data -type f -name '*.txt' -print0 |
while IFS= read -r -d '' fail; do
  echo "Töötlen: $fail"
  head -n 1 "$fail"
done
```

Väldi pimesi hävitavaid käske

`find` ja `xargs` lähevad eriti ohtlikuks siis, kui lõppu pannakse:

- `rm`
- `chmod`
- `mv`

Seetõttu tasub alati enne teha kuivem kontroll:

```
find . -type f -name '*.log'
```

ja alles siis panna lõppu päris tegevus.

Veel parem on alustada mittelahutava käsuga nagu:

```
find . -type f -name '*.log' -print0 | xargs -0 ls -lh
```

Tüüpilised kasutused

- leia kõik kindla laiendiga failid
- anna need failid edasi `wc`, `grep`, `ls` või mõnele skriptile
- väldi failinimede lõhkumist tühikute peal

See peatükk seostub hästi teemadega Teksti otsimine: grep ja sugulased ja Esimene shelliskript.

Päris näide: loenda kõik `data/` tekstifailid kokku

Repo `data/` kaust on hea turvaline koht `find`-i harjutamiseks.

```
find data -type f -name '*.txt' -print0 | xargs -0 wc -l
```

Siin:

- `find` leiab kõik `.txt` failid
- `-print0` eraldab need ohutult
- `xargs -0 wc -l` annab igale failile reaarvu

See on hea näide, sest siin ei ole vaja midagi kustutada ega muuta, ainult vaadata.

Päris näide: keeruliste nimedega failid

Kui tahad näha, miks `-print0` ja `-0` on päriselt vajalikud, tee selline väike töökaust:

```
mkdir -p otsi-naide/'Tallinn andmed'
cp data/sample-words.txt 'otsi-naide/Tallinn andmed/sonad 1.txt'
cp data/sample-text.txt 'otsi-naide/Tallinn andmed/tekst 2.txt'
find otsi-naide -type f -name '*.txt' -print0 | xargs -0 ls -lh
```

Selle töövoo mõte on:

- failinimedes on tühikud
- naiivne `xargs` võiks need lõhkuda

- `-print0 | xargs -0` hoiab need tervikuna koos

Päris näide: ohutu alternatiiv `-exec`

Sama töö saab sageli teha ka ilma `xargs`-ita:

```
find otsi-naide -type f -name '*.txt' -exec wc -l {} +
```

See on sageli kõige lihtsam kuju siis, kui:

- tahad ühe käsu kõikidele tulemustele rakendada
- ei taha mõelda sisendi eraldamisele eraldi torus

Päris näide: leia logifail ja vaata selle suurust

```
find data -type f -name '*.log' -exec ls -lh {} +
```

See on väike, aga päris praktiline muster:

- leia failid
- ära hävita midagi
- vaata enne suurust või arvu

Just nii peakski `find`-iga töötamine algama.

Minitest

1. Loo vähemalt üks fail, mille nimes on tühik.
2. Leia see fail `find`-iga.
3. Näita seda faili `ls -l` abil läbi töövoo `-print0 | xargs -0`.
4. Seleta ühe lausega, miks `-print0` ja `-0` koos käivad.
5. Loenda `data/` kausta tekstifailide read käsuga `find ... -print0 | xargs -0 wc -l`.
6. Tee sama töövoog uuesti kujul `while IFS= read -r -d '' fail; do ...; done`.

Esimene shelliskript

Selles peatükis teeme väga väikese esimese shelliskripti ja vaatame, mida tähendavad shebang, `chmod +x`, argumendid, `if`, `for` ja exit code.

Loogika

Shelliskript on lihtsalt tekstifail, kus on järjest käsud, mida shell käivitab.

Esimese skripti eesmärk ei ole veel teha midagi keerulist. Piisab sellest, kui saad kätte viis põhiasja:

1. kuidas skript algab
2. kuidas ta käivitatavaks teha

3. kuidas talle argumente anda
4. kuidas teha lihtne tingimus
5. kuidas tagastada edu või viga

Kiirspikker

- `#!/usr/bin/env bash` valib Bashi
- `chmod +x fail.sh` teeb faili käivitatavaks
- `$1` tähendab esimest argumenti
- `"$@"` tähendab kõiki argumente
- `if ... fi` teeb tingimusloogika
- `for ... do ... done` kordab tegevust
- `exit 0` tähendab edu, `exit 1` tähendab viga

Käivita need käsud

```
mkdir -p skripti-naide
cd skripti-naide
cat > tervita.sh <<'EOF'
#!/usr/bin/env bash
if [ $# -eq 0 ]; then
  echo "Kasuta: $0 nimi..." >&2
  exit 1
fi
for nimi in "$@"; do
  echo "Tere, $nimi!"
done
EOF
chmod +x tervita.sh
./tervita.sh Mari Jaan
./tervita.sh
echo $?
```

Skripti loomine heredoc-iga

Skripti ei pea alati kokku panema pika `printf` käsuga. Väga tavaline ja loetav viis on kasutada here-doc'i.

Näide:

```
cat > tere.sh <<'EOF'
#!/bin/sh
echo "Tere skriptist"
pwd
EOF
```

Selle loogika on:

- `cat > tere.sh` kirjutab väljundi faili `tere.sh`
- `<<'EOF'` tähendab, et järgmised read lähevad faili kuni reale `EOF`
- jutumärkides `EOF` hoiab ära selle, et shell hakkaks neid ridu juba loomise hetkel ise laiendama

See on väga mugav, kui skript on juba mitmerealine.

`sh skript.sh`, `bash skript.sh`, `zsh skript.sh` ja `./skript.sh`

See on üks kõige tähtsamaid erinevusi shelliskriptide juures.

Need käsud ei tähenda päris sama asja:

```
sh skript.sh
bash skript.sh
zsh skript.sh
./skript.sh
```

Praktiline loogika on:

- `sh skript.sh` käsib fail sisu tõlgendada shellil `sh`
- `bash skript.sh` käsib fail sisu tõlgendada Bashil
- `zsh skript.sh` käsib fail sisu tõlgendada Zsh-l
- `./skript.sh` kasutab skripti shebang-rida

See tähendab väga tähtsat asja:

- kui käivitad `bash skript.sh`, siis otsustab tõlgendaja sina käsureal
- kui käivitad `./skript.sh`, siis otsustab tõlgendaja skripti esimene rida

Seepärast ei ole shebang ainult "ilus esimene rida", vaid päris käitumise osa.

Väike võrdlusnäide

Loo kõigepealt väga lihtne skript:

```
cat > naide.sh <<'EOF'
#!/bin/sh
echo "shell: $0"
echo "pwd: $(pwd)"
EOF
```

Nüüd proovi:

```
sh naide.sh
bash naide.sh
zsh naide.sh
chmod +x naide.sh
./naide.sh
```

Kõik need võivad selle lihtsa skripti puhul töötada, sest skript kasutab väga tavalist ja ühilduvat süntaksit.

See on hea esimene õppetund:

- lihtne POSIX-laadne skript töötab sageli mitmes shellis
- keerulisemad Bashi või Zsh eripärad enam mitte

Millal `sh` ja millal `bash`

Hea rusikareegel on:

- kui skript kasutab ainult lihtsat ja laialt ühilduvat süntaksit, sobib `#!/bin/sh`
- kui skript kasutab Bashi erisusi, siis kasuta `#!/usr/bin/env bash`

Näiteks Bashi-spetsiifiline on sageli:

- `[[ ... ]]`
- massiivid
- mõni mugavam parameetrilaiendus

Näide, mis töötab Bashis, aga mitte tingimata `sh`-s

```
cat > bash-only.sh <<'EOF'
#!/usr/bin/env bash
nimi="Mari"
if [[ $nimi == M* ]]; then
  echo "See on Bashi [[ ]] naide"
fi
EOF
```

Käivita:

```
bash bash-only.sh
chmod +x bash-only.sh
./bash-only.sh
```

Kui proovid sama käsuga:

```
sh bash-only.sh
```

siis võib see anda vea, sest `sh` ei pruugi Bashi süntaksit toetada.

Just siin tuleb shebang ja õige tõlgendaja valik päriselt mängu.

Shebang

Skripti esimene rida:

```
#!/usr/bin/env bash
```

ütleb, millise tõlgendiga see fail käivitada.

See on põhjus, miks sama tekstifail võib käituda skriptina, mitte lihtsalt tavalise tekstina.

Kui käivitad faili kujul:

```
./skript.sh
```

siis süsteem ei "arva niisama", et see on Bash või Zsh. Ta vaatab faili algust.

Kui esimene rida on näiteks:

```
#!/bin/sh
```

siis püütakse fail käivitada shelliga `sh`.

Kui esimene rida on:

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

siis püütakse fail käivitada Perliga.

See tähendab:

- täitmisõigus ütleb, et faili tohib käivitada
- shebang ütleb, millega seda käivitada

Mõlemat on sageli vaja korraga.

Näide: sama idee teise tõlgendajaga

Skript ei pea olema ainult shelliskript. Sama loogika töötab ka teiste tõlgendatud keeltega.

Näide:

```
cat > tere.pl <<'EOF'
#!/usr/bin/env perl
print "Tere Perl-ist\n";
EOF
chmod +x tere.pl
./tere.pl
```

Siin:

- fail on tavaline tekstifail
- täitmisõigus lubab selle käivitada
- shebang ütleb süsteemile, et faili peab lugema Perl

Kui Perl puudub, siis ei saa faili käivitada, isegi kui `chmod +x` on tehtud.

`chmod +x`

Faili sisu üksi ei tee sellest veel käivitatavat skripti.

Selleks on vaja:

```
chmod +x tervita.sh
```

Pärast seda saad käivitada:

```
./tervita.sh Mari
```

Kui täitmisõigust ei ole, siis võid testi jaoks käivitada ka nii:

```
bash tervita.sh Mari
```

Või:

```
sh tervita.sh Mari
zsh tervita.sh Mari
```

Aga siis valid shelli sina käsureal, mitte skript ise shebang-reaga.

Argumendid

Kui kirjutad:

```
./tervita.sh Mari Jaan
```

siis:

- `$1` on `Mari`
- `$2` on `Jaan`
- `"$@"` tähendab kõiki argumente koos

Esimese skripti juures on `"$@"` väga kasulik, sest saad ühe korraga läbi käia kõik antud nimed.

`if`

Selles skriptis on tingimus:

```
if [ $# -eq 0 ]; then
```

Selle mõte on:

- kui argumente ei antud
- siis näita kasutusjuhendit
- ja lõpeta veakoodiga

See on väga tavaline muster.

`for`

Tsükkel:

```
for nimi in "$@"; do
  echo "Tere, $nimi!"
done
```

käib kõik argumendid ükshaaval läbi.

See on esimese skripti juures hea näide, sest seal on kohe näha, kuidas üks väike tööriist saab mitut sisendit töödelda.

Exit code

`exit code` on väga tähtis, sest see ütleb teistele programmidele ja shellile, kas töö õnnestus.

Tavaline rusikareegel:

- `0` tähendab edu
- muu arv tähendab viga

Selles skriptis:

- ilma argumentideta lõpetab ta `exit 1`
- argumentidega lõpetab ta edukalt

Pärast käsu jooksutamist saad viimast koodi vaadata nii:

```
echo $?
```

Minitest

1. Tee oma skript, mis tervitab ühte või mitut nime.
2. Muuda see käivitatavaks.
3. Käivita skript nii argumentidega kui ilma argumentideta.
4. Vaata `echo $?` abil, mis exit code jäi viimase käigu järel.
5. Seleta ühe lausega, mis roll on shebang'il.
6. Loo üks skript here-doc'i abil kujul `cat > skript.sh <<'EOF'`.
7. Proovi vahet `sh skript.sh` ja `./skript.sh` vahel.

`cron` ja ajastatud tööd

Selles peatükis vaatame, kuidas panna lihtne käsk regulaarselt käima.

Loogika

Ajastatud töö tähendab, et käsk jookseb kindlal ajal ilma selleta, et peaksid ise terminalis kohal olema.

See on kasulik näiteks siis, kui tahad:

- teha regulaarset varukoopiat
- kirjutada logisse mõõtmist
- käivitada puhastus- või sünkroonkäigu

Alguses piisab täiesti sellest, kui mõistad kolme asja:

1. kus ajastus kirjas on
2. et kasutada tuleb sageli täisradasid
3. et väljund tasub logifaili suunata

Kiirspikker

- `crontab -l` näitab sinu croni ridu
- `crontab -e` avab cron-tabeli muutmiseks
- viis ajavälja tähendavad minut, tund, kuupäev, kuu ja nädalapäev
- `*/15 * * * * käsk` tähendab “iga 15 minuti järel”

Käivita need käsud

Kõige ohutum esimene samm on lihtsalt vaadata olemasolevat croni:

```
crontab -l
```

Näidisrea võid kõigepealt kirjutada faili:

```
cat > naide.cron <<'EOF'
*/15 * * * * /bin/date >> "$HOME"/cron-naide.log 2>&1
EOF
cat naide.cron
```

Kui tahad selle päriselt paigaldada, siis järgmine samm oleks:

```
crontab naide.cron
crontab -l
```

Cronirea kuju

Lihtne cronirida näeb välja nii:

```
*/15 * * * * /bin/date >> "$HOME"/cron-naide.log 2>&1
```

Väljad vasakult paremale:

1. minut
2. tund
3. kuupäev
4. kuu
5. nädalapäev
6. käsk

Näite tähendus on:

- iga 15 minuti järel
- käivita `/bin/date`
- lisa väljund faili `cron-naide.log`

Miks täisrajad tähtsad on

Croni keskkond on tavaliselt palju väiksem kui sinu tavaline interaktiivne shell.

See tähendab, et käsk, mis töötab terminalis nii:

```
date
```

tasub cronis kirjutada pigem nii:

```
/bin/date
```

Sama kehtib sageli ka skriptide, Pythoni ja teiste tööriistade kohta.

Väljundi suunamine

Kui cron töötab taustal, siis on väga kasulik suunata väljund faili:

```
>> "$HOME"/cron-naide.log 2>&1
```

See tähendab:

- lisa tavaline väljund faili lõppu
- lisa samasse faili ka veaväljund

Kui töö ei käi ootuspäraselt, on see logifail esimene koht, kust vaadata.

Testi käsku enne käsitsi

Väga hea reegel on:

enne kui paned käsu croni, käivita ta käsitsi täpselt samas kujus.

Näiteks:

```
/bin/date >> "$HOME"/cron-naide.log 2>&1
tail -n 5 "$HOME"/cron-naide.log
```

Kui see ei tööta käsitsi, ei tööta see tõenäoliselt ka cronis.

Linux ja macOS

Croni põhimõte on sarnane, aga tänapäeva süsteemides on olemas ka teised ajastusmehhanismid:

- Linuxis kohtad sageli ka systemd timereid
- macOS-is on loomulikum süsteem `launchd`

Sellegipoolest on `cron` väga hea esimene mudel, mille pealt ajastatud töid mõista.

Minitest

1. Vaata, kas sinu kasutajal on juba mõni cronirida olemas.
2. Kirjuta üks näidisrea fail, mis käivitaks käsu iga 15 minuti järel.
3. Käivita sama käsk enne käsitsi.
4. Seleta ühe lausega, miks täisrada on cronis tähtis.

Git, GitHub ja töövoog

Selles peatükis vaatame kõigepealt, mis asi on versioonihaldus, siis Git-i põhikäske ja alles pärast seda GitHubi.

Loogika

Git-i põhimõte ei ole lihtsalt "saada failid GitHubi". Loogika on tavaliselt:

1. tööta lokaalselt
2. salvesta muutused commit'idena
3. sünkroniseeri need kaugserveriga

See on põhjus, miks Git ja GitHub tuleb lahus hoida:

- Git on lokaalne versioonihaldus
- GitHub on koht, kus seda repot jagada ja arutada

Mis on versioonihaldus

Versioonihaldus tähendab, et faili või projekti muutused talletatakse ajalooks.

Praktiline kasu on selles, et saad:

- näha, mis muutus
- minna mõttes või käsuga tagasi eelmise seisu juurde
- eristada üksteisest loogilisi muudatusi
- jagada sama projekti teistega

Ilma selle mõtteta jäävad ka Git-i käsud kergesti lihtsalt mehaaniliseks loeteluks.

Kiirspikker

- `git status` näitab tööpuu seisu
- `git add` lisab muudatused stage'i
- `git commit -m '...'` teeb commit'i
- `git pull` toob muudatused
- `git push` saadab muudatused serverisse

Kõige tavalisemad käsud alguses:

- `git status`
- `git diff`

- `git add fail`
- `git commit -m '...'`
- `git log --oneline --graph --decorate`

Käivita need käsud

```
mkdir -p ~/tmp/git-naide
cd ~/tmp/git-naide
git init
printf 'esimene rida\n' > naide.txt
git status
git add naide.txt
git commit -m 'Lisa naidefail'
git log --oneline
```

Kui repol on juba GitHubi aadress ja sa tahad selle enda masinasse tõmmata, siis tüüpiline algus on:

```
git clone git@github.com:kasutaja/projekt.git
cd projekt
git status
```

```
git diff
git log --oneline --graph --decorate -n 10
```

Kõige tavalisem töövoog

Kui töötad igapäevaselt, siis see katab suure osa vajadusest:

```
git status
git pull --rebase
git checkout -b parandus
```

tee muudatused, siis:

```
git diff
git add fail1 fail2
git commit -m 'Paranda näited peatükis SSH'
git push -u origin parandus
```

Selle loogika tugevus on:

- enne tõmbad värske seisu
- töötad eraldi harus
- vaatad muudatused üle enne commit'i

Mis on haru

`haru` ehk `branch` on eraldi töölõng sama projekti sees.

Kõige tavalisem mõtteviis on:

- `main` või `master` on põhirida
- väike parandus või uus mõte tehakse eraldi harus
- hiljem saab selle põhireaga ühendada

Sellepärast on käsk:

```
git checkout -b parandus
```

tegelikult kaks asja korraga:

- `git checkout` liigub teise haru peale
- `-b` ütleb, et see haru luuakse kohe samal ajal

`add`, `commit`, `pull`, `push` omavahel

- `git add` ütleb, millised muudatused lähevad järgmisse commit'i
- `git commit` salvestab selle loogilise muudatuse lokaalselt
- `git pull` toob teised muudatused sinu masinasse
- `git push` saadab sinu commit'id serverisse

See järjekord aitab vältida tunnet, et Git on "müstiline".

Miks `git diff` on üks tähtsamaid käske

Kui `git status` ütleb, et midagi on muutunud, siis `git diff` näitab, mis täpselt muutus.

See on väga oluline, sest Git-i juures ei piisa ainult teadmisest:

- "fail muutus"

Sageli on vaja näha:

- mis read muutusid
- kas muudatus on loogiline
- kas midagi läks kogemata kaasa

Hea tööharjumus on:

1. tee muudatus
2. vaata `git diff`
3. alles siis tee `git add`

Kõige tavalisemad `diff`-i kujud

Need kolm katavad suure osa igapäevavajadusest:

```
git diff
git diff --cached
git diff HEAD
```

Nende loogika on:

- `git diff` näitab veel stage'imata muudatusi
- `git diff --cached` näitab seda, mis on juba stage'is ja läheks järgmisse commit'i
- `git diff HEAD` näitab kõiki muudatusi võrreldes viimase commit'iga

See on üks Git-i parimaid "aha" kohti, sest siis saad aru, et tööpuul, stage'il ja commit'il on eri roll.

Väike praktiline näide

```
printf 'esimene rida\n' > naide.txt
git add naide.txt
git commit -m 'Lisa naidefail'
printf 'teine rida\n' >> naide.txt
git diff
```

Siis näed, et `git diff` näitab ainult viimast muutust võrreldes viimase commit'iga.

Kui nüüd teha:

```
git add naide.txt
git diff --cached
```

siis näed juba seda versiooni, mis on valmis commit'i minema.

Kuidas muudatus tagasi võtta ilma paanikata

See on üks Git-i kõige praktilisemaid osi.

Kui tahad tööpuus oleva muudatuse tagasi võtta:

```
git restore naide.txt
```

Kui tahad faili stage'ist maha võtta, aga mitte selle sisu ära kaotada:

```
git restore --staged naide.txt
```

See tähendab:

- `git restore fail` muudab tööpuud
- `git restore --staged fail` muudab stage'i

Need on väga head käsud just selleks, et mitte sattuda kohe raskemate või hävitavamate käskude juurde.

`.gitignore`

Kõiki faile ei tasu Git-i panna.

Tüüpilised näited:

- `.venv/`

- `__pycache__/`
- `dist/`
- ajutised logifailid

Lihtne näide:

```
cat > .gitignore <<'EOF'
.venv/
__pycache__/
dist/
*.log
EOF
```

Põhimõte on:

- kui fail on genereeritud, ajutine või masinapõhine, siis ta ei peaks tavaliselt versioonihalduses olema

Kui fail on juba Git-i lisatud, siis ainult `.gitignore` hiljem üksi ei eemalda teda automaatselt repost.

Kuidas `diff`-i lugeda

Alguses piisab väga lihtsast mõtteviisist:

- rida märgiga `-` tähendab eemaldatud rida
- rida märgiga `+` tähendab lisatud rida
- ülejäänud read annavad ümbruse ehk konteksti

Näide:

```
-vana rida
+uus rida
veel üks uus rida
```

See tähendab:

- üks vana rida eemaldati
- asemele tuli uus sisu

Sa ei pea alguses kogu diff-formaati detailideni teadma. Tähtis on, et oskad enne commit'i muudatuse sisuliselt üle vaadata.

Miks see GitHubi jaoks ka tähtis on

GitHubi pull request'i keskne vaade on sisuliselt diff.

See tähendab:

- kui teed lokaalselt `git diff`-i vaatamise harjumuseks
- siis on ka GitHubi review palju loogilisem

Hea rusikareegel on:

- enne `git add`: `git diff`
- enne `git commit`: `git diff --cached`
- enne pull request'i: veel kord kontrolli kogu muudatust

Päris näide: väike repo nullist

Kui tahad Git-i töövoogu rahulikult harjutada ilma päris projekti puutumata, tee väike testirepo:

```
mkdir -p ~/tmp/git-naide
cd ~/tmp/git-naide
git init
printf 'esimene rida\n' > naide.txt
git add naide.txt
git commit -m 'Lisa naidefail'
printf 'teine rida\n' >> naide.txt
git diff
```

Seejärel:

```
git add naide.txt
git diff --cached
git commit -m 'Lisa teine rida'
git log --oneline --graph --decorate
```

See on hea "päris näide", sest siin saad ühe väikese faili peal läbi teha:

- tööpuu muutuse
- diff'i
- stage'i
- commit'i
- ajaloo

Päris näide: uus haru väikese paranduse jaoks

```
git checkout -b parandus
printf 'kolmas rida\n' >> naide.txt
git diff
git add naide.txt
git commit -m 'Lisa kolmas rida'
git log --oneline --graph --decorate -n 5
```

Siin on väga hästi näha, miks haru on kasulik:

- saad teha ühe loogilise paranduse eraldi
- ajalugu jääb selgem
- hiljem on seda lihtsam GitHubi pushida või pull request'iks teha

Päris näide: `restore` ja `.gitignore`

```
printf 'ajutine rida\n' >> naide.txt
git diff
git restore naide.txt
printf 'testlogi\n' > debug.log
cat > .gitignore <<'EOF'
debug.log
EOF
git status
```

Siin:

- `git restore naide.txt` võtab tööpuu muudatuse tagasi
- `.gitignore` aitab vältida seda, et ajutine fail üldse Git-i satuks

See on palju rahulikum töövoog kui pimesi "proovin midagi reset'ida".

Mõistlikud alias'ed

Kui kasutad Git-i palju, võivad need olla head:

```
alias gs='git status'
alias ga='git add'
alias gd='git diff'
alias gdc='git diff --cached'
alias gc='git commit'
alias gp='git pull'
alias gph='git push'
alias gl='git log --oneline --graph --decorate'
```

Need võiks panna samasse shelli config-faili, kus `ll`.

GitHubi seos SSH-ga

Kui kasutad GitHubi aadressi kujul:

```
git clone git@github.com:kasutaja/projekt.git
```

siis kasutab Git taustal SSH-d. Seetõttu on GitHubi töövoog tihedalt seotud peatükiga SSH ja võtmed.

GitHubi töövoog lühidalt

1. tee repo koopia või klooni see
2. loo eraldi haru
3. tee muudatused
4. commit'i väikeste loogiliste sammudena
5. push'i haru GitHubi
6. ava pull request

Minitest

1. Vaata käsu `git status` väljundit mõnes repos.
2. Uuri, mis vahe on `git pull` ja `git fetch`.
3. Pane kirja, miks on kasulik töötada eraldi harus.
4. Käivita `git log --oneline --graph --decorate -n 5` mõnes repos.
5. Muuda üht faili, vaata `git diff`, seejärel stage'i see ja vaata `git diff --cached`.

Pythoni venv ja eraldatud keskkonnad

Selles peatükis selgitame, miks `venv` vajalik on ja kuidas seda kasutada.

Loogika

`venv` ei ole Pythoni juures lihtsalt järjekordne rituaal, vaid väga praktiline viis hoida projektid üksteisest lahus.

Põhiküsimus ei ole "kuidas teha `.venv`", vaid:

- miks mitte paigaldada kõike lihtsalt süsteemi
- miks üks projekt töötab ja teine enam mitte
- miks IDE, terminal ja `pip` peavad nägema sama Pythoni keskkonda

See on seotud paketihalduse ja IDE peatükkidega, sest:

- `pip` paigaldab paketid
- `venv` määrab, kuhu need paigaldatakse
- IDE peab kasutama sama keskkonda

Miks `venv` vajalik on

Pythoni paketid võivad eri projektides vajada erinevaid versioone. Virtuaalkeskkond hoiab projektisõltuvused eraldi.

Kõige tavalisemad päris probleemid ilma `venv`-ta on:

- projekt A tahab `requests` ühte versiooni, projekt B teist
- `pip install ...` paigaldab paketi "kuhugi", aga hiljem ei saa aru, millise Pythoniga see seotud on
- IDE kasutab üht interpreterit, terminal teist
- süsteemi Python või muud tööriistad saavad kogemata sinu katsetustest mõjutatud

Lühidalt:

- `venv` aitab vältida segadust
- `venv` teeb projektid korratavamaks
- `venv` teeb veaotsingu lihtsamaks

Mis `venv` tegelikult teeb

`venv` loob projektikausta sisse eraldi Pythoni keskkonna, kus on:

- oma `python`
- oma `pip`
- oma installitud paketid

Tüüpiline kaust on:

```
.venv/
```

Oluline mõte on:

- süsteemi Python jääb alles oma kohale
- projekt kasutab omaenda koopiat või viidet Pythoni keskkonnale
- kui oled selle keskkonna aktiveerinud, siis käsud `python` ja `pip` viitavad just sellele projektile

See on põhjus, miks prompti ette ilmub sageli:

```
(.venv)
```

See ei ole iluasi. See on kasulik hoiatus, et praegu töötad projekti lokaalses keskkonnas.

Ilma `venv`-ta vs koos `venv`-ga

Ilma `venv`-ta võib töövoog olla selline:

```
pip install requests
python app.py
```

Probleem on selles, et hiljem ei pruugi olla selge:

- millise `pip`-iga sa paigaldasid
- millise `python`-iga sa jooksutad
- kas paigaldasid süsteemi, kasutaja või mõne muu keskkonna alla

Koos `venv`-ga on loogika palju selgem:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install requests
python app.py
```

Siis kehtib rusikareegel:

- kõik selle projekti Pythoni paketid lähevad `.venv` sisse
- kui projekt ei tööta, otsid viga sellest keskkonnast
- kui projekt enam ei vaja seda keskkonda, võid `.venv` isegi kustutada ja uuesti luua

Miks see on algajale kasulik

`venv` on kasulik mitte ainult suures meeskonnas, vaid just algajale, sest:

- ta teeb "kus mu paketid on?" küsimuse palju lihtsamaks
- ta vähendab hirmu, et rikud süsteemi Pythoni ära
- ta õpetab kohe projekti tasemel mõtlema

Hea algaja mõtteviis on:

- üks projekt, üks `.venv`
- projekti terminal, IDE ja `pip` peavad viitama samale keskkonnale

Kas neid `venv`-sid peab olema palju

Tavaliselt mitte. Hea rusikareegel on:

- üks projekt, üks `.venv`
- kui sul on kaks eri projekti, siis neil võiks olla eri `venv`-d
- sama projekti iga väikese katse jaoks ei pea uut `venv`-i looma

See tähendab praktiliselt:

- kui töötad ühe repo sees, piisab enamasti ühest `.venv`-st
- kui teed täiesti teist projekti teise sõltuvuste komplektiga, tee uus `.venv`
- kui kaks projekti vajavad eri versioone samast paketist, peavad neil olema eri `venv`-d

Halb harjumus on:

- üks suur globaalne `pip install ...` kõigi projektide jaoks

Hea harjumus on:

- iga Pythoni projekt hoiab oma sõltuvused iseenda juures

Millal `venv` ei ole nii oluline

Kõigi ühe-realiste katsetuste jaoks ei pea alati `venv`-i looma.

Näiteks:

- `python3 -c 'print("tere")'`
- väga lühike ühekordne skript
- puhas õppimine ilma lisapakettideta

Aga niipea kui:

- projektis on välised sõltuvused
- tahad kasutada IDE-d
- tahad projekti hiljem uuesti käivitada
- jagad projekti teistega

siis on `venv` juba väga mõistlik harjumus.

Kiirspikker

- `python3 -m venv .venv` loob keskkonna
- `source .venv/bin/activate` aktiveerib selle
- `python -m pip install ...` paigaldab paketi
- `python -m pip list` näitab selle keskkonna pakette
- `deactivate` väljub keskkonnast

Kõige tavalisemad käsud

- `python3 -m venv .venv` loo projektile keskkond
- `source .venv/bin/activate` aktiveeri see shellis
- `python -m pip install requests` paigalda pakett sellesse keskkonda
- `python -m pip list` vaata, mis on paigaldatud
- `deactivate` välju keskkonnast

Käivita need käsud

```
mkdir -p ~/tmp/python-naide
cd ~/tmp/python-naide
python3 -m venv .venv
source .venv/bin/activate
python -m pip install requests
python -m pip list
deactivate
```

Mida siin ekraanil näha võiks

Sageli muutub prompt näiteks selliseks:

```
(.venv) vilo@macbook python-naide %
```

See tähendab, et:

- oled endiselt samas kataloogis
- aga `python` ja `pip` tulevad nüüd `.venv` keskkonnast

Kontrolli seda soovi korral nii:

```
command -v python
command -v pip
```

Tõenäoline tulemus on, et teed osutavad nüüd kausta `.venv/bin/`.

Hea praktiline töövoog

Kui alustad uut Pythoni projekti, siis üsna hea vaikimisi rütm on:

```
mkdir uus-projekt
cd uus-projekt
```

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip
python -m pip install requests
```

Seejärel:

- vali IDE-s interpreteerijaks `.venv/bin/python`
- hoia projektifailid samas kaustas
- ära paigalda projekti sõltuvusi niisama süsteemi

Üks aus tähelepanek

`venv` ei lahenda kõiki sõltuvuste probleeme automaatselt.

Ta ei tee näiteks sinu eest:

- versioonide lukustamist
- `requirements.txt` või `pyproject.toml` haldust
- pakettide konfliktide mõistmist

Aga ta teeb ühe väga suure asja ära: ta piirab segaduse ühe projekti sisse.

`venv` vs Docker

Need kaks ei ole päris samad tööriistad.

`venv` isoleerib:

- Pythoni paketid

Docker isoleerib:

- terve käivituskeskkonna
- operatsioonisüsteemi kasutajaruumi
- süsteemipaketid ja sõltuvused
- vajadusel ka teenused ja võrgukeskkonna

Hea lühike reegel on:

- kui probleem on ainult Pythoni pakettides, vali `venv`
- kui vajad tervet ühtlast keskkonda, vali Docker

Millal piisab `venv`-st

`venv` on sageli täiesti piisav siis, kui:

- teed puhast Pythoni projekti
- vajad ainult `pip`-i kaudu paigaldatavaid teeke
- töötad üksi või väikeses tiimis
- tahad kiiresti arendada ilma konteinerikihti juurde toomata

Näited:

- väike CLI-tööriist
- andmetöötluse skript
- lihtne veebirakendus arenduse alguses
- testide jooksutamine lokaalses Pythoni projektis

Millal minna Dockerisse

Docker muutub mõistlikuks siis, kui:

- tahad, et kõigil oleks sama keskkond
- vajad lisaks Pythonile süsteemipakette nagu `libpq`, `ffmpeg`, `imagemagick`
- projekt käib koos andmebaasi, Redis-e, Nginx-i või muu teenusega
- tahad sama keskkonda arenduses, CI-s ja serveris

Näited:

- veebirakendus koos Postgres-iga
- teenus, mis sõltub kindlast Linuxi paketist
- meeskonnaprojekt, kus "minu masinas töötab" on sage probleem

Kas kasutada `venv`-i Dockeri sees

Enamasti:

- lokaalses arenduses võib sul olla `venv`
- Dockeri konteineri sees ei ole eraldi `venv` sageli vajalik

Põhjus on lihtne:

- konteiner ise juba isoleerib keskkonna
- ühe rakenduse konteineris ei ole tavaliselt vaja Pythoni pakette veel teise kihi sisse peita

Tavaline praktiline muster on:

- kohalikus masinas arendad `venv`-ga
- tootmises või ühtses arenduskeskkonnas jooksutad Dockerit

`venv` konteineri sees võib siiski mõnikord olla mõistlik, kui:

- konteineris on mitu eri Pythoni töövoogu
- tahad väga teadlikult hoida eraldi tööriistu ja rakendust
- sul on eriline buildi- või testivajadus

Aga alguses tasub mõelda nii:

- väljaspool konteinerit: `venv`
- konteineri sees: enamasti piisab konteinerist endast

Minitest

1. Loo uus virtuaalkeskkond.
2. Aktiveeri see.
3. Paigalda üks lihtne pakett ja kuva installitud paketid.
4. Kontrolli, kuhu käsud `python` ja `pip` aktiveerimise järel osutavad.
5. Selgita ühe lausega, miks `venv` on kasulik isegi siis, kui sul on ainult üks väike projekt.

Dockeri alused

Selles peatükis teeme praktilise sissejuhatuse konteineritesse ja Dockeri põhimõistetesse.

Loogika

Docker võimaldab käivitada tarkvara eraldatud keskkonnas nii, et sama image annaks võimalikult sarnase tulemuse eri masinates. See on seotud arenduskeskkonna, paketihalduse ja serveritööga.

Oluline on kohe alguses eristada:

- `venv` hoiab lahus Pythoni paketid
- Docker hoiab lahus terve käivituskeskkonna

See on põhjus, miks Docker ei ole lihtsalt "suurem venv".

Põhimõtted

- image on valmis ehituskihiline pakend
- container on image'ist käivitatud protsess
- registry on koht, kust image'eid alla laadida
- port ühendab konteineri teenuse sinu masina pordiga
- volume või bind mount annab püsiva või jagatud failisüsteemi osa

Image, container, port ja volume

Need mõisted tasub võimalikult vara selgeks teha:

- image retseptist ehitatud valmis pakend
- container selle image'i põhjal käivitatud konkreetne protsess
- port viis, kuidas konteineri teenus nähtavaks teha hostmasinas
- volume või bind mount viis, kuidas andmeid püsivalt või jagatult kasutada

Lihtne võrdlus:

- image on nagu retsept või valmis pakend
- container on sellest tehtud päris jooksutatud eksemplar

Sa võid ühest image'ist käivitada mitu konteinerit.

Miks Docker üldse kasulik on

Docker lahendab teistsugust probleemi kui `venv`.

Kõige tavalisemad põhjused Dockeri kasutamiseks on:

- sama rakendus peab töötama eri masinates ühtemoodi
- projekt vajab lisaks Pythonile või Node'ile ka süsteemipakette
- rakendus käib koos teiste teenustega nagu Postgres, Redis või Nginx
- arendus, test ja tootmine peavad olema võimalikult sarnased

Lühidalt:

- `venv` aitab küsimusega "millised Pythoni paketid siin on"
- Docker aitab küsimusega "milline terve keskkond siin üldse jookseb"

Kiirspikker

- `docker pull` tõmbab image'i
- `docker run` käivitab konteineri
- `docker ps` näitab jooksvaid konteinereid
- `docker images` näitab image'eid
- `docker exec -it` siseneb konteinerisse
- `docker logs` näitab konteineri väljundit

Kõige tavalisemad lipud

- `docker run --rm`
- `docker run -it`
- `docker exec -it`
- `docker ps`

Praktiliselt tähendavad need enamasti:

- `docker run --rm` kustuta konteiner pärast lõpetamist
- `docker run -it` interaktiivne terminal
- `docker exec -it` sisene töötavasse konteinerisse
- `docker ps` vaata jooksvaid konteinereid

Kasulikud lisad:

- `docker logs konteiner` vaata konteineri väljundit
- `docker run -p 8000:8000` ava port hostmasinasse
- `docker run -v "$PWD":/app` jaga kohalik kaust konteineriga

`venv` vs Docker

Hea otsustuspuu on:

- ainult Pythoni sõltuvused lähevad segamini: `venv`
- vajad kindlat Linuxi keskkonda või süsteemipakette: Docker

- vajad mitut teenust koos: Docker
- tahad ainult kiiret lokaalset Pythoni arendust: `venv`

Näide, kus piisab `venv`-st:

- väike Pythoni skript
- üks CLI-tööriist
- andmetöötlusnotebook või väike teek

Näide, kus Docker on loomulikum:

- veebirakendus koos andmebaasiga
- projekt, mis vajab `postgresql-client`, `ffmpeg` või muud süsteemipaketti
- tiimitöö, kus keskkonnad kipuvad masinate vahel erinema

Kas konteineris on vaja veel `venv`-i

Tavaliselt mitte.

Hea algreegel on:

- väljaspool Dockerit: kasuta projektis `venv`-i
- Dockeri sees: ära lisa `venv`-i ainult harjumusest

Põhjus:

- konteiner juba isoleerib keskkonna
- üks rakendus ühes konteineris on tavaliselt piisavalt eraldatud

Seega on väga tavaline ja täiesti normaalne, et Dockerfile teeb lihtsalt:

```
FROM python:3.13-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Siin ei ole eraldi `venv`-i vaja, sest konteiner ise ongi eralduskiht.

Praktiline soovitus sinu jaoks

Kui hakkad Dockerit aktiivselt kasutama, siis üsna hea mõtteviis on:

- kohaliku puhta Pythoni arenduse jaoks kasuta `venv`-i
- kui projekt peab jooksma ühtemoodi eri masinates või vajab teenuseid, tee Docker
- ära pane lisaks konteinerisse `venv`-i, kui sul ei ole selleks väga selget põhjust

See hoiab süsteemi lihtsana:

- arenduse ajal tead, kas oled `venv`-s või konteineris

- tootmise ja tiimitöö jaoks on keskkond ühtsem
- veaotsingus on vähem kihte, mida segamini ajada

Käivita need käsud

```
docker pull alpine
docker run --rm alpine echo tere
```

```
docker run -it --rm ubuntu bash
docker ps
docker images
```

Kõige väiksem arusaadav `Dockerfile`

Kui tahad Dockeri mõttest päriselt aru saada, siis kõige kasulikum on teha üks väga väike rakendus ja panna see image'isse.

Näiteks fail `app.py`:

```python
print("Tere konteinerist")
```

Ja selle kõrvale `Dockerfile`:

```
FROM python:3.13-slim
WORKDIR /app
COPY app.py .
CMD ["python", "app.py"]
```

Siin toimub:

- `FROM python:3.13-slim` valib Pythoni baaskeskkonna
- `WORKDIR /app` määrab töökausta konteineri sees
- `COPY app.py .` kopeerib faili image'isse
- `CMD ["python", "app.py"]` ütleb, mis käivitub konteineri startimisel

See on hea esimene näide, sest siin ei ole veel kõrvalist keerukust.

`docker build` ja `docker run`

Kui `Dockerfile` ja `app.py` on samas kaustas, siis:

```
docker build -t tere-rakendus .
docker run --rm tere-rakendus
```

Tulemus peaks olema umbes:

```
Tere konteinerist
```

Oluline loogika:

- `docker build` teeb image'i
- `docker run` käivitab selle image'i põhjal konteineri

- `--rm` kustutab konteineri pärast lõpetamist

Kui tahad konteinerisse sisse vaadata, on abiks:

```
docker run -it --rm python:3.13-slim bash
```

See on hea viis vaadata, milline keskkond konteineri sees päriselt on.

Portide avamine

Kui konteineri sees jookseb veebiserver, siis ei piisa ainult sellest, et protsess töötab. Sageli on vaja port hostmasinasse edasi anda.

Näide:

```
docker run --rm -p 8000:8000 python:3.13-slim python -m http.server 8000
```

Siin:

- konteineri sees kuulab server porti `8000`
- `-p 8000:8000` seob selle sinu masina pordiga `8000`

Pärast seda saad tavaliselt minna brauseris aadressile:

```
http://localhost:8000
```

Bind mount ja püsivad andmed

Kui tahad, et konteiner näeks sinu kohalikku kausta, on väga tavaline kasutada bind mount'i.

Näide:

```
docker run --rm -it -v "$PWD":/app -w /app python:3.13-slim bash
```

Siin:

- `-v "$PWD":/app` jagab praeguse kausta konteinerisse
- `-w /app` teeb selle konteineri töökaustaks

See on arenduses väga kasulik, sest saad muuta faile hostmasinas ja näha neid kohe konteineris.

Kui räägitakse volume'itest, siis mõeldakse sageli kahte eri asja:

- bind mount: kindel hosti tee nagu `"$PWD":/app`
- named volume: Dockeri hallatav püsiv andmeala

Algajale on bind mount tavaliselt kõige lihtsam esimene samm.

Konteineris arendamine: mis jääb hosti ja mis jookseb konteineris

See on koht, kus Docker muutub päriselt arendustööriistaks, mitte ainult demo- või deploy-vahendiks.

Kõige olulisem loogika on:

- lähtekood ja Git ajalugu elavad tavaliselt hostmasina kaustas
- konteiner annab sellele koodile ühtse jooksutuskeskkonna
- muudatusi teed tavaliselt oma redaktoris või IDE-s
- rakendust käivitad, testid ja silud konteineri sees

Hea algreegel on:

- ära kirjuta olulist koodi "anonüümselt" ainult konteineri sees
- hoia projekt failidena oma kaustas ja jaga see kaust bind mount'iga konteinerile

Muidu võib juhtuda, et:

- konteiner kustub
- sisse kirjutatud failid kaovad
- sa ei saa hästi aru, mis on hostis ja mis oli ainult konteineri sees

Kõige lihtsam arendusvoog bind mount'iga

Kui sul on näiteks väike Pythoni projekt, siis väga praktiline esimene arendusvoog on:

```
docker run --rm -it -v "$PWD":/app -w /app python:3.13-slim bash
```

Selle loogika:

- `"$PWD":/app` jagab sinu praeguse projekti kausta konteinerisse
- `-w /app` teeb selle konteineri töökaustaks
- sa saad hostmasinas faile muuta ja konteiner näeb neid kohe

Näiteks:

```
python -m pip install -r requirements.txt
python app.py
pytest
```

See on hea siis, kui tahad:

- proovida sõltuvusi puhtas keskkonnas
- mitte solkida hostmasina Pythoni
- kontrollida, et projekt töötab ka mujal kui sinu enda masinas

Miks see erineb "docker build && docker run" töövoost

Neid kaht töövoogu tasub eristada:

- `docker build` + `docker run` hea siis, kui tahad kontrollida valmis image'it
- bind mount'iga interaktiivne konteiner hea siis, kui tahad aktiivselt arendada

Teisisõnu:

- valmis rakenduse proovimiseks ehitad image'i
- igapäevaseks arendamiseks tahad sageli, et kood oleks mount'iga kohe nähtav

Praktiline arendus `docker compose` abil

Kui projektis on rohkem kui üks teenus, muutub arenduses mugavaks `docker compose`.

Väga tüüpiline arenduskuju on:

`compose.yaml`

```yaml
services:
  app:
    image: python:3.13-slim
    working_dir: /app
    volumes:
      - .:/app
    command: bash -lc "python -m pip install -r requirements.txt && python app.py"
    ports:
      - "8000:8000"

  db:
    image: postgres:16
    environment:
      POSTGRES_DB: opik
      POSTGRES_USER: opik
      POSTGRES_PASSWORD: opikpass
```

Selle mõte:

- kood elab hostmasinas
- `app` teenus kasutab seda otse läbi mount'i
- `db` annab teise teenusena andmebaasi
- kogu komplekti saad käivitada ühe käsuga

Tüüpilised arenduskäsud:

```
docker compose up
docker compose logs -f app
docker compose exec app bash
docker compose down
```

Need on arenduses väga tähtsad:

- `up` käivitab tööruumi
- `logs -f` näitab jooksvaid logisid
- `exec` lubab töötavasse konteinerisse sisse minna
- `down` peatab komplekti

Mida konteineris teha ja mida mitte

Hea praktiline jaotus on:

- hostmasinas: muuda faile, tee Git commit'e, halda dokumentatsiooni
- konteineris: installi sõltuvusi, käivita rakendus, testi, silu keskkonda

See ei ole absoluutne reegel, aga algajale väga tervislik tööjaotus.

Eriline hoiatus:

- kui teed konteineri sees `git clone`, lood uusi faile ja ei kasuta mount'i
- siis töötad sisuliselt konteineri sisemises failisüsteemis
- see võib olla ajutine ja segadust tekitav

Kuidas konteineris arendust mõelda

Hea vaimne mudel on:

- hostmasin on sinu töölaud
- Git repo on päris "tõde"
- konteiner on vahetatav tööpink

Kui tööpink maha võtta ja uuesti püsti panna, peaksid olulised asjad alles olema:

- lähtekood
- konfiguratsioon
- andmed, mida tahtsid säilitada

See on põhjus, miks:

- kood pannakse tavaliselt mount'iga hostist sisse
- andmebaasi andmed pannakse volume'isse
- buildi- ja arenduskäsud kirjutatakse `Dockerfile`-i või `compose.yaml`-i

`docker logs` ja `docker exec`

Konteinerit ei pea alati kohe interaktiivselt käivitama, et aru saada, mis toimub.

Sageli piisab neist kahest käsust:

```
docker logs konteineri-nimi
docker exec -it konteineri-nimi bash
```

Loogika:

- `docker logs` näitab, mida protsess on kirjutanud `stdout`-i ja `stderr`-i
- `docker exec -it` lubab sul töötavasse konteinerisse sisse minna

Need on veaotsingus ühed kõige kasulikumad käsud.

Kui rakendus vajab pakette

Siis lisandub tavaliselt `requirements.txt`.

Näide:

`requirements.txt`

```
requests==2.32.3
```

`Dockerfile`

```
FROM python:3.13-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
CMD ["python", "app.py"]
```

Siin on oluline:

- Pythoni paketid lähevad image'i sisse
- konteineri sees ei ole vaja eraldi `venv`-i
- image ise on selle rakenduse eraldatud keskkond

Millal tuleb mängu `docker compose`

`docker run` sobib hästi ühe konteineri jaoks.

Kui aga projektis on mitu seotud teenust, näiteks:

- rakendus
- andmebaas
- vahemälu

siis muutub mugavamaks `docker compose`.

Hea mõtteviis on:

- `docker run` on üksiku konteineri käivitamine
- `docker compose` on mitme seotud teenuse kirjeldamine ühes failis

Lihtne `docker compose` näide: Python + Postgres

See on juba päris eluline näide, sest väga paljud rakendused ei jookse üksi, vaid koos andmebaasiga.

`compose.yaml`

```yaml
services:
  app:
    build: .
    command: python app.py
    depends_on:
      - db
    environment:
      POSTGRES_HOST: db
      POSTGRES_DB: opik
      POSTGRES_USER: opik
      POSTGRES_PASSWORD: opikpass

  db:
    image: postgres:16
    environment:
      POSTGRES_DB: opik
      POSTGRES_USER: opik
      POSTGRES_PASSWORD: opikpass
```

Siin:

- `app` on sinu Pythoni rakendus
- `db` on Postgres
- nimi `db` toimib teenusevõrgus hostinimena
- `depends_on` ütleb, et rakendus sõltub andmebaasikonteinerist

Käivitus:

```
docker compose up --build
```

Peatamine:

```
docker compose down
```

Oluline täpsustus:

- `depends_on` aitab käivitusjärjekorraga
- see ei tähenda automaatselt, et Postgres on juba päriselt valmis ühendusi vastu võtma
- päris projektis on sageli vaja ka "oota kuni DB on valmis" loogikat

Millal valida kumb

Kasuta:

- `docker run`, kui tahad kiiresti testida üht image'it või üht käsku
- `docker compose`, kui projektis on mitu teenust

Hea rusikareegel on:

- üks konteiner: `docker run`

- rakendus + andmebaas + muud teenused: `docker compose`

Seos arenduskonteineritega

Kui kasutad VS Code'i või mõnd muud IDE-d, võib sama loogika olla pakitud arenduskonteineri kujule.

Siis:

- Docker käivitab konteineri
- IDE ühendub sinna sisse
- sinu projektikaust mount'itakse tööruumina konteinerisse

See tähendab, et arenduskonteiner ei ole "teine asi" kui Docker. See on pigem Dockeri peal ehitatud arendusmugavuskiht.

Üks lihtne võrdlus

`venv`-ga töövoog:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install requests
python app.py
```

Dockeri töövoog:

```
docker build -t minu-rakendus .
docker run --rm minu-rakendus
```

Esimesel juhul isoleerid Pythoni paketid.

Teisel juhul isoleerid kogu rakenduse käivituskeskkonna.

Praktiline soovitus alustamiseks

Kui hakkad Dockerit õppima või arendama, mine selles järjekorras:

1. tee kõige väiksem `Dockerfile`
2. õpi `docker build`
3. õpi `docker run`
4. alles siis mine `docker compose` juurde

Nii püsib selge:

- mis on image
- mis on container
- millal vajad mitme teenuse haldust

Minitest

1. Tõmba alla mõni väike image.
2. Käivita konteiner, mis väljastab ühe rea.
3. Selgita oma sõnadega image'i ja containeri vahet.
4. Selgita ühe lausega, mis vahe on `venv`-il ja Dockeril.
5. Kirjelda ühe lausega, millal kasutaksid `docker compose`-i tavalise `docker run` asemel.

IDE-d ja arenduskeskkonnad

Selles peatükis vaatame soovitatavaid tööriistu, terminali integreerimist ja arenduskeskkondade ülesseadmist.

Loogika

Arenduskeskkond on tervik, mitte ainult tekstiredaktor. Terminal, Git, keelekeskkond ja vajadusel Docker või kaugühendus peavad töötama koos.

Kiirspikker

- `python3 --version` kontrolli Pythoni olemasolu
- `python3 -m pip --version` kontrolli `pip`-i olemasolu
- `node --version` kontrolli Node.js-i olemasolu
- `npm --version` kontrolli `npm`-i olemasolu
- `git --version` kontrolli Git-i olemasolu
- `docker --version` kontrolli Dockeri olemasolu

Praktiliselt tähendab see:

- terminal peab olema käepärast
- Git peab olema kasutatav
- projekti Python või Node keskkond peab olema õigesti valitud
- vajadusel peab IDE oskama töötada Dockeriga või kaugmasinaga

Soovitatav lähtekoht

Algajale on tavaliselt hea kombinatsioon:

- terminal
- üks kerge tekstiredaktor või IDE
- Git
- Python või muu põhitööriist

Levinud valikud

- VS Code: väga levinud, laiendatav, hea Remote SSH tugi
- PyCharm: tugev Pythoni tugi

- Vim või Neovim: kiire klaviatuuripõhine töö
- JetBrainsi tööriistad üldisemalt: tugev projektitugi

Mida IDE-s jälgida

- sisseehitatud terminal
- Git-i integratsioon
- projekti virtuaalkeskkond
- Dockeri tugi
- kaugühenduste tugi

VS Code ja arenduskonteinerid

Kui projekt kasutab Dockerit, siis väga praktiline järgmine samm on arenduskonteiner.

Selle põhimõte on:

- projektifailid jäävad sinu masinasse
- VS Code avab sama projekti Dockeri konteineri sees
- terminal, interpreter ja tööriistad jooksevad konteineris

See aitab eriti siis, kui:

- projekt vajab kindlat Linuxi keskkonda
- mitmel arendajal peab olema sama töölaud
- hostmasinat ei taha liigsete tööriistadega täita

Lühidalt:

- `venv` aitab Pythoni pakettidega
- Docker aitab kogu keskkonnaga
- arenduskonteiner ühendab selle IDE kasutusmugavusega

Minimaalne `devcontainer.json`

Väga väike näide:

`.devcontainer/devcontainer.json`

```
{
  "name": "Python Dev Container",
  "image": "python:3.13-slim",
  "workspaceFolder": "/workspace",
  "mounts": [
    "source=${localWorkspaceFolder},target=/workspace,type=bind"
  ],
  "postCreateCommand": "python -m pip install -r requirements.txt",
  "customizations": {
    "vscode": {
```

```
      "extensions": [
        "ms-python.python",
        "ms-python.vscode-pylance"
      ]
    }
  }
}
```

Mida see tähendab:

- kasutatakse olemasolevat Pythoni image'it
- kohalik projekt mount'itakse `/workspace` alla
- pärast loomist paigaldatakse sõltuvused
- VS Code saab automaatselt vajalikud laiendused

Tüüpiline töövoog arenduskonteineriga

1. paigalda VS Code'is laiendus `Dev Containers`
2. ava projektikaust
3. vali käsk `Dev Containers: Reopen in Container`
4. oota, kuni konteiner ehitatakse või käivitatakse
5. tööta edasi nagu tavaliselt, aga IDE terminal on nüüd konteineri sees

Hea mõte on jälgida:

- kus jookseb Python või Node
- kus tehakse `pip install` või `npm install`
- kas avatud terminal on hostis või konteineris

Millal eelistada arenduskonteinerit

Arenduskonteiner on eriti hea siis, kui:

- projektis on Docker niikuinii olemas
- tahad, et terve tiim kasutaks sama arenduskeskkonda
- kasutad palju süsteemipakette või teenuseid

Kui projekt on väga väike ja puhas, võib olla lihtsam:

- kasutada lihtsalt `venv-i`
- või jooksutada üksikuid käske bind mount'iga konteineris

Seos tavalise Dockeri arendusega

Arenduskonteiner ei asenda Dockeri põhimõtteid, vaid peidab osa käske sinu eest ära.

Sama loogika jääb alles:

- kood on hostmasinas

- konteiner annab keskkonna
- mount ühendab need kaks

Kui saad käsureal aru:

- `docker run -v "$PWD":/app ...`
- `docker compose up`
- `docker compose exec app bash`

siis on sul palju lihtsam mõista ka IDE arenduskonteinereid.

Käivita need käsud

Kontrolli, kus Python süsteemist leitakse:

```
command -v python3
python3 --version
python3 -m pip --version
```

Kontrolli, kas Git ja Docker on saadaval:

```
git --version
docker --version
```

Kontrolli Node.js ja npm olemasolu:

```
node --version
npm --version
```

Pythoni töövoog IDE-s

Pythoni projektis tasub tavaliselt siduda IDE konkreetse virtuaalkeskkonnaga:

1. loo projektis `.venv`
2. vali IDE-s interpreteerijaks selle keskkonna Python
3. paigalda sõltuvused sinna, mitte suvaliselt süsteemi

Node.js ja npm töövoog IDE-s

JavaScripti või TypeScripti projektis tasub jälgida:

- kas projektis on `package.json`
- kas sõltuvused on paigaldatud käsuga `npm install`
- millised skriptid on kirjas plokis `scripts`

Tüüpilised käsud:

```
npm install
npm run dev
npm test
```

Minitest

1. Pane kirja, millist redaktorit või IDE-d sa kasutad.
2. Kontrolli, kas IDE terminal kasutab sama shelli mis tavaline terminal.
3. Uuri, kuidas selles IDE-s Git commit'i teha.
4. Kontrolli, kuidas valida IDE-s Pythoni interpreter või Node.js projekt.
5. Kui kasutad VS Code'i, uuri, kus menüüs on `Reopen in Container`.

Andmeteaduse eelteadmised käsurea vaates

Selles peatükis seome kokku, milliseid teadmisi andmeteaduse või andmeanalüüsi suund tavaliselt eeldab ja kuidas käsurida aitab neid praktiliselt toetada.

Loogika

Kui räägitakse andmeteaduse eelteadmistest, siis tavaliselt mõeldakse vähemalt neid plokke:

- programmeerimine, eriti Python
- andmebaasid, SQL ja relatsiooniline mõtteviis
- failivormingud nagu `CSV`, `JSON` ja `XML`
- statistika, tõenäosusteooria ja matemaatiline mõtlemine

Käsurida ei asenda neid kõiki, aga ta aitab neid kokku siduda.

Just siin on käsurea suur väärtus:

- näed kiiresti, mis failid sul üldse on
- saad kontrollida andmete kuju enne, kui lähed suuremasse tööriista
- saad teha väikseid filtreid, loendusi ja ümberkujundusi
- õpid töövoogu, mis on hiljem kasulik ka Pythonis, SQL-is ja Dockeris

See õpik katab tugevamalt:

- käsurea loogika
- failide ja voogude töötluse
- Pythoni keskkonnad
- SQLite'i ja SQL-i alguse
- arendustöövoo, Git-i ja Dockeri

See õpik ei püüa eraldi õpetada põhjalikult:

- statistikat
- tõenäosusteooriat
- lineaaralgebrat
- `R`-i

Kiirspikker

- Pythoni venv ja eraldatud keskkonnad aitab projektid korras hoida

- CSV, JSON ja XML käsureal aitab andmete kuju kiiresti näha
- Andmebaasi algus: sqlite ja Python annab esimese SQL-i ja relatsioonilise mudeli tunnetuse
- Teksti teisendamine ja Vood ja tabelid annavad väikeste andmetööde baasi

Hea rusikareegel on:

- enne suurt tööriista vaata andmeid väikese käsuga
- enne keerulist analüüsi kontrolli, et saad aru, mis kujul andmed üldse on

Kõige tavalisemad vajadused

Programmeerimine

Andmetöö juures tähendab see sageli:

- väikest automaatikat
- andmete lugemist failist
- tulemuste salvestamist
- skriptide korduvat käivitamist

See tuleb kõige rohkem välja Pythoni, shelliskriptide ja töövoogude peatükkides.

Andmevormingud

Väga tihti ei ole probleem kohe mitte statistikas, vaid selles, et:

- fail on vales vormingus
- veerud ei ole seal, kus arvasid
- kirjed on pesastatud
- andmed on teksti sees, mitte tabelina

Seetõttu on `CSV`, `JSON` ja `XML` mõistmine väga praktiline baasoskus.

SQL ja relatsiooniline mõtteviis

SQL ei tähenda ainult "käsk andmebaasile", vaid ka teatud andmemudelit.

Kasulikud põhiküsimused on:

- mis on tabel
- mis on rida ja veerg
- mis on primaarvõti
- kuidas kaks tabelit omavahel seotakse

Kui see loogika on olemas, on ka keerulisemad päringud palju vähem müstilised.

Statistika ja matemaatiline mõtlemine

Seda osa ei saa käsurea või SQL-iga asendada.

Oluline aus mõte on:

- käsurida aitab andmeid ette valmistada
- Python või `R` aitab neid töödelda
- statistiline mõtlemine aitab tulemusi mõista

Kõik kolm on eri asjad.

Näited

Väga tüüpiline väike andmetöö rada võib olla selline:

1. vaata faili esimesi ridu
2. kontrolli, mis väljad seal on
3. tee lihtne filtreerimine või loendus
4. pane andmed SQLite tabelisse
5. tee esimene SQL päring
6. loe tulemus Pythoniga sisse

See tähendab, et päris tööriistajoon võib olla:

```
fail -> head/less/grep -> column/cut/tr -> sqlite3 -> python3
```

Ja just selle pärast on käsurida andmeteaduse stardis kasulik:

- ta aitab väikeste sammudega kiiresti pilti ette saada
- ta ei sunni kohe suurt keskkonda avama
- ta teeb veaallikad nähtavamaks

Minitest

1. Nimeta neli plokki, mida andmeteaduse eelteadmistena kõige sagedamini mainitakse.
2. Selgita ühe lausega, miks `CSV`, `JSON` ja `XML` ei ole sama asi.
3. Selgita ühe lausega, miks SQL ja statistika ei ole asendatavad oskused.
4. Pane kirja üks väike töövoog, kus kasutaksid nii käsurida, SQLite'i kui Pythonit.

CSV, JSON ja XML käsureal

Selles peatükis vaatame kolme väga levinud andmevormingut ja seda, kuidas neid käsureal kiiresti uurida.

Loogika

Need kolm vormingut esindavad sageli kolme eri mõtteviisi:

- `CSV` on tavaliselt lihtne tabel
- `JSON` on sageli objektide ja massiivide puu
- `XML` on märgenditega puustruktuur

Kui saad aru, millist kuju andmed võtavad, on palju lihtsam otsustada:

- kas kasutada `cut`, `column` ja `sort`
- kas võtta appi `jq`
- kas minna SQLite'i või Pythoni peale

Oluline praktiline mõte on:

- käsurida sobib hästi kiireks vaatamiseks
- keerulisema loogika jaoks on sageli parem minna Pythonisse või SQL-i

Kiirspikker

- `head fail.csv` vaata CSV esimesi ridu
- `column -s, -t < fail.csv` kuva lihtne CSV tabelina
- `cut -d, -f1 fail.csv` võta üks CSV veerg lihtsal juhul
- `python3 -m json.tool fail.json` kuva JSON loetavalt
- `jq '.voti' fail.json` vali JSON-ist välju
- `grep '<tag>' fail.xml` leia XML-ist kiirelt märgendeid
- `xmllint --format fail.xml` vorminda XML loetavaks

Kõige tavalisemad tööriistad

`CSV`

Lihtsa CSV puhul on väga kasulikud:

- `head`
- `column`
- `cut`
- `sort`
- `wc`

Tähtis hoiatus:

- päris CSV võib sisaldada jutumärke, komasid välja sees ja reavahetusi
- siis ei pruugi `cut -d,` olla enam piisavalt täpne

Seega:

- kiireks vaatamiseks on shell väga hea
- päris keerulise CSV jaoks on parem kasutada Pythonit või spetsiaalset tööriista

`JSON`

JSON on väga levinud:

- API vastustes
- seadistusfailides
- logides

- veebiteenuste väljundis

Kõige praktilisemad esimesed tööriistad on:

- `python3 -m json.tool`
- `jq`

Kui `jq` puudub, saad JSON-i vähemalt ilusti vaadata `python3 -m json.tool` käsuga.

XML

XML-i kohtab tänapäeval vähem kui JSON-i, aga ta on endiselt oluline:

- vanemates süsteemides
- teaduslikes andmevahetustes
- mõnes konfiguratsioonis
- dokumentide ja metaandmete juures

Hea algreegel:

- `grep` sobib kiireks piilumiseks
- `xmllint` sobib ilusaks vorminduseks
- keerulisema XML loogika jaoks on parem kasutada spetsiaalsemaid tööriistu või Pythonit

Näited

Näide: lihtne CSV

```
cat > tudengid.csv <<'EOF'
name,city,score
Mari,Tartu,91
Jaan,Tallinn,84
Liis,Narva,88
EOF
```

Vaatame faili:

```
head -n 4 tudengid.csv
column -s, -t < tudengid.csv
cut -d, -f1 tudengid.csv
```

Siin:

- `head` näitab algust
- `column` muudab lihtsa CSV visuaalselt tabeliks
- `cut -d, -f1` võtab esimese veeru

Näide: lihtne `JSON`

```
cat > tudengid.json <<'EOF'
{
  "students": [
    {"name": "Mari", "city": "Tartu", "score": 91},
    {"name": "Jaan", "city": "Tallinn", "score": 84},
    {"name": "Liis", "city": "Narva", "score": 88}
  ]
}
EOF
```

Vaatame JSON-i:

```
python3 -m json.tool tudengid.json
jq '.students[].name' tudengid.json
```

Siin:

- `python3 -m json.tool` teeb JSON-i loetavaks
- `jq '.students[].name'` käib massiivi läbi ja võtab nimed

Näide: lihtne `XML`

```
cat > tudengid.xml <<'EOF'
<students>
  <student city="Tartu">
    <name>Mari</name>
    <score>91</score>
  </student>
  <student city="Tallinn">
    <name>Jaan</name>
    <score>84</score>
  </student>
</students>
EOF
```

Vaatame XML-i:

```
grep '<name>' tudengid.xml
xmllint --format tudengid.xml
```

Siin:

- `grep` annab kiire märgendi-põhise piilumise
- `xmllint --format` teeb struktuuri palju loetavamaks

Kui `xmllint` puudub, siis see ei ole üllatav. Ta ei ole igas süsteemis vaikimisi olemas.

Mida meelde jätta

- `CSV` on kõige lähemal tabelile
- `JSON` sobib hästi pesastatud andmetele
- `XML` on samuti puustruktuur, aga märgenditega
- käsurida sobib väga hästi esimeseks vaatamiseks
- keerulisem töö läheb sageli edasi SQLite'i, Pythonisse või mõnda teise tööriista

Minitest

1. Loo üks väike `CSV` fail kolme reaga ja kuva see `column` abil tabelina.
2. Loo üks väike `JSON` fail ja kuva sellest ainult nimed.
3. Loo üks väike `XML` fail ja otsi sellest üks märgend `grep` abil.
4. Selgita ühe lausega, miks `cut -d,` ei pruugi alati päris CSV jaoks piisata.

Andmebaasi algus: SQLite ja Python

Selles peatükis teeme esimese praktilise silla tabelite, SQL-i ja Pythoni vahele.

Loogika

SQLite on hea esimene andmebaas, sest ta ei vaja eraldi serverit ja elab lihtsalt failina.

See teeb ta heaks sillaks nende teemade vahel:

- failid ja failisüsteem
- SQL päringud
- relatsiooniline andmemudel
- Pythoni programm, mis andmebaasi kasutab

Oluline mõte on:

- `CSV` fail on lihtsalt tekstifail
- andmebaas lisab sellele struktuuri, päringud ja seosed

Relatsioonilise mõtteviisi kõige tähtsamad algmõisted on:

- tabel: ühe teema andmed
- rida: üks kirje
- veerg: ühe omaduse koht
- primaarvõti: rea unikaalne tunnus
- võõrvõti: viide teise tabeli reale
- `JOIN`: kahe tabeli kokku sidumine seose järgi

SQLite on hea, sest need mõisted saab läbi proovida ilma, et peaksid kohe serverit või pilvekeskkonda haldama.

Kiirspikker

- `sqlite3 andmed.db` ava andmebaas
- `.tables` näita tabeleid
- `.schema` näita tabelite struktuuri
- `select * from tabel limit 5;` kuva paar esimest rida
- `select ... from a join b on ...;` ühenda kaks tabelit
- `select city, count(*) from students group by city;` koonda read rühmade kaupa

Väga praktilised 1-linerid on:

```
sqlite3 andmed.db '.tables'
sqlite3 andmed.db '.schema'
sqlite3 andmed.db 'select * from students limit 5;'
sqlite3 andmed.db 'select s.name, r.score from results r join students s on s.id = r.student
sqlite3 andmed.db 'select city, count(*) from students group by city order by count(*) desc;
```

Need on head just sellepärast, et ei pea alati minema interaktiivsesse `sqlite3` shelli, kui tahad lihtsalt kiiret vastust.

Kõige tavalisemad käsukujud

- `create table ...;` loo tabel
- `insert into ... values (...);` lisa read
- `select * from ...;` kuva kõik read
- `select ... from ... where ...;` filtreeri
- `select ... from a join b on ...;` ühenda tabelid
- `select ..., count(*) from ... group by ...;` loenda rühmade kaupa

Hea rusikareegel:

- üks tabel kirjeldab üht tüüpi asja
- teise tabeli viide esimesele tabelile tehakse võõrvõtmega
- `JOIN` ei ole eraldi maagia, vaid viis need read omavahel kokku viia

Näited

Näide: üks tabel

Loome väga väikese andmebaasi ühe tabeliga:

```
sqlite3 andmed.db <<'EOF'
drop table if exists students;
create table students (
  id integer primary key,
  name text not null,
  city text,
  score integer
```

```
);
insert into students (name, city, score) values
  ('Mari', 'Tartu', 91),
  ('Jaan', 'Tallinn', 84),
  ('Liis', 'Narva', 88);
select * from students;
EOF
```

Siin:

- `id integer primary key` annab igale reale unikaalse tunnuse
- `name`, `city` ja `score` on veerud
- iga `insert` lisab ridu

Kasulikud järgmised vaated:

```
sqlite3 andmed.db '.schema students'
sqlite3 andmed.db 'select name, score from students order by score desc;'
sqlite3 andmed.db 'select city, count(*) from students group by city;'
```

Näide: kaks tabelit ja `JOIN`

Nüüd teeme andmemudeli natuke realistlikumaks. Punktid ei pea olema samas tabelis nagu tudengi põhiandmed.

```
sqlite3 andmed.db <<'EOF'
drop table if exists results;
drop table if exists students;

create table students (
  id integer primary key,
  name text not null,
  city text
);

create table results (
  id integer primary key,
  student_id integer not null,
  subject text not null,
  score integer not null,
  foreign key (student_id) references students(id)
);

insert into students (id, name, city) values
  (1, 'Mari', 'Tartu'),
  (2, 'Jaan', 'Tallinn'),
  (3, 'Liis', 'Narva');
```

```
insert into results (student_id, subject, score) values
  (1, 'matemaatika', 91),
  (1, 'python', 95),
  (2, 'matemaatika', 84),
  (2, 'python', 79),
  (3, 'matemaatika', 88),
  (3, 'python', 90);
EOF
```

Nüüd:

- tabel `students` hoiab tudengite põhiandmeid
- tabel `results` hoiab tulemusi
- `results.student_id` viitab `students.id` väljale

See ongi relatsioonilise andmemudeli põhiidee: seosed tehakse võtmete kaudu, mitte suvalise tekstilise kokkusobitamisega.

Vaatame tulemusi:

```
sqlite3 andmed.db 'select * from students;'
sqlite3 andmed.db 'select * from results;'
```

Ja nüüd ühendame need:

```
sqlite3 andmed.db "
select s.name, s.city, r.subject, r.score
from results r
join students s on s.id = r.student_id
order by s.name, r.subject;
"
```

Siin:

- `r` ja `s` on lühikesed aliased tabelinimedele
- `join students s on s.id = r.student_id` ütleb, kuidas read kokku viiakse
- väljundis näed ühe tabeli asemel kahe tabeli kombineeritud pilti

Näide: `GROUP BY`

Sageli ei taheta näha kõiki ridu, vaid kokkuvõtet.

Näiteks tudengite keskmine tulemus:

```
sqlite3 andmed.db "
select s.name, round(avg(r.score), 1) as avg_score
from results r
join students s on s.id = r.student_id
group by s.id, s.name
```

```
order by avg_score desc;
"
```

Ja näiteks linnade kaupa tudengite arv:

```
sqlite3 andmed.db "
select city, count(*) as students_in_city
from students
group by city
order by students_in_city desc, city;
"
```

See on oluline vahe:

- `JOIN` toob seotud read kokku
- `GROUP BY` teeb neist koondvaate

Näide Pythoniga

SQLite ja Python sobivad hästi kokku, sest Pythonis on `sqlite3` moodul kohe olemas.

```python
import sqlite3

conn = sqlite3.connect("andmed.db")
cur = conn.cursor()

cur.execute("""
select s.name, round(avg(r.score), 1) as avg_score
from results r
join students s on s.id = r.student_id
group by s.id, s.name
order by avg_score desc
""")

for name, avg_score in cur.fetchall():
    print(f"{name}: {avg_score}")

conn.close()
```

Siin:

- Python ei asenda SQL-i, vaid kasutab seda
- SQL teeb andmete valiku ja koondamise
- Python saab tulemuse kätte ja teeb sellega edasi, mida vaja

Hea tööjaotus on sageli just selline:

- SQL: vali, ühenda, koonda
- Python: töötle, teisenda, visualiseeri, ehita suurem programm

Minitest

1. Loo tabel `students`, kus on vähemalt väljad `id`, `name` ja `city`.
2. Loo teine tabel, mis viitab tudengi `id` väärtusele võõrvõtmega.
3. Tee päring, mis kasutab `JOIN`-i, et kuvada mõlema tabeli info koos.
4. Tee päring, mis kasutab `GROUP BY`-d, et anda väike kokkuvõte.
5. Selgita ühe lausega, miks `CSV` fail ja andmebaasitabel ei ole sama asi.

Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java

Selles peatükis võrdleme lühidalt, kuidas eri keelte programmid sünnivad ja käivituvad.

Loogika

See peatükk ei ole mõeldud nende keelte õppimiseks, vaid ühe väga praktilise vahe mõistmiseks:

- mõni fail käivitatakse tõlgendiga otse
- mõni fail kompileeritakse kõigepealt teise vormi
- mõni tulemus on päris binaar
- mõni tulemus vajab eraldi runtime'i või virtuaalmasinat

See teema seob kokku:

- shebang'i ja käivitatavad skriptid
- `PATH`-i ja käskude leidmise
- Dockeri ja builditööriistad
- LaTeX-i kompileerimise idee

Kiirspikker

- `sh hello.sh` käivitab shelliskripti
- `python3 hello.py` käivitab Pythoni faili
- `cc hello.c -o hello-c` kompileerib C programmi
- `c++ hello.cpp -o hello-cpp` kompileerib C++ programmi
- `go run hello.go` kompileerib ja käivitab Go näite
- `go build -o hello-go hello.go` ehitab Go binaari
- `cargo run` ehitab ja käivitab Rusti projekti
- `javac Hello.java` kompileerib Java klassi
- `java Hello` käivitab Java klassi JVM-is

Kõige tavalisemad käsud

- `sh skript.sh`
- `./skript.sh`

- `python3 hello.py`
- `cc hello.c -o hello-c`
- `c++ hello.cpp -o hello-cpp`
- `go build -o hello-go hello.go`
- `cargo build`
- `cargo run`
- `javac Hello.java`
- `java Hello`

Üks kiire võrdlustabel

Keel	Lähtefail	Tüüpiline käivitus	Mis tekib
Shell	`hello.sh`	`sh hello.sh` või `./hello.sh`	tavaliselt eraldi buildi ei teki
Python	`hello.py`	`python3 hello.py`	lähtefail, mõnikord `__pycache__`
C	`hello.c`	`./hello-c` pärast kompileerimist	päris binaar
C++	`hello.cpp`	`./hello-cpp` pärast kompileerimist	päris binaar
Go	`hello.go`	`go run hello.go` või `./hello-go`	päris binaar
Rust	`src/main.rs`	`cargo run` või binaar `target/` all	päris binaar + `target/`
Java	`Hello.java`	`java Hello`	`.class` fail, käivitus JVM-is

Shell: skript ja shebang

Shelli näide on kõige lihtsam, sest see on lihtsalt tekstifail, mida loeb shell.

```
cat > hello.sh <<'EOF'
#!/bin/sh
echo "Tere, maailm!"
echo "Tere, maailm!"
echo "Tere, maailm!"
EOF
chmod +x hello.sh
./hello.sh
```

Siin:

- fail on tavaline tekstifail
- `chmod +x` lubab selle käivitada
- shebang `#!/bin/sh` ütleb, milline shell faili tõlgendab

Teine võimalus on:

```
sh hello.sh
```

Siis valid shelli käsurealt ise.

Python: tõlgendatud käivitus

Pythoni näide näeb välja samuti lihtne, aga erinevalt shellist käivitad sa tavaliselt faili Pythoni tõlgendiga.

```
cat > hello.py <<'EOF'
for _ in range(3):
    print("Tere, maailm!")
EOF
python3 hello.py
```

Siin:

- lähtefail jääb tekstifailiks
- `python3` loeb selle sisse ja käivitab
- tavaliselt ei teki kohe samas mõttes eraldi käivitatavat binaari

C: kompileeri binaariks

C on klassikaline näide keelest, kus lähtekood tuleb kõigepealt kompileerida.

```
cat > hello.c <<'EOF'
#include <stdio.h>

int main(void) {
    for (int i = 0; i < 3; i++) {
        puts("Tere, maailm!");
    }
    return 0;
}
EOF
cc hello.c -o hello-c
./hello-c
```

Siin:

- `hello.c` on lähtefail
- `cc ... -o hello-c` teeb sellest binaari
- `./hello-c` käivitab juba kompileeritud programmi

See on oluline vahe võrreldes shelli või Pythoniga: enne käivitamist tuleb ehitada eraldi käivitatav fail.

C++: sarnane C-ga, aga teise kompilaatoriga

```
cat > hello.cpp <<'EOF'
#include <iostream>

int main() {
    for (int i = 0; i < 3; i++) {
        std::cout << "Tere, maailm!" << std::endl;
    }
    return 0;
}
EOF
c++ hello.cpp -o hello-cpp
./hello-cpp
```

Loogika on sama nagu C puhul:

- lähtekood
- kompileerimine
- binaar

Go: lihtne tee ühe binaarini

Go on käsureaõppe jaoks väga tänulik näide, sest buildi loogika on sirge.

```
cat > hello.go <<'EOF'
package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        fmt.Println("Tere, maailm!")
    }
}
EOF
go run hello.go
go build -o hello-go hello.go
./hello-go
```

Siin on kaks eri tööviisi:

- `go run` kompileerib ja käivitab ühe hooga
- `go build` teeb päris binaari

See teeb Go-st väga hea näite, kui tahad näha vahet "jooksuta kohe" ja "ehita fail valmis".

Rust: builditööriist on osa tavalisest töövoost

Rustis kasutatakse väga sageli tööriista `cargo`.

```
mkdir -p hello-rust
cd hello-rust
cargo init --bin .
cat > src/main.rs <<'EOF'
fn main() {
    for _ in 0..3 {
        println!("Tere, maailm!");
    }
}
EOF
cargo run
cargo build --release
./target/release/hello-rust
```

Siin:

- `cargo init --bin .` teeb projekti karkassi
- `cargo run` ehitab ja käivitab
- `cargo build --release` teeb optimeerituma binaari

Rusti näide on hea, sest siin on juba näha, et mõnes keeles ei käi build ainult ühe kompilaatorikäsuga, vaid terve töövoo kaudu.

Java: kompileerimine ja eraldi runtime

Java on hea kontrast C-le ja Go-le, sest kompileerimine toimub küll enne, aga tulemus ei ole tavaliselt otse natiivne binaar.

```
cat > Hello.java <<'EOF'
public class Hello {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            System.out.println("Tere, maailm!");
        }
    }
}
EOF
javac Hello.java
java Hello
```

Siin:

- `javac` teeb failist `Hello.class`
- `java Hello` käivitab selle JVM-is

See on oluline vahe:

- C, C++, Go ja Rust annavad sulle tavaliselt natiivse binaari
- Java annab sulle klassifaili, mida käitab Java runtime

Millal tekib mis asi

Hea meelespea on:

- shell: tekstifail, mida tõlgendatakse
- Python: tekstifail, mida tõlgendatakse
- C/C++: natiivne binaar
- Go: natiivne binaar
- Rust: natiivne binaar, mida haldab `cargo`
- Java: baitkood, mida käitab JVM

Just see on põhjus, miks nende keelte buildi- ja käivitusloogika erineb.

Praktiline soovitus

Kui tahad mõtet kiiresti kätte saada, mine selles järjekorras:

1. shell
2. Python
3. C
4. Go
5. Rust
6. Java

See järjekord liigub lihtsast tõlgendamisest kuni keerukama builditööriista ja runtime'i loogikani.

Minitest

1. Tee shelliskript, mis prindib sama rea kolm korda.
2. Tee sama Pythonis.
3. Kompileeri väike C programm ja käivita see.
4. Võrdle `go run` ja `go build` käitumist.
5. Vaata, mis fail tekib pärast `javac Hello.java`.
6. Selgita ühe lausega, miks Java erineb C-st käivitamise mõttes.

LaTeX käsurealt

Selles peatükis näitame, kuidas paigaldada LaTeX ja kompileerida esimene dokument käsurealt.

Loogika

LaTeX on selles raamatus pigem järgmine samm pärast Markdowni, kui tahad jõuda väga hea PDF-väljundini.

See peatükk on seotud buildi ja dokumentide vormindamisega, mitte Linuxi baaskasutuse kõige esimese ringiga.

Milleks LaTeX kasulik on

LaTeX sobib eriti hästi siis, kui vajad:

- kvaliteetset PDF-i
- täpset vormindust
- valemeid, ristviiteid ja bibliograafiat

Kiirspikker

- `pdflatex fail.tex` kompileerib PDF-i
- `xelatex fail.tex` kasutab moodsamat tekstirenderdust
- `latexmk -pdf fail.tex` teeb korduskompileerimise mugavamaks

Minimaalne näide

```
\documentclass{article}
\begin{document}
Tere, maailm!
\end{document}
```

Salvesta see faili `tere.tex` ja kompileeri:

```
pdflatex tere.tex
```

või:

```
latexmk -pdf tere.tex
```

Paigaldus

Näited sõltuvad süsteemist:

```
sudo apt install texlive-latex-base
brew install --cask basictex
```

macOS-is piisab sageli `basictex`-ist. Kui vajad suuremat ja täielikumat LaTeX-i komplekti, kasuta selle asemel `mactex-no-gui`.

```
brew install --cask mactex-no-gui
```

Praktikas ei paigaldata `basictex` ja `mactex-no-gui` cask'e tavaliselt koos.

Minitest

1. Loo fail `tere.tex`.
2. Kompileeri see PDF-iks.
3. Kontrolli, millised väljundfailid LaTeX kõrval tekitab.

Lisa A: kopeeritavad minitestid

Siia koondame lühikesed peatükkide kaupa jaotatud copy-paste harjutused.

Need plokid on teadlikult lühikesed ja kasutavad juba mõnda varem seletatud lühikuju:

- `~` tähendab kodukataloogi
- `mkdir -p` loob puuduva teekonna vahekaustad

Failid ja kataloogid

```
mkdir -p ~/tmp/test1
cd ~/tmp/test1
touch a.txt b.txt
mkdir kaust
mv b.txt kaust/
ls -la
```

Torud ja suunamine

```
printf 'üks\nkaks\nkolm\n' | wc -l
printf 'tere\n' > proov.txt
printf 'juurde\n' >> proov.txt
cat proov.txt
```

grep ja sort

```
printf 'kass\nkoer\nkass\n' > loomad.txt
grep 'kass' loomad.txt
sort loomad.txt | uniq -c
```

Python ja venv

```
mkdir -p ~/tmp/venvtest
cd ~/tmp/venvtest
python3 -m venv .venv
source .venv/bin/activate
python -V
deactivate
```

Lisa B: spikrite register

Siia koondame kõige lühemad meelespead, mida saab kasutada kiirviitena.

Failid

- `pwd` kus ma olen
- `ls -la` mida siin näha on
- `ls -lt | head` vaata, mis viimati muutus
- `du -a . | sort -nr | less` leia suurimad failid ja kaustad
- `du -sh .[!.]* * 2>/dev/null | sort -h` võrdle ka peidetud kaustu
- `cp` kopeeri
- `mv` liiguta või nimeta ümber
- `rm fail` kustuta fail vaikselt
- `rmdir kaust` kustuta tühi kataloog
- `rm -r kaust` kustuta kataloog koos sisuga
- `mkdir -p ~/tmp/proov` loo ohutu harjutuskaust
- `sha256sum fail` arvuta faili räsi
- `find . -name 'muster'` otsi faile
- `find . -type f -size +100M` otsi suuri faile
- `find . -type f -mtime -7` otsi hiljuti muudetud faile

Tekst

- `cat` kuva fail
- `less` sirvi faili
- `less` sees `78g` mine reale 78
- `less` sees `25%` mine 25% peale faili sees
- `tail -f logi.txt` jälgi kasvavat logi
- `tail -n 50 logi.txt | less` sirvi viimaseid logiridu
- `grep` otsi
- `grep -R 'muster' .` otsi rekursiivselt tervest puust
- `sort` sorteeri
- `uniq -c` loenda kordused
- `seq -w 0 99 | pr -5 -t` pane numbrid mitmesse veergu
- `seq -w 0 9999 | pr -8 -t -l 1250` tee üks `pr` loogiline leht

Õigused

- `ls -l` vaata õigusi
- `chmod +x fail` tee käivitatavaks
- `chown kasutaja:grupp fail` muuda omanikku

Võrk

- `ssh host` logi sisse

- `scp fail host:/tee/` kopeeri üle võrgu
- `rsync -av allikas/ host:/siht/` sünkroniseeri

Arendus

- `git status`
- `git diff`
- `git diff --cached`
- `git add`
- `git commit`
- `python3 -m venv .venv`
- `python3 -u skript.py` näita Pythoni väljundit kohe
- `cat > skript.sh <<'EOF'` loo mitmerealine skript here-doc'iga
- `cc hello.c -o hello-c` kompileeri C programm
- `go build -o hello-go hello.go` ehita Go binaar
- `cargo run` ehita ja käivita Rusti projekt
- `javac Hello.java` kompileeri Java klass
- `docker run --rm image käsk`
- `docker run --rm -it -v "$PWD":/app -w /app python:3.13-slim bash` ava projekt konteineris
- `docker build -t nimi .` ehita image
- `docker compose up --build` käivita mitme teenuse arenduskomplekt
- `docker compose logs -f app` jälgi teenuse logisid
- `docker compose exec app bash` sisene töötavasse teenusesse
- `column -s, -t < fail.csv | less -S` kuva lihtne CSV tabelina
- `python3 -m json.tool fail.json | less` vaata JSON-i loetavalt
- `jq '.voti' fail.json` vali JSON-ist välju
- `xmllint --format fail.xml | less` vorminda XML loetavaks
- `sqlite3 andmed.db '.tables'` kuva SQLite tabelid
- `sqlite3 andmed.db 'select * from tabel limit 5;'` kuva paar esimest rida
- `sqlite3 andmed.db 'select a.name, b.score from b join a on a.id=b.a_id;'` tee lihtne `JOIN`

Loogika

- `käsk1 ; käsk2` käivita käsud järjest
- `|` suuna väljund edasi
- `>` kirjuta faili
- `>>` lisa faili lõppu
- `2>` kirjuta vead eraldi faili
- `> fail 2>&1` kirjuta nii väljund kui vead samasse faili
- `tee fail` näita ekraanil ja kirjuta faili
- `&&` jätka ainult edu korral
- `||` jätka vea korral
- `echo $?` kuva eelmise käsu exit code

- `set -o pipefail` ära peida toru sees tekkinud vigu
- `type nimi` vaata, kas nimi on alias, builtin või programm
- `command -v nimi` vaata, mida shell käivitaks

Protsessid

- `käsk &` saada töö taustale
- `ps aux` vaata protsesse
- `top` või `htop` jälgi protsesse
- `kill PID` lõpeta protsess
- `kill %1` lõpeta shelli töö numbri järgi
- `jobs` näita shelli töid
- `bg %1` jätka tööd 1 taustal
- `fg %2` too töö 2 ette
- `wait` oota taustatööd ära
- `nohup käsk > fail 2>&1 &` jäta töö sessioonist vähem sõltuvaks
- `disown %1` eemalda töö shelli tööde nimekirjast
- `ps aux | sort -nrk 3 | head` vaata CPU sööjaid
- `ps aux | sort -nrk 4 | head` vaata mälusööjaid

Ajalugu

- `history` kuva käsuajalugu
- `history | tail -n 20` kuva viimased ajalookirjed hiljem, kui torud on juba selged
- `alias h='history | tail -n 20'` tee ajaloo lühikäsk
- `!!` korda eelmist käsku
- `!n` korda ajaloo kirjet numbri järgi
- `!sona` korda viimast sobivat käsku

Lisa C: sõnastik ja terminoloogia

See lisa on ühtaegu:

- lugeja jaoks lühike sõnastik
- tulevaste muudatuste jaoks terminoloogiline alus

Kui raamatu eesti keelt hiljem muudetakse või laiendatakse, tasub eelistada siin toodud kujusid kogu raamatu ulatuses.

Toimetuspõhimõtted

Raamatus kasutame üldiselt neid eelistusi:

- kasutame `kataloog`, mitte `folder`
- kasutame `haru`, mitte `branch`, välja arvatud siis, kui viidatakse käsule või kasutajaliidese terminile

- kasutame `konteiner`, mitte `container`, kui jutt ei ole käsu süntaksist
- kasutame `virtuaalkeskkond`, kui räägime mõistest, ja `venv`, kui viitame konkreetsele tööriistale või käsule
- kasutame `lipp`, kui räägime käsurea lühikesest või pikast võtmekujust praktilises tähenduses
- kasutame `valik`, kui mõeldakse üldisemat käsu käitumist või valikute perekonda
- kasutame `repo` kui praktilist Git-i lühivormi; pikem kuju on `repositoorium`
- kasutame `build` skriptide ja failinimede kontekstis, aga jooksvas tekstis sobib sageli paremini `koostamine`

Üldmõisted

- `terminal`: tekstipõhine keskkond, kus käske sisestatakse
- `käsurida`: üks konkreetne käsk koos argumentidega
- `shell`: käsutõlk, mis loeb käsurida ja käivitab käske
- `käsk`: programm või shelli sisseehitatud toiming, mida käsurealt käivitatakse
- `argument`: käsule etteantud sisend, näiteks failinimi või muster
- `lipp`: käsu valik, tavaliselt kujul `-n` või `--help`
- `valik`: üldisem nimetus käsu lisakäitumise määramiseks
- `sisend` ehk `stdin`: andmed, mida käsk loeb
- `väljund` ehk `stdout`: tavaline väljund, mida käsk kirjutab
- `veaväljund` ehk `stderr`: eraldi väljund vigade ja hoiatuste jaoks
- `puhverdamine` ehk `buffering`: olukord, kus programm kogub väljundi ajutiselt kokku enne, kui selle ekraanile, faili või torusse edasi saadab
- `flush`: puhvri kohene tühjendamine, et väljund jõuaks kohe nähtavale või edasi järgmisse kohta
- `toru`: kuju `|`, millega ühe käsu väljund suunatakse teise käsu sisendiks
- `ümbersuunamine`: väljundi või sisendi suunamine faili või mujale
- `exit code`: käsu lõpetuskood; tavaliselt `0` tähendab edu

Failid ja süsteem

- `fail`: andmeüksus failisüsteemis
- `kataloog`: koht, mis sisaldab faile ja teisi katalooge
- `tee` ehk `path`: failini või kataloogini viiv asukoht
- `kodukataloog`: kasutaja isiklik põhikataloog, sageli `~`
- `peidetud fail`: tavaliselt punktiga algav fail või kataloog, mida paljud tööriistad vaikimisi ei näita
- `punktiga algav nimi`: fail või kataloog nimega nagu `.zshrc` või `.git`; seda nimetatakse sageli ka peidetud kirjeks
- `õigused`: reeglid, mis määravad lugemise, kirjutamise ja käivitamise
- `omanik`: kasutaja, kellele fail kuulub
- `grupp`: kasutajate rühm, mille järgi saab õigusi jagada

- `root`: süsteemi eriline administraatori kasutaja, kellel on väga laiad õigused
- `sudo`: tööriist, millega käivitatakse üks käsk ajutiselt kõrgemate õigustega
- `täitmisõigus`: õigus faili käivitada
- `täitmisbitt`: faili täidetavust märkiv õiguste osa
- `rekursiivne`: tegevus, mis läheb ka alamkataloogidesse ja nende sisu kallale
- `force` ehk `-f`: käitumine, mis surub maha osa hoiatusi või kinnitusi; seda tuleb kasutada ettevaatlikult
- `räsi`: lühike sõrmejälg, mis kirjeldab faili sisu
- `krüptoräsi`: räsi, mida kasutatakse tervikluse kontrolliks, näiteks `SHA-256`

Shell ja tekstitöötlus

- `globbing`: shelli mustrilaiendus kujudele nagu `*`, `?`, `[]`
- `quote'imine`: erimärkide mõju piiramine jutumärkide abil
- `escape'imine`: ühe märgi erikäitumise väljalülitamine, tavaliselt `\` abil
- `shellimuutuja`: jooksva shelli sees hoitav muutuja
- `keskkonnamuutuja`: muutuja, mis antakse edasi alamprotsessidele
- `alias`: lühinimi mõnele pikemale käsule
- `shelli sisseehitatud käsk` ehk `builtin`: käsk, mis on shelli enda sees, mitte eraldi programmina kettal
- `reserveeritud sõna` ehk `keyword`: shelli süntaksi osa nagu `if`, `then`, `for`, `do`, `done`
- `shelli funktsioon`: shellis defineeritud käsuplokk, mida saab nimega käivitada
- `regulaaravaldis`: mustrikeel tekstis vastete leidmiseks
- `sõne`: täpne tekstijupp, mida ei tõlgendata regulaaravaldisena
- `filter`: käsk, mis loeb ridu ja väljastab neist ainult vajaliku osa

Võrk ja kaugkasutus

- `host`: võrgus olev sihtmasin; sageli praktiliselt sama mis serveri aadress
- `server`: masin või teenus, kuhu ühendutakse
- `port`: numbriline võrgukanal teenuse jaoks
- `SSH`: turvaline protokoll kaugmasinasse logimiseks ja käskude käivitamiseks
- `võtmepaar`: avaliku ja privaatse võtme paar autentimiseks
- `port forwarding`: võrguühenduse suunamine ühest pordist teise
- `WSL` ehk `Windows Subsystem for Linux`: viis käitada Windowsis Linuxi kasutajaruumi

Git ja GitHub

- `repo`: Git-i hoidla või repositoorium

- `haru`: eraldi arendusjoon Git-is
- `commit`: loogiline muudatuse salvestus Git-is
- `remote`: kaugrepo, millega lokaalne repo suhtleb
- `origin`: vaikimisi peamise kaugrepo nimi
- `tag`: nimetatud tähis mõne commit'i juures
- `väljalase` ehk `release`: teadlikult välja antud versioon, tavaliselt seotud kindla tag'iga
- `snapshot`: säilitamiseks tehtud väljundikoopia, mida järgmine build üle ei kirjuta
- `verstapost`: oluline seis, mis tasub eraldi nime all alles hoida
- `pull request`: GitHubi arutelupõhine muudatusettepanek harust teise
- `diff`: muudatuste vaade enne või pärast commit'i
- `stage`: Git-i vaheala, kuhu valitakse järgmisse commit'i minevad muudatused

Paketid ja arenduskeskkond

- `pakett`: paigaldatav tarkvaraüksus või sõltuvus
- `paketihaldur`: tööriist pakettide paigaldamiseks, eemaldamiseks ja uuendamiseks
- `sõltuvus`: teek või pakett, mida projekt vajab
- `virtuaalkeskkond`: eraldatud keskkond projektisõltuvuste jaoks
- `IDE`: integreeritud arenduskeskkond
- `koostamine` ehk `build`: lähtefailidest kasutatava väljundi tekitamine
- `kompileerimine`: lähtekoodi või dokumendi tõlkimine teise vormi, näiteks PDF-iks
- `tõlgendaja` ehk `interpreter`: programm, mis loeb lähtekoodi ja käivitab seda otse
- `kompilaator`: programm, mis tõlgib lähtekoodi teise vormi, sageli binaariks või baitkoodiks
- `binaar`: kompileeritud käivitatav fail masina jaoks
- `baitkood`: vahevorm, mida käitab eraldi runtime või virtuaalmasin
- `runtime`: käivituskeskkond, mida programm tööks vajab
- `JVM`: Java Virtual Machine, mis käivitab Java klassifaile ja baitkoodi
- `Homebrew` ehk `brew`: levinud paketihaldur macOS-is
- `PowerShell`: Windowsi käsukeskkond ja skriptikeel

Andmed ja andmebaasid

- `CSV`: lihtne tabelivorming, kus väljad on tavaliselt komadega eraldatud
- `JSON`: võtme-väärtuse ja massiivide vorming, mida kohtab palju API-des ja seadistusfailides
- `XML`: märgendipõhine puustruktuuriga vorming
- `relatsiooniline andmemudel`: viis kirjeldada andmeid tabelite ja nende seoste kaudu
- `rida`: üks kirje tabelis

- `veerg`: üks omadus või väli tabelis
- `primaarvõti` ehk `primary key`: väli, mis eristab iga rea teistest
- `võõrvõti` ehk `foreign key`: väli, mis viitab teise tabeli primaarvõtmele
- `JOIN`: SQL-i operatsioon, mis seob ridu eri tabelitest
- `skeem`: andmebaasi struktuuri kirjeldus, näiteks tabelid, väljad ja seosed

Docker

- `image`: valmis konteineri aluskihtide kogum
- `konteiner`: töötav isoleeritud protsess või protsesside komplekt image'i põhjal
- `registry`: koht, kust image'eid hoitakse ja kust neid alla laaditakse
- `bind mount`: hostmasina kindla tee sidumine konteineri teega
- `named volume`: Dockeri hallatav püsiv andmeala
- `arenduskonteiner` ehk `devcontainer`: IDE-ga seotud Dockeri-põhine arenduskeskkond

Dokumendid

- `Markdown`: lihtne märgistuskeel tekstidokumentide kirjutamiseks
- `LaTeX`: märgistus- ja küljendussüsteem kvaliteetsete dokumentide jaoks
- `PDF`: lõppväljund jagamiseks või printimiseks
- `HTML`: veebis kuvamiseks sobiv väljund

Lisa D: mis veel on puudu ja mida lisada järgmisena

See lisa ei ole kriitika olemasolevale materjalile, vaid järgmise toimetusringi töölaud. Viimane suurem ring tõi sisse:

- failisüsteemi kaardi
- kettaruumi peatüki
- esimese shelliskripti
- lihtsa veaotsingu
- võrgu põhitööriistad
- logid ja teenused
- `tmux`/`screen`
- `find`/`xargs` ohutuma kuju
- `cron`-i alused

Seega on nüüd olemas palju tugevam tervik kui varem. Järgmine küsimus ei ole enam "mis kõige tähtsam puudub", vaid "mis järgmine sügavusaste annaks kõige rohkem juurde".

Loogika

Praegune käsikiri on nüüd tugev algaja ja varase kesktaseme kombinatsioon:

- käsurea baas on olemas
- süsteemi pilt on olemas
- failide, võrgu ja tekstivoo tööriistad on olemas
- arendaja töövood on olemas

Edasised lisad peaksid nüüd pigem süvendama, mitte lihtsalt katma esmast baasi.

Tugevad järgmised kandidaadid

1. R ja notebook'ide sild

Kui siht on andmeteaduse või analüüsi suund, siis järgmine loomulik täiendus oleks:

- `R`
- Jupyter või notebook'i loogika
- millal kasutada shelli, millal SQL-i, millal Pythonit või R-i

2. Git-i järgmine aste

Praegu on Git-i baas olemas, aga järgmine tugev samm oleks:

- harud
- merge
- rebase põhimõte
- konfliktide lugemine
- remote ja origin loogika

3. Shelliskriptide teine aste

Pärast esimest skripti võiks järgmine ring tuua:

- funktsioonid
- `case`
- `set -euo pipefail`
- ajutised failid
- veakindlam sisenditöötlus

4. Statistika ja matemaatika sild

See õpik ei pea muutuma statistikakursuseks, aga kasulik oleks üks lühike peatükk, mis sõnastab:

- miks tõenäosusteooria on andmetöö juures tähtis
- mis vahe on andmete vaatamisel ja järelduste tegemisel
- millal shelli või SQL-i oskus ei asenda statistilist mõtlemist

5. `systemd` timerid

Pärast `cron`-i oleks loogiline järgmine samm Linuxi poolel:

- timerid
- service + timer koos
- millal timer on mõistlikum kui `cron`

6. Backup ja taastamine

Praegu on kopeerimine ja sünkroonimine olemas, aga eraldi ülesandepõhine peatükk võiks katta:

- varukoopia tegemise põhimõtted
- testitud taastamise tähtsuse
- checksum'id
- snapshot'i ja arhiivi vahe

Mida ma ikka veel ei lisaks esimesena

Need teemad võivad olla huvitavad, kuid ei anna veel kõige suuremat võitu:

- väga sügav `awk` või `sed`
- keeruline Docker Compose maailm
- prompt'i peenhäälestus
- kerneli või süsteemikutsete süvateooria

Need sobivad paremini järgmisteks väljaanneteks või eraldi edasijõudnute peatükkideks.

Hinnang praegusele materjalile

Praegune käsikiri on minu hinnangul:

- sisuliselt tugev
- praktiline
- hästi kasutatav referents
- algajale päriselt navigeeritav

Kõige tugevamad küljed:

- palju kopeeritavaid näiteid
- loogikaseletused enne käsuloendeid
- tugevam süsteemipilt kui varem
- nüüd ka selgem peatükkide hierarhia

Järgmine suur kvaliteedihüpe tuleks ilmselt mitte enam "rohkemate baaspeatükkide", vaid mõne valitud teema sügavamast teisest astmest.