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-72-g420c5e5

Commit: 420c5e5

Tag: v0.2.0-draft

Kuupäev: 2026-04-30 22:10:34 EEST

Sisukord

- 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: edasised võimalused ja lugejate soovid
- Lisa E: usaldusväärsed viited ja lisalugemine
- Lisa F: shelli seadistusfailid bash ja zsh jaoks

Kuidas seda õpikut kasutada

See õpik on praktiline tööriist:

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

Kui oled täiesti algaja, alusta nii

Kui sul puudub käsurea kogemus, siis ei pea sa kogu raamatut korraga haarama. Rahulik esimene rada on:

1. Terminali esimesed sammud
2. Abi leidmine: man, –help ja info
3. Kataloogid ja failid
4. Teksti vaatamine ja liikumine
5. Failide muutmine: nano ja esimene kokkupuude vim-iga
6. Käskude kuju ja argumentide loogika
7. Sisend, väljund, torud ja suunamine
8. Esimene tervikharjutus: 30 minutit

Tööta alguses oma harjutuskaustas, tee väikesi samme ja kontrolli iga sammu järel tulemust. Täpsema teekaardi eri eesmärkide jaoks leiad peatükist Õpitee ja õppetunnid.

Kui tahad üsna alguses turvalist harjutuskohta, kus saab käske läbi proovida ja ülesandeid lahendada, siis võid kasutada ka Tartu Ülikooli keskkonda terminal.cs.ut.ee. Seal saad katsetada valmis harjutustega ilma, et peaksid omaenda masinas kohe kõike seadistama.

Mida sa ei pea veel teadma

Esimeses ringis ei pea sa veel tundma:

- regulaaravaldisi
- Git-i, SSH-d ega Dockerit
- keerulisemaid käsuahelaid ja töövooge
- süsteemi administraatori võtteid

Piisab sellest, kui oskad terminalis liikuda, abi vaadata, faile leida ja väikseid muudatusi turvaliselt kontrollida.

Põhimõisted: CLI ja GUI

Selles õpikus kohtad sageli sõna `CLI`. See tuleb ingliskeelsest väljendist `command-line interface`.

Selle kõige loomulikum eestikeelne vaste on:

- `CLI` ehk käsurealiides

Tavaline alternatiiv sellele on:

- `GUI` ehk graafiline kasutajaliides

Praktiline vahe on lihtne:

- `CLI` tähendab, et suhtled arvutiga käske kirjutades
- `GUI` tähendab, et suhtled akende, nuppude, menüüde ja ikoonide kaudu

Terminal on tavaliselt see programm või aken, mille kaudu `CLI`-d kasutatakse. Shell on omakorda käsutõlk selle terminali sees.

Õpik keskendub peamiselt `CLI`-le, kuid võrdleb seda vajadusel ka `GUI`-ga.

Kuidas peatükki lugeda

Enamik peatükke töötab kõige paremini nii:

1. loe kõigepealt loogikaosa, et saada aru, milleks seda tööriista vaja on
2. vaata süntaksit või kiirspikrit
3. proovi lühikesed näited ise läbi
4. tee minitest või harjutus

Kui mõni sümbol, lipp või käsk tundub liiga vara võõras, peatu korraks ja mine eelmise sammu juurde tagasi. Õpik on mõeldud kasutamiseks väikeste, kontrollitavate osadena.

Näidete põhimõte

Näited peaksid olema:

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

Hea näide õpetab ühe töövõtte loogikat ja näitab kohe tulemust.

Näiteks:

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

Siin kontrollid alguskohta, lood harjutuskausta, kirjutad faili kaks rida ja kontrollid tulemuse üle.

Kaks lugemisviisi

Õpikut tasub kasutada kahel viisil:

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

Kui tahad lisaks õpikule ka ametlikke ja usaldusväärseid algmaterjale, siis vaata lisa Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Saa aru, mis on terminal, shell, CLI ja GUI, ning vali rahulik algusrada enne keerulisemaid töövooge.

Põhitee

- `Terminali esimesed sammud` — alusta siit
- `Abi leidmine` — otsi tuge

- `Kataloogid ja failid` — tee muudatusi
- `Teksti vaatamine` — loe rahulikult

Tüüpilised kujud

- `pwd` — kontrolli asukohta
- `ls` — vaata sisu
- `man ls` — loe abi
- `cd ~/tmp` — mine tmp-kausta; sobib harjutamiseks

Põhimõisted

- `CLI` — käsurida
- `GUI` — graafiline liides
- `terminal` — aken käsureaks
- `shell` — tõlgendab käske

Pane tähele: Sa ei pea esimeses ringis veel teadma Git-i, Dockerit ega keerulisemaid töövooge.

Edasi: Järgmine loomulik samm: Õpitee ja õppetunnid.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Õpitee ja õppetunnid

See peatükk aitab valida, mis järjekorras lugeda. Raamatut saab 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

Kiirtee: algajast töövõimeliseks kasutajaks

Staatus: soovitatav, kui tahad kiiresti saada praktiliselt kasutatavaks käsurea kasutajaks.

See ei ole kõige rahulikum rada, aga viib kiiresti seisuni "ma saan päris asju tehtud".

Vaata peatükke selles järjekorras:

1. Terminali esimesed sammud
2. Abi leidmine: man, –help ja info
3. Kataloogid ja failid
4. Sisend, väljund, torud ja suunamine
5. Lihtne veaotsing käsureal
6. Failide kopeerimine ja sünkroonimine
7. Kauglogimine ja SSH
8. Logid ja teenused
9. Git, GitHub ja töövoog

Pärast seda kiirteed oskad juba:

- liikuda terminalis kindlamalt ja otsida abi ilma paanikata
- kopeerida faile, ühendada käske torudega ja lugeda lihtsamaid veateateid
- logida serverisse, tuua või saata faile ja vaadata logidest, mis toimub
- teha väikese muudatuse Gitis ning kontrollida enne commit'i, mis päriselt muutus

Õpitee 1: täiesti algaja

Staatus: vajalik esimeses ringis.

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 muutmine: nano ja esimene kokkupuude vim-iga
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

Pärast seda rada 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

Staatus: vajalik pärast esimest ringi.

Kui baas on all, liigu 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 aitab süsteemi "musta kasti" lahti võtta.

Pärast seda rada oskad:

- paigutada tähtsamad süsteemikaustad ja tööriistad õigesse konteksti
- lugeda õigusi, kasutajaid ja `PATH`-i ilma liigse müstikata
- aru saada, kust tulevad paketid, käsud ja paljud tüüpilised vead

Õpitee 3: igapäevane Linuxi ja serveri kasutaja

Staatus: soovitatav, kui töötad serverite, kaugühenduste või suuremate failipuudega.

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 rada 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

Pärast seda rada oskad:

- kopeerida faile ja katalooge masinate vahel
- logida sisse SSH kaudu ja kasutada võtmeid turvalisemalt
- lugeda logisid ning hallata pikemalt jooksvaid töid

Õpitee 4: tekst, filtrid ja automatiseerimine

Staatus: soovitatav kohe pärast baasi, kui tahad käsurea päris jõudu kasutada.

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 tähtsamaid õpiteid, sest siin tekib "väikeste tööriistade ühendamise" tunnetus.

Pärast seda rada oskad:

- otsida ja filtreerida ridu mustri järgi
- sortida, loendada ja teisendada tekstivooge
- kirjutada esimesi lühikesi shelliskripte ja ajastatud töid

Õpitee 5: arendaja suund

Staatus: soovitatav, kui kasutad käsurida tarkvaraprojektides.

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:

- 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 näitavad, kuidas käsurida seob andmed, programmid ja dokumendid üheks töövooks

Pärast seda rada oskad:

- hoida projekti muudatusi Git-is korras
- isoleerida sõltuvusi ja arenduskeskkondi
- lugeda, käivitada ja ehitada mitut tüüpi arendusprojekte

Õpitee 6: andmeteaduse stardirada

Staatus: soovitatav, kui tahad käsurida kasutada andmete puhastamiseks ja uurimiseks.

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, sest:

- 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

Pärast seda rada oskad:

- töödelda tekstifaile ja tabeleid käsureal
- eristada CSV-d, JSON-i ja XML-i
- teha esimesi SQL-päringuid ning siduda see Pythoni töövooga

Minimaalne 7 päeva plaan

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

1. päev: Terminali esimesed sammud, Abi leidmine, Kataloogid ja failid

2. päev: Teksti vaatamine ja liikumine, Failide muutmine, Käskude kuju
3. päev: Sisend, väljund, torud ja suunamine, Esimene tervikharjutus, Linux, Unix, GNU, macOS, Windows ja shellid
4. päev: Failisüsteemi kaart, Kettaruum, Õigused
5. päev: Kasutajad, grupid ja sudo, Muutujad ja PATH, Paketihaldus, Lihtne veaotsing
6. päev: Võrgu põhitööriistad, Failide kopeerimine ja sünkroonimine, Kauglogimine ja SSH, Veebist sisu toomine
7. päev: grep, Teksti teisendamine, sort, uniq, wc, pr, join, Esimene shelliskript, Git, GitHub ja töövoog

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

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: See peatükk aitab valida, mis järjekorras lugeda. Raamatut saab kasutada ka käsiraamatuna, kuid alguses on lihtsam liikuda kindla õpitee järgi.

Õpiteed

- `Täiesti algaja` — loe esimesena
- `Süsteemi pildi loomine` — pärast esimest ringi
- `Igapäevane Linuxi ja serveri kasutaja` — kaugühendus ja failid
- `Tekst, filtrid ja automatiseerimine` — käsurea jõud
- `Arendaja suund` — Git, Docker, build
- `Andmeteaduse stardirada` — failid, vormingud, SQL

Soovitatud järjestused

- `Terminali esimesed sammud → Abi leidmine → Kataloogid ja failid → Teksti vaatamine ja liikumine` — algaja põhirada
- `Failide muutmine → Käskude kuju → Torud ja suunamine → Esimene tervikharjutus` — seo alus tervikuks
- `Failisüsteemi kaart → Kettaruum → Õigused → Võrgu põhitööriistad` — ehita süsteemi pilt
- `grep → Teksti teisendamine → sort, uniq, wc → Kompileerimine ja käivitamine` — filtrid ja arendus

Märgendid ja kontroll

- `vajalik` — loe nüüd
- `soovitatav` — hea järgmine
- `hiljem` — ära kiirusta
- `Pärast seda rada oskad …` — kontrolli edenemist

Pane tähele: Ära püüa kogu raamatut korraga läbi võtta; vali kõigepealt üks rada ja liigu selle sees rahulikult edasi.

Edasi: Järgmine loomulik samm: Terminali esimesed sammud.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Terminali esimesed sammud

Terminalis kirjutad käsu reale, vajutad `Enter` ja näed vastust. Enne esimesi käske on kasulik eristada kolme sõna: terminal, shell ja viip.

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. Terminal, shell ja viip

`Terminal` on tekstipõhine aken või rakendus. Selle sees töötab tavaliselt `shell`, näiteks `zsh` või `bash`. Shell loeb sinu käsurea, tõlgendab selle käsuks ja käivitab vastava programmi või shelli enda toimingu.

Lihtne pilt:

- `terminal` on aken, kus sa kirjutad ja näed vastust
- `shell` on programm terminali sees, mis käske tõlgendab
- `viip` ehk `prompt` on shelli kuvatav kutse: nüüd võid käsu sisestada

Näiteks:

```
~/proov %
```

või:

```
kasutaja@arvuti:~$
```

Viip näitab tavaliselt mõnda neist asjadest:

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

Viiba täpne kuju võib olla erinev. Hea rusikareegel on:

- viip on shelli kuvatav kasutajaliidese osa
- `pwd` ütleb kindlalt, kus sa päriselt oled

Kui tahad teada, milline shell parajasti töötab, küsi:

```
echo "$SHELL"
```

See näitab tavaliselt shelli teed, näiteks `/bin/zsh` või `/bin/bash`.

Hiljem kohtad ka shelli muutujaid nagu `PS1` ja `PROMPT`. Need ei kuulu terminaliaknale, vaid shellile: nende abil otsustab shell, millise viiba terminalis kuvab.

Kui oled lihtsalt uudishimulik, võid vaadata:

```
echo "$PS1"
```

zsh-s võib kasulik olla ka:

```
print -r -- "$PROMPT"
```

Need käsud näitavad viiba tehnilist kuju, mitte alati täpselt sama pilti, mida terminal ekraanil kuvab.

Oluline: ära kopeeri viipa käsu ette kaasa

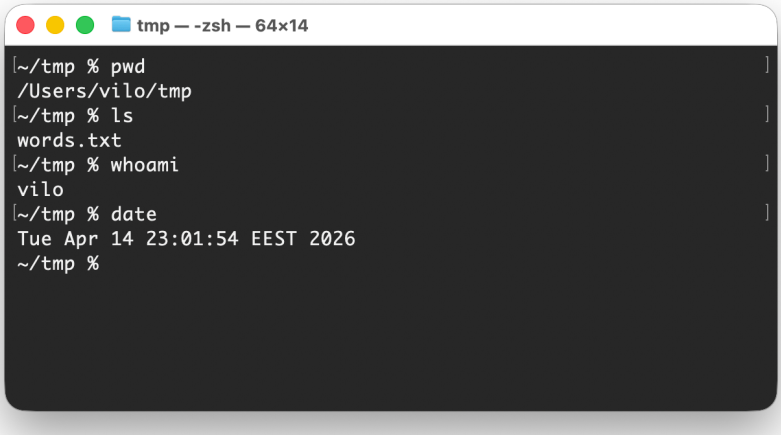
Kui terminalis on näha näiteks:

```
~/proov % pwd
```

siis kopeeritav käsk on ainult:

```
pwd
```

Viip ise ei ole käsu osa. See on shelli kuvatav tähis, mis näitab, et shell ootab sinu sisestust.



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

Pildi mõte:

1. viip on lühike, et käsud oleksid loetavad
2. `pwd`, `ls`, `whoami` ja `date` ainult näitavad infot
3. ükski neist käskudest ei muuda faile

2. Esimesed ohutud käsud

Alusta nende neljaga:

```
pwd
ls
whoami
date
```

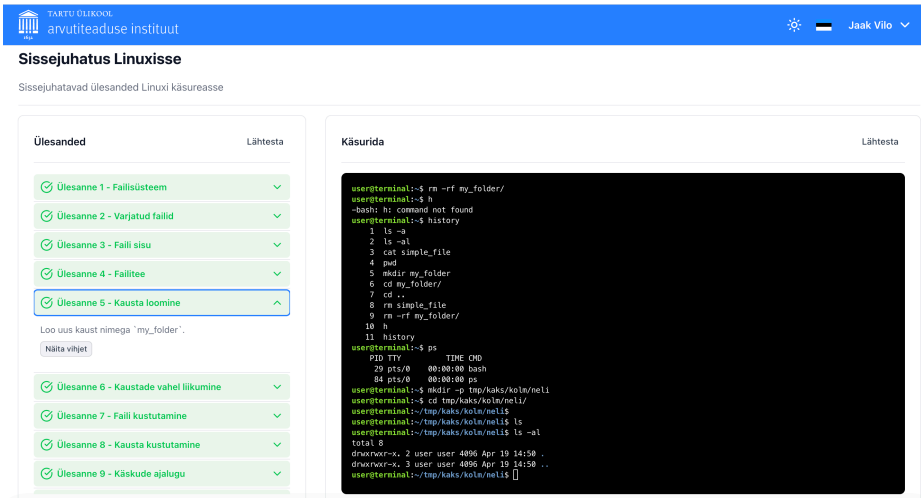
Need näitavad, kus sa oled, mis sinu ümber on, kes sa oled ja mis kell süsteemi arvates on.

Kui tahad harjutada valmis keskkonnas

Kui sul ei ole oma Linuxi masinat käepärast või tahad lahendada valmis ülesandeid, siis on hea varajane harjutuskoht terminal.cs.ut.ee.

See sobib alguses hästi, sest:

- saad käske kohe läbi proovida
- saad lahendada väikseid sissejuhatavaid ülesandeid
- ei pea enne kõike oma masinas valmis seadistama



Joonis 2: Tartu Ülikooli `terminal.cs.ut.ee` harjutuskeskkond, kus vasakul on näha sissejuhatavad ülesanded ja paremal terminaliaken, milles saab käske kohe läbi proovida.

Pildi mõte:

1. vasakul on ülesannete nimekiri, mida saab ükshaaval avada
2. paremal on terminaliaken, kus käske kohe läbi proovida
3. keskkond sobib eriti hästi varajasteks katsetusteks ja väikesteks kontrollitud ülesanneteks

Kui tahad oma terminaliaknas samasugust lihtsat prompti, siis ajutised käsud on:

`zsh` jaoks:

```
PROMPT='user@terminal:%~$ '
```

`bash` jaoks:

```
PS1='user@terminal:\w$ '
```

Need muudavad viipa ainult käesolevas shellis. Uues aknas tuleb tagasi sinu tavaline viip.

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: kirjuta nii palju nime algusest, kui tead, vajuta `Tab`, ja kui sellest ei piisa, lisa mõni täht juurde.

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: kaust on sinu enda alal ja päris projektid ei lähe 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 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.

Hiljem tasub teada

On olemas ka kiiremad ajaloo-otseteed nagu `!!`, `!25` ja `!ls`. Need käivitavad mõne vana käsu väga kiiresti uuesti. Algaja põhivoos ei ole neid veel vaja: rahulikum ja turvalisem on kasutada esialgu `history`, nooleklahve ja `Ctrl-r`.

8. Esimesed kasulikud klahvid

Mõned klahvikombinatsioonid aitavad juba esimestel päevadel väga palju:

- `Ctrl-c` katkestab parajasti töötava programmi või käsu töö
- `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:

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

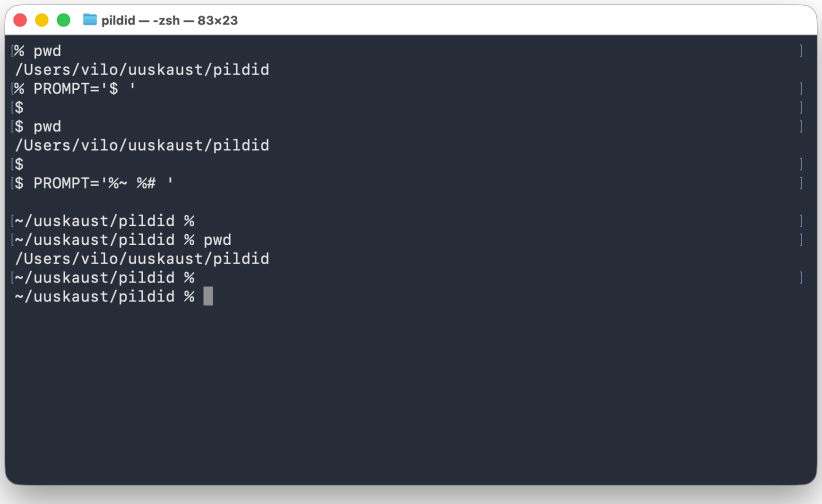
Seega uus viip ei tähenda automaatselt viga. Mõnikord tähendab see lihtsalt, et käsul ei olnud midagi ekraanile näidata.

10. Viip võib olla eri kujuga, aga `pwd` kontrollib asukohta

Pea meeles peatüki alguse põhimõtet: viip on shelli kuvatav rida, mitte kindel tõend sinu asukoha kohta. Sama töökausta võib näidata väga erineva viibaga.

```
$ pwd
/Users/kasutaja/proov
```

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



Joonis 3: Terminali näide, kus viip tehakse järjest lühemaks: kõigepealt on näha pikk tee, siis ainult `$` ja lõpuks lühike kuju `~/uuskaust/pildid %`.

Pildi mõte:

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

Järeldus: viiba kuju võib muutuda, aga tegelik asukoht tuleb endiselt käsust `pwd`.

11. 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. Too eelmine käsk uuesti ette ülesnoole abil või otsi seda `Ctrl-r` abil.
6. Selgita ühe lausega, miks `pwd` on sageli kindlam kui viiba kuju.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Vaata kõigepealt rahulikult, kus sa oled, kes sa oled ja mis selles kaustas on; alles siis tee esimene väike muudatus.

Käsud ja esimesed sammud

- `pwd` — vaata asukohta
- `ls` — vaata sisu
- `whoami` — vaata kasutajat
- `date` — vaata praegust kuupäeva ja aega
- `history` — vaata käsuajalugu
- `cd ~/tmp` — mine tmp-kausta; sobib harjutamiseks
- `mkdir proov && cd proov` — loo proovikaust

Olulisemad lipud, märgid ja kiirnupud

- `Tab` — lõpeta nimi
- `Ctrl-c` — katkesta programmi töö
- `Ctrl-r` — otsi ajaloost
- `Ctrl-a` — rea algus
- `Ctrl-e` — rea lõpp
- `~/proov %` — näide viibast ehk promptist

Pane tähele: Ära kopeeri viipa käsu ette kaasa; kopeeri ainult käsk ise.

Edasi: Järgmine loomulik samm: Abi leidmine: man, –help ja info.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

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.

Esimene abirada

Kui käsk on võõras, liigu tavaliselt selles järjekorras. Nii saad kõigepealt kohaliku, sinu süsteemi kohta käiva info.

Käsk või kuju	Milleks	Mida tavaliselt näed
`man käsk`	rahulik tervikpilt	lehitsetav käsiraamat
`käsk --help`	kiire meeldetuletus	lühike tekst otse terminalis
`whatis käsk`	ühe käsu lühikirjeldus	üks kirjeldav rida
`apropos sõna`	otsi käsu nime teema järgi	mitu võimalikku vastet
`info teema`	pikem GNU dokumentatsioon	eraldi lehitsetav abivaade

Hea harjumus on otsida abi kõigepealt käsu enda dokumentatsioonist, mitte loota juhuslikule veebinäitele.

Tüüpilised algaja vead

- ehmatatakse ära, kui `man` “võtab terminali üle”
- arvatakse, et `--help` peab kõigil käskudel töötama täpselt sama moodi
- minnakse kohe veebi, kuigi kohalik abi annaks vastuse kiiremini

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. Kui `man` puudub või `--help` ei tööta

Mõnes minimaalses süsteemis, konteineris või eriprogrammis võib juhtuda, et:

- `man` ei ole paigaldatud
- käsk ei tunne kuju `--help`
- abi tuleb hoopis mõne teise võtmega

Siis tasub liikuda nii:

1. proovi kõigepealt `käsk --help`
2. kui see ei tööta, vaata, kas dokumentatsioon või veateade vihjab kujule `-h`
3. kasuta `whatis` või `apropos`, et leida õige käsu nimi
4. kui vajadus jääb püsima, tule hiljem tagasi paketihalduri peatüki juurde ja paigalda puuduv dokumentatsioon

Oluline mõte on lihtne: kui üks abivõte ei tööta, siis see ei tähenda veel, et abi üldse puudub.

4. `-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`

5. 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
```

6. 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

7. 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`.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Kui sa ei mäleta käsu kuju või lippu, vaata kõigepealt käsu enda abi: man, –help, whatis, apropos ja vajadusel info.

Põhikujud

- `man ls` — loe põhijuhendit
- `grep --help` — kiire lippude abi
- `whatis ls` — üks lause
- `apropos ssh` — otsi teema järgi
- `info coreutils 'ls invocation'` — GNU sügavam abi

Olulisemad lipud, märgid ja kiirnupud

- `q` — välju man-ist
- `/tekst` — otsi man-ist
- `n` — järgmine vaste
- `g` — algusesse
- `G` — lõppu

Pane tähele: Kui man tundub “terminali üle võtvat”, siis vajuta lihtsalt q; see on normaalne täisekraani vaade, mitte ummik.

Edasi: Järgmine loomulik samm: Kataloogid ja failid.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Kataloogid ja failid

Unix-laadses süsteemis on peaaegu kõik töö lõpuks failide ja kataloogidega töötamine. Selles peatükis teed esimesed päris muudatused, aga teed neid kontrollitult.

Loogika

Failidega töötamisel hoia rütm lihtne:

1. kontrolli, kus oled
2. vaata, mis seal juba on
3. tee üks väike muudatus
4. kontrolli tulemus üle
5. kustuta alles siis, kui oled kindel

See rütm on tähtsam kui üksik käsk. Enamik algaja vigu tuleb sellest, et muudatus tehakse vales kataloogis.

Kiire orientiir

Käsk või märk	Milleks	Mida tavaliselt näed
`pwd`	näita praegust kataloogi	üks täistee
`ls`	loetle kataloogi sisu	nimed või tühi väljund
`cd kataloog`	liigu kataloogi	edukal juhul sageli vaikne
`mkdir kataloog`	loo üks kataloog	edukal juhul sageli vaikne
`mkdir -p tee/kataloog`	loo vajadusel terve puuduv tee	edukal juhul sageli vaikne
`touch fail.txt`	loo tühi fail või uuenda ajatemplit	edukal juhul sageli vaikne
`cp a.txt b.txt`	kopeeri fail	edukal juhul sageli vaikne
`mv vana uus`	nimeta ümber või liiguta	edukal juhul sageli vaikne
`rm fail.txt`	kustuta fail ilma prügikastita	edukal juhul sageli vaikne
`rmdir kataloog`	kustuta tühi kataloog	edukal juhul sageli vaikne

Tühi väljund ei tähenda automaatselt viga. Paljud failikäsud ütlevad midagi ainult siis, kui midagi läks valesti.

Tüüpilised algaja vead

- `pwd` jääb vaatamata ja fail luuakse valesse kohta
- tühikutega nimi jäetakse jutumärkideta
- `mv` käsu puhul ei märgata, kas tegu on ümbernimetamise või liigutamisega
- arvatakse, et `rm` viib faili prügikasti
- kasutatakse `rm -rf`, sest see "töötab alati", mõistmata, kui palju see võib kustutada

Tee endale harjutuskataloog

Ära harjuta esimesi kustutamis- ja liigutamiskäske päris projektikataloogis. Tee eraldi harjutuskoht.

```
pwd
mkdir -p ~/tmp/faili-naited
cd ~/tmp/faili-naited
pwd
ls
```

Siin:

- `~` tähendab sinu kodukataloogi
- `mkdir -p ~/tmp/faili-naited` loob vajadusel ka vahekataloogi `tmp`
- kui `tmp` või `faili-naited` oli juba olemas, ei ole see viga
- viimane `pwd` kinnitab, et oled õiges kohas

`mkdir -p` on hea harjutuskäsk just sellepärast, et ta teeb puuduva tee valmis. Ilma `-p`-ta oskab `mkdir` luua ainult viimase kataloogi siis, kui vahepealne tee on juba olemas.

Teed: `.`, `..` ja `~`

Kolm märki tulevad failitöös pidevalt tagasi.

Kuju	Tähendus
`.`	praegune kataloog
`..`	ülemine kataloog
`~`	sinu kodukataloog

Näited:

```
pwd
cd ..
pwd
cd ~/tmp/faili-naited
pwd
```

Kui käsk kasutab kuju `./fail.txt`, tähendab see faili `fail.txt` siinsamas praeguses kataloogis. See on sama fail, mida tähistab ka lihtsalt `fail.txt`, aga `./` teeb asukoha nähtavamaks.

Failinimed ja tühikud

Shell jagab käsu tühikute kohalt osadeks. Kui faili või kataloogi nimes on tühik, pane nimi jutumärkidesse.

```
mkdir "Minu failid"
cd "Minu failid"
touch "esimene fail.txt"
ls
cd ..
```

Algajana on lihtsam kasutada failinimedes sidekriipsu või alakriipsu:

```
minu-failid
esimene_fail.txt
```

Tühikutega nimed on lubatud, aga käsureal nõuavad need rohkem tähelepanu.

Kataloogi sisu vaatamine

`ls` on kõige tavalisem esimene vaade.

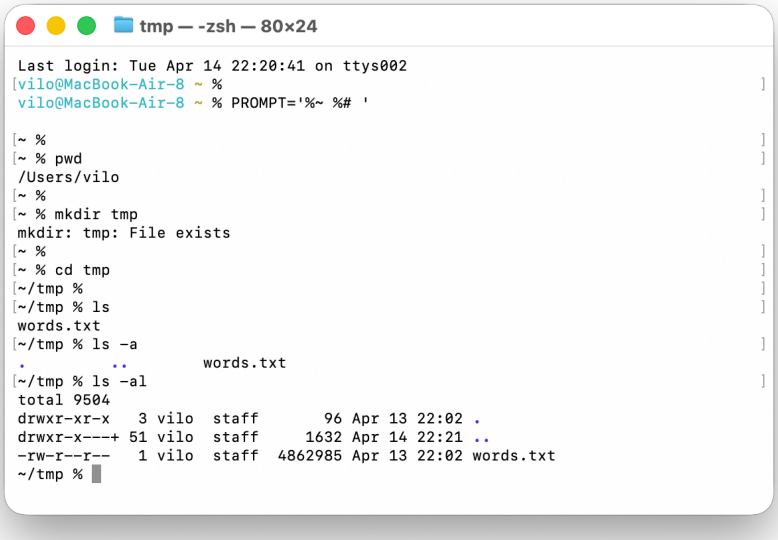
```
ls
ls -l
ls -a
ls -la
ls -A
```

Nende vahe:

- `ls` näitab tavalisi nimesid
- `ls -l` näitab detailvaadet
- `ls -a` näitab ka punktiga algavaid peidetud nimesid
- `ls -la` ühendab detailvaate ja peidetud nimed
- `ls -A` näitab peidetud nimesid, aga jätab `.` ja `..` välja

Selles aknas on oluline järjekord:

1. `pwd` näitab praegust asukohta
2. `mkdir tmp` proovib luua kataloogi, aga saab teate `File exists`
3. see ei ole ohtlik viga, vaid tähendab, et kataloog oli juba olemas
4. `cd tmp` liigub olemasolevasse kataloogi
5. `ls` näitab tavalist sisu
6. `ls -a` lisab `.` ja `..`
7. `ls -al` näitab sama pika detailvaatena



Joonis 4: Terminali näide, kus võrreldakse käske `ls`, `ls -a` ja `ls -al` ning nähakse ka olukorda, kus `mkdir tmp` annab teate `File exists`.

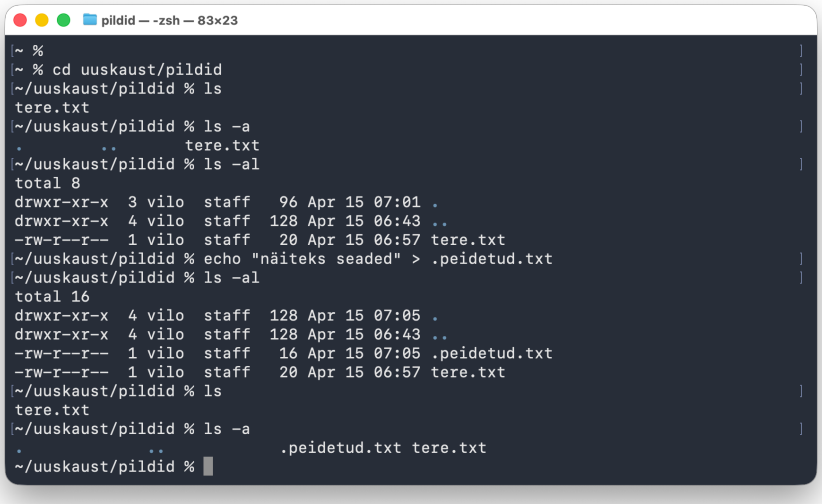
Punktiga algavad nimed

Punktiga algavad nimed on tavaliselt peidetud:

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

Need ei ole kadunud ega erilise kaitse all. See on kuvamiskokkulepe: paljud tööriistad jätavad punktiga algavad nimed vaikimisi näitamata.

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



Joonis 5: Terminali näide, kus kõigepealt võrreldakse käske `ls`, `ls -a` ja `ls -al`, seejärel luuakse peidetud fail `.peidetud.txt` ja vaadatakse sama kausta sisu uuesti.

Pildi põhisõnum on lihtne:

- punktiga algav fail on olemas
- tavaline `ls` ei pruugi seda näidata
- `ls -a` või `ls -A` teeb peidetud nimed nähtavaks

macOS Finderis saab peidetud failide kuvamist lülitada klahviga:

`Command-Shift-.`

Loo, kopeeri, nimeta ümber ja liiguta

Nüüd tee väike tervikharjutus samas harjutuskataloogis.

```
cd ~/tmp/faili-naited
pwd
touch esimene.txt
cp esimene.txt koopia.txt
mkdir arhiiv
mv koopia.txt arhiiv/
ls
ls arhiiv
```

Siin:

- `touch esimene.txt` loob tühja faili, kui seda veel ei ole
- `cp esimene.txt koopia.txt` teeb koopia
- `mkdir arhiiv` loob kataloogi
- `mv koopia.txt arhiiv/` liigutab faili kataloogi `arhiiv`

Kui käsk õnnestub, ei pruugi ta midagi öelda. Tulemuse kontrollimiseks kasuta `ls`.

`mv` teeb kaks eri asja

`mv` võib tähendada ümbernimetamist:

```
mv vana.txt uus.txt
```

Sama käsk võib tähendada liigutamist:

```
mv fail.txt arhiiv/
```

Eristus tuleb viimasest argumendist:

- kui lõpus on uus failinimi, nimetad ümber
- kui lõpus on kataloog, liigutad sinna sisse

Kui kahtled, tee enne:

```
ls -l
ls -l arhiiv
```

`touch` ja ajatempel

`touch` teeb kaks asja:

- kui faili ei ole, loob tühja faili
- kui fail on olemas, uuendab tavaliselt faili muutmisaega

Näide:

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

`cat tyhi.txt` ei näita midagi, sest fail on tühi. See on normaalne.

Kui tahad näha, millised failid on viimati muutunud, kasuta:

```
ls -lt
```

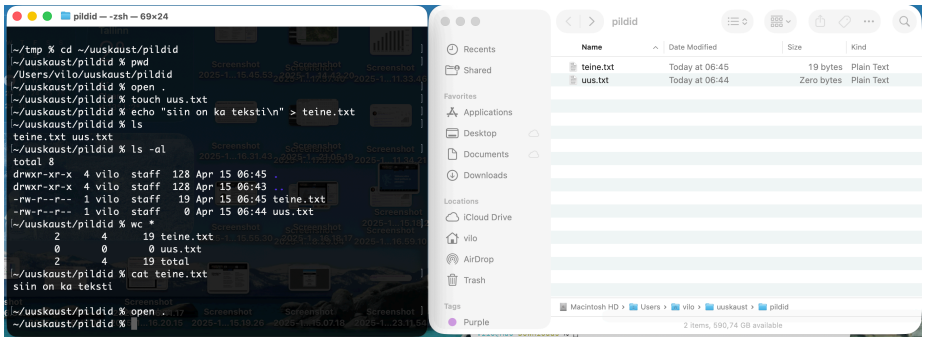
`ls -lt` sorteerib detailvaate aja järgi.

Terminal ja Finder näitavad sama kohta

Terminal ja graafiline failihaldur ei ole eri maailmad. Nad võivad näidata sama kataloogi.

macOS-is ava praegune kataloog Finderis nii:

```
open .
```



Joonis 6: 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.

Selles näites:

1. `cd ~/uuskaust/pildid` liigub kindlasse kataloogi
2. `pwd` kontrollib asukoha üle
3. `open .` avab sama kataloogi Finderis
4. `touch uus.txt` loob tühja faili
5. `echo "siin on ka teksti" > teine.txt` loob tekstiga faili
6. `ls` ja `ls -al` näitavad samu faile terminalis
7. Finder näitab sama tulemust graafiliselt

Õppetund: kui fail tekib Terminalis, näed seda tavaliselt ka Finderis, ja vastupidi.

Kustutamine: `rm`, `rmdir`, `rm -r`

Kustutamine on selle peatüki kõige riskantsem osa.

Enne kustutamist tee alati:

```
pwd
ls
```

Üksik fail:

```
rm fail.txt
```

Tühi kataloog:

```
rmdir tyhi-kataloog
```

Kataloog koos sisuga:

```
rm -r vana-kataloog
```

`rmdir` on algajale hea, sest ta töötab ainult tühja kataloogiga. Kui kataloogis on midagi sees, käsk peatub.

`rm -r` läheb kataloogi sisse ja kustutab rekursiivselt ka selle sisu. Seda kasuta ainult siis, kui oled teinud `pwd` ja `ls` kontrolli.

Miks `rm -rf` on eraldi ohtlik

Kuju:

```
rm -rf vana-kataloog
```

tähendab:

- `-r` kustutab rekursiivselt
- `-f` sunnib kustutamist ja vähendab küsimist

Käsureal ei ole vaikimisi Finderi või Windows Exploreri prügikasti. Kui `rm` õnnestub, on fail tavakasutuse mõttes kohe läinud.

Hea algaja rusikareegel:

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

Failide leidmine

`ls` näitab praegust kataloogi. `find` otsib kataloogipuu seest.

Ohutud otsingud:

```
find . -name '*.txt'
find . -type f
find . -type d
find . -maxdepth 2 -type f
```

Kui näed juhendis kuju:

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

siis ära käivita seda pimesi. Esmalt tee sama otsing ilma `-delete` osata:

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

Alles siis otsusta, kas kustutamine on tõesti õige. `find`-i ohutumad töövood tulevad hiljem eraldi peatükis.

Faili sisu kiire võrdlus räsi abil

Kui tahad kontrollida, kas kaks faili on sisult samad, saab kasutada räsi.

```
printf 'tere\n' > a.txt
cp a.txt b.txt
shasum -a 256 a.txt b.txt
```

Kui kahe faili `SHA-256` räsi on sama, on nende sisu praktilises mõttes sama.

Linuxis näed sageli ka käsku:

```
sha256sum a.txt b.txt
```

macOS-is on levinud:

```
shasum -a 256 a.txt b.txt
```

Selle teema põhjalikum koht on hiljem andmete tervikluse ja allalaadimiste juures. Siin piisab mõttest: sama sisu annab sama räsi.

Minitest

1. Loo harjutuskataloog `~/tmp/faili-naited`.
2. Mine selle kataloogi sisse ja kontrolli asukohta käsuga `pwd`.
3. Loo fail `readme.txt`.
4. Tee failist koopia nimega `readme.bak`.
5. Nimeta koopia ümber nimeks `vana.txt`.
6. Loo peidetud fail `.salajane`.
7. Võrdle käske `ls`, `ls -a` ja `ls -A`.
8. Kustuta üks tavaline fail alles pärast `pwd` ja `ls` kontrolli.
9. Leia `find` abil kõik `.txt` failid oma harjutuskataloogist.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Failitöö põhiharjumus on: kontrolli asukohta, tee üks väike muudatus, vaata tulemus üle ja kustuta alles pärast teadlikku pausi.

Põhikujud

- `pwd` — kontrolli asukohta
- `ls` — vaata sisu
- `mkdir -p ~/tmp/faili-naited` — loo terve puuduv tee
- `cd ~/tmp/faili-naited` — mine harjutama
- `touch fail.txt` — loo fail
- `cp algne.txt koopia.txt` — tee koopia
- `mv vana.txt uus.txt` — nimeta ümber
- `mv fail.txt kaust/` — liiguta kausta
- `rm fail.txt` — kustuta fail
- `rmdir tyhi-kataloog` — eemalda tühi kataloog
- `find . -name '*.txt'` — otsi ohutult
- `shasum -a 256 a.txt b.txt` — võrdle sisu macOS-is

Olulisemad lipud, märgid ja kiirnupud

- `.` — see kaust
- `..` — ülemine kaust
- `~` — kodukataloog
- `ls -a` — ka peidetud
- `ls -A` — peidetud ilma . ja .. kirjeteta
- `ls -la` — detailne vaade
- `"fail nimi.txt"` — nimi tühikuga
- `rm -r` — rekursiivne kustutus

Pane tähele: Enne rm või rm -r tee väike paus: kontrolli pwd, vaata ls, alles siis kustuta; rm ei vii faili prügikasti.

Edasi: Järgmine loomulik samm: Teksti vaatamine ja liikumine.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Teksti vaatamine ja liikumine

Loogika

Faili vaatamiseks vali käsk selle järgi, kui palju sisu tahad korraga näha:

- lühikese faili jaoks kasuta `cat`
- pika faili sirvimiseks ja otsimiseks kasuta `less`
- faili alguse vaatamiseks kasuta `head`

- faili lõpu või värskeimate logiridade vaatamiseks kasuta `tail`

`cat` trükib kogu faili kohe terminali. Kui fail on pikk, täidab see ekraani paljude ridadega ja varasem osa jääb terminali kerimisajalukku. Sellisel juhul on `less` parem, sest saad faili sees rahulikult liikuda, otsida ja `q`-ga väljuda.

Kiirülevaade

Eesmärk on valida vaatamiseks tööriist, mis ei uputa ekraani üle ega muuda faili.

Käsk	Milleks	Mida tavaliselt näed
`cat fail`	trüki kogu fail välja	pikk fail võib kiiresti mööda joosta
`less fail`	lehitse ja otsi mugavalt	eraldi lehitsetav vaade
`head fail`	vaata algust	lühike väljavõte algusest
`tail fail`	vaata lõppu	lühike väljavõte lõpust
`tail -f fail`	jälgi lõppu reaalajas	käsk jääb uusi ridu ootama

Tüüpilised algaja vead

- kasutatakse `cat`-i väga pika faili peal, kuigi `less` oleks parem
- unustatakse, et `less`-ist tuleb `q`-ga välja tulla
- aetakse segi “vaatan faili lõppu” ja “jälgin faili muutumist”

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 aitab siis, kui fail on pikk ja tahad kerimise asemel 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.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: lühikese faili jaoks kasuta cat; pika faili sirvimiseks ja otsimiseks kasuta less; faili alguse vaatamiseks kasuta head

Põhikujud

- `less fail.txt` — ava lehitseja
- `head -n 7 fail.txt` — esimesed read
- `tail -n 7 fail.txt` — viimased read
- `tail -f app.log` — jälgi logi
- `less *` — mitu faili korraga
- `cat` — näita faili
- `seq` — numbrijada
- `touch` — loo või aja tempel

`less`-i klahvid

- `q` — välju
- `/tekst` — otsi edasi
- `n` — järgmine vaste
- `N` — eelmine vaste
- `v` — ava redaktoris
- `:n / :p` — järgmine/eelmine fail

Pane tähele: cat sobib lühikese faili jaoks; pika faili puhul vali less, head või tail.

Edasi: Järgmine loomulik samm: Failide muutmine: nano ja esimene kokkupuude vim-iga.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Failide muutmine: nano ja esimene kokkupuude vim-iga

Loogika

Need tööriistad on seotud nii:

- `less` aitab faili sisu enne muutmist mugavalt üle vaadata
- `nano` on alguses kõige lihtsam tekstiredaktor
- `vim` ja `vi` on sama pere klassikalised tekstiredaktorid

Praktiliselt tähendab see, et alguses piisab täiesti sellest:

- vaata faili `less` abil
- tee lihtne muudatus `nano` abil
- õpi `vim`-ist või `vi`-st vähemalt kindlalt väljuma

Kiirülevaade

Eesmärk on liikuda vaatamiselt muutmisele nii, et tead, millal fail päriselt muutub.

Käsk või tööriist	Milleks	Mida tavaliselt näed
`less fail`	vaata faili mugavalt	lehitsetav vaade, fail ei muutu
`nano fail`	muuda faili lihtsalt	redaktor allserva kiirnuppudega
`vim fail`, `vi fail`	klassikaline redaktor oskajale	modaalne vaade; väljumiseks on vaja käske
salvestamine	kirjuta muudatus faili	alles siis muutub faili sisu

Tüüpilised algaja vead

- aetakse segi faili vaatamine ja faili muutmine
- avatakse `vim` kogemata ja ei teata, kuidas välja saada
- unustatakse, et redaktor muudab päris faili, mitte ainult ekraanivaadet

Kiirspikker

- `less fail.txt` vaata faili sisu mugavalt
- `nano fail.txt` lihtne tekstiredaktor
- `vim fail.txt` klassikaline redaktor
- `vi fail.txt` sama pere variant
- `Ctrl-O` salvesta `nano`-s
- `Ctrl-X` välju `nano`-st
- `Ctrl-W` otsi `nano`-s

Kõige sagedasemad `vim` käsud alguses:

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

`nano` kõige vajalikumad klahvid

Alguses piisab täiesti neist kolmest:

- `Ctrl-O` salvestab faili
- `Ctrl-X` väljub redaktorist
- `Ctrl-W` otsib tekstist

Praktiline rada on tavaliselt selline:

1. ava fail `nano` abil
2. tee muudatus
3. vajuta `Ctrl-O`, siis Enter

4. vajuta `Ctrl-X`

Hea on meeles pidada, et `nano` näitab allservas ise peamisi kiirnuppe. Kui unustad klahvi, vaata ekraani allaossa.

Käivita need käsud

```
printf 'esimene\nteine\nkolmas\n' > 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
```

Kui `less`-i otsimine või liikumine on meelest läinud, vaata tagasi peatükki Teksti vaatamine ja liikumine.

Kui `nano` puudub

Mõnes süsteemis ei ole `nano` vaikimisi paigaldatud.

Siis on sul kolm lihtsat varianti:

- proovi, kas `nano` on siiski olemas käsuga `command -v nano`
- kasuta ajutiselt `vi` või `vim`
- paigalda `nano` hiljem siis, kui paketihaldus on juba tuttav

Alguses on kõige tähtsam teada, kuidas tundmatust redaktorist turvaliselt väljuda.

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.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: less aitab faili sisu enne muutmist mugavalt üle vaadata; nano on alguses kõige lihtsam tekstiredaktor; vim ja vi on sama pere klassikalised tekstiredaktorid

Põhikujud

- `less fail.txt` — vaata faili sisu mugavalt
- `nano fail.txt` — lihtne tekstiredaktor
- `vim fail.txt` — klassikaline redaktor
- `vi fail.txt` — sama pere variant
- `command -v nano` — kontrolli olemasolu

`vi/vim` minispikker

- `Esc` — välju sisestusest
- `:q` — välju
- `:q!` — välju salvestamata
- `:wq` — salvesta ja välju

Pane tähele: Kui nano puudub või vim hirmutab, siis peamine algaja oskus on see, et sa oskad redaktorist kindlalt väljuda.

Edasi: Järgmine loomulik samm: Käskude kuju ja argumentide loogika.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Käskude kuju ja argumentide loogika

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

Alusta ühest näitest

Võta kõigepealt üks lihtne näide:

```
ls -la /etc
```

Siin on kolm eri rolli:

- `ls` on käsk

- `-la` on lipud ehk valikud
- `/etc` on argument ehk koht, mille peal käsk töötab

Sama loogika kordub väga paljudes teistes käskudes.

Näiteks:

```
cp vana.txt uus.txt
```

Siin:

- `cp` on käsk
- eraldi lippu ei ole
- `vana.txt` ja `uus.txt` on argumendid

Põhireegel

Enamasti saab käsurea lugeda 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 -la /etc
grep -n root fail.txt
cp fail.txt koopia.txt
```

Tüüpilised algaja vead

- aetakse segi, mis on valik ja mis on argument
- arvatakse, et kõik lipud töötavad igal käsul ühtmoodi
- unustatakse jutumärgid siis, kui nimes on tühikud

Kui saad aru valikutest, argumentidest ja jutumärkidest, tuleb üllatusi vähem.

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`

Need on siin näidetes rajad ja failinimed, millele käsk rakendub. Argument ei ole alati failinimi: see võib olla ka kataloogitee, muster, URL või muu väärtus, mida käsk vajab.

Rusikareegel jääb samaks:

```
käsk [valikud] [argumendid]
```

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 *.txt
ls words.0?.txt
ls pilt[12].png
```

Need tähendavad:

- `*.txt` valib kõik `.txt` lõpuga failid
- `words.0?.txt` valib failid nagu `words.00.txt`, `words.01.txt`, `words.02.txt`
- `[]` sobitab ühe märgi etteantud hulgast

Üldreegel nende märkide taga on:

- `*` sobitab null või rohkem märki
- `?` sobitab täpselt ühe märgi

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 täht-tähelt nii, nagu see on kirjutatud
- `"..."` 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 teksti võtta täht-tähelt nii, nagu see on kirjutatud
- 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 `--`.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Õpi ühe näite pealt eristama käsku, lippu ja argumenti; siis ei tundu uus käsk iga kord täiesti uus keel.

Üks näide

- `ls -la /etc` — terve kuju
- `ls` — käsk
- `-la` — lipud
- `/etc` — argument
- `cp vana.txt uus.txt` — kaks argumenti

Tüüpilised kujud

- `grep -n 'root' fail.txt` — lipp + muster + fail
- `cp 'fail nimi.txt' koopia.txt` — nimi tühikuga
- `rm -- --algab-kriipsuga.txt` — ära tõlgenda lipuna
- `ls *.txt` — kõik .txt lõpuga failid
- `ls words.0?.txt` — üks märk keskel

Olulised märgid

- `-` — lühike lipp
- `--` — pikk lipp või stop
- `"..."` — hoia koos; muutujad toimivad
- `'...'` — täht-tähelt nagu kirjas
- `*` — failimuster
- `?` — üks märk

Pane tähele: Kui nimes on tühikuid või erimärke, pane nimi jutumärkidesse; see on üks kõige tavalisemaid algaja komistuskohti.

Edasi: Järgmine loomulik samm: Sisend, väljund, torud ja suunamine.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Sisend, väljund, torud ja suunamine

Loogika

Kõige tähtsam mõte:

- käsk võib kirjutada tavalist väljundit
- käsk võib anda ka eraldi veateateid
- tavalise väljundi võib suunata faili
- tavalise väljundi võib anda teisele käsule
- käske saab ka lihtsalt järjest või tingimusega käivitada

Need ei ole sama asi. Faili suunamine, toru ja tingimuslik käsujada lahendavad eri probleeme.

Kiirülevaade

Eesmärk on panna käsud koostööd tegema: salvesta väljund, ühenda käsud või käivita järgmine samm tingimusega.

Märk või käsk	Milleks	Mida tavaliselt näed
`>`	kirjuta stdout faili üle	ekraanil vähem, failis uus sisu

Märk või käsk	Milleks	Mida tavaliselt näed
`>>`	lisa stdout faili lõppu	fail kasvab, vana sisu jääb alles
`|`	anna stdout järgmise käsu sisendiks	näed tavaliselt viimase käsu tulemust
`;`	käivita käsud järjest	järgmine käsk jookseb igal juhul
`&&`	jätka ainult edu korral	järgmine käsk jookseb ainult `0` koodi järel
`||`	käivita varuplaan vea korral	järgmine käsk jookseb vea järel
`2>`	suuna veateated faili	stderr läheb eraldi faili
`tee`	näita ja salvesta korraga	väljund on ekraanil ja failis

Lõpetuskood ei paista otse välja; vajadusel küsi seda käsuga `echo $?`.

Tüüpilised algaja vead

- aetakse segi `;` ja `|`
- kirjutatakse fail kogemata `>` abil üle, kuigi taheti lisada `>>`
- eeldatakse, et `sudo echo ... > fail` annab suunamisele samuti root-õiguse
- imestatakse, miks edukas käsk vaikis, kuigi väljund läks tegelikult faili või torusse

1. Tavaline väljund, veaväljund ja lõpetuskood

Käsureal on hea eristada kolme mõistet:

- `stdout`: tavaline väljund, mida käsk kirjutab
- `stderr`: eraldi veaväljund vigade ja hoiatuste jaoks
- lõpetuskood ehk `exit code`: märge selle kohta, kas käsk lõppes edukalt

Praktiline rusikareegel on:

- `0` tähendab tavaliselt edu
- mõni muu arv tähendab tavaliselt, et midagi läks valesti

Kui tahad näha viimase käsu lõpetuskoodi, saad kasutada:

```
ls sonad.txt
echo $?
ls puuduv_fail
echo $?
```

See mõte muutub oluliseks hiljem siis, kui jõuame märkideni `&&` ja `||`.

2. > 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 vana sisu üle.

3. >> 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
```

Peamine vahe on lihtne: `>` kirjutab üle, `>>` lisab lõppu.

4. 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 `>` abil faili kirjutamine.

5. 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. Vahe on lihtne: `;` käivitab käsud järjest, `|` ühendab käsud voona.

6. `&&` ja `||`

Need kaks märki lisavad käsujadale tingimuse. Nad ei vaata faili sisu ega toru kaudu tulevaid ridu, vaid eelmise käsu lõpetuskoodi.

`&&`

```
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

7. 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

Kui tahad veateate lihtsalt ära peita, siis võid kirjutada:

```
ls puuduv_fail 2> /dev/null
```

`/dev/null` on erifail, kuhu saadetud väljund kaob ära. Seda võib mõelda kui musta auku väljundile. Aprillinaljana öeldakse vahel, et `/dev/null` sai täis, aga päriselt tähendab see lihtsalt kohta, kuhu saab midagi teadlikult "mitte kuhugi" saata.

8. `tee`

`tee` on kasulik siis, kui tahad väljundit korraga ekraanil näha ja faili salvestada.

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
```

9. `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 esimene kuju, mida pähe õppida, aga hea on teada, et selline võimalus on olemas.

10. 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

11. 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`.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: Ühenda käske nii, et ühe väljund saab järgmise sisendiks, või saada väljund faili; vea- ja tavaväljundit tasub mõelda eraldi.

Põhikäsud

- `echo` — prindi tekst
- `printf` — vorminda tekst
- `cat` — näita faili
- `tee` — ekraan ja fail
- `wc` — loe kokku

Tüüpilised kujud

- `käsk > valjund.txt` — stdout faili
- `käsk >> logi.txt` — lisa stdout faili lõppu juurde
- `käsk 2> vead.txt` — stderr eraldi
- `käsk > valjund.txt 2> vead.txt` — eralda mõlemad
- `käsk 2> /dev/null` — peida veateated
- `cat fail.txt | wc -l` — toru loendusse
- `käsk ; järgmine` — järgmine alati
- `käsk && järgmine` — järgmine ainult edul
- `käsk || varuplaan` — varuplaan vea korral
- `käsk > koik.txt 2>&1` — stdout ja stderr koos

Olulisemad lipud, märgid ja kiirnupud

- `>` — stdout faili
- `>>` — lisa stdout faili lõppu juurde
- `|` — anna edasi järgmisele
- `;` — käivita lihtsalt järjest
- `&&` — tee edasi edu korral
- `||` — tee edasi vea korral
- `2>` — stderr eraldi faili
- `2>&1` — stderr samasse kohta
- `/dev/null` — must auk väljundile

Pane tähele: Õpi kõigepealt rahulikult >, » ja |; alles siis lisa sellele &&, || ja 2>&1.

Edasi: Järgmine loomulik samm: Esimene tervikharjutus: 30 minutit.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

Esimene tervikharjutus: 30 minutit

See peatükk ei ole enam täiesti esimene kokkupuude käsureaga. Kui terminali põhimõtted, failid, abi leidmine ja suunamine on juba tuttavad, saad siin teha ühe lühikese tervikharjutuse algusest lõpuni läbi.

Loogika

See harjutus seob 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 ilma uusi sümboleid või käske juurde toomata.

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

See on käsureal väga tavaline tööviis: 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. Linux, Unix, GNU, macOS, Windows ja shellid
2. Failisüsteemi kaart
3. Kettaruum ja süsteemi maht

4. Õigused, omanikud ja täitmisbitid

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 `>>`.

Peatüki täisspikker

Tase: **Algaja**

Eesmärk: kontrolli alguskohta; tööta eraldi harjutuskaustas; lisa faili read väikeste sammudena

Lühike töövoog

- `pwd && ls` — kontrolli alguskohta
- `mkdir proov && cd proov` — loo töökaust
- `echo tere > sonad.txt` — tee esimene rida
- `echo maailm >> sonad.txt` — lisa järgmine rida
- `cat sonad.txt && wc -l sonad.txt` — kontrolli tulemust
- `cp sonad.txt koopia.txt` — tee koopia

Olulised märgid

- `>` — kirjuta üle
- `>>` — lisa faili lõppu juurde
- `Ctrl-c` — katkesta programmi töö
- `man` — vaata abi

Pane tähele: Kui mõni käsk on vahepeal hägusaks läinud, peatu ja vaata man või eelmised Osa I peatükid üle.

Edasi: Järgmine loomulik samm: Linux, Unix, GNU, macOS, Windows ja shellid.

Osa PDF: ./spikrid/osa-i-esimesed-sammud-spikker.pdf

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.

Kiirülevaade

Eesmärk on saada mõisteline kaart: mis tuleb süsteemist, mis shellist ja mis konkreetsetest käsureatööriistadest.

Mõiste või käsk	Milleks	Mida tavaliselt näed
Linux	tuum, mitte kogu süsteem	süsteemiinfo `uname` väljundis
GNU	paljud tuttavad käsureatööriistad	käsud ja nende GNU-spetsiifilised lipud

Mõiste või käsk	Milleks	Mida tavaliselt näed
shell	loeb ja tõlgendab käsurida	viip, muutujad ja shelli sisekäsud
macOS, Windows	sarnane pind, eri tööriistad	sama käsk võib käituda erinevalt
`uname -a`	kontrolli süsteemi ja kernelit	üks pikk süsteemirida
`echo "$SHELL"`	näita vaikimisi shelli	tee nagu `/bin/zsh`
`ps -p $$`	näita käimasolevat shelli	shelliprotsessi rida

Tüüpilised algaja vead

- arvatakse, et Linux, Unix ja GNU tähendavad sama asja
- aetakse segi terminal, shell ja operatsioonisüsteem
- eeldatakse, et macOS-i või Windowsi käsk käitub täpselt nagu Linuxis

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

Mõni Linuxi käsk puudub macOS-is ka täiesti sama nime all. Näiteks:

- Linuxis kasutatakse tihti `ip a`
- macOS-is on tavalisem `ifconfig`
- Linuxis kasutatakse tihti `ss -ltn`
- macOS-is on tavalisem `lsof -iTCP -sTCP:LISTEN -n -P` või `netstat -anv -p tcp | grep LISTEN`

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

Kui tahad mõnda Linuxi tööriista ka nime poolest lähemale tuua, siis tee seda vajaduspõhiselt, mitte "igaks juhuks". Näiteks võrgukäskude puhul võib hiljem abiks olla `brew install iproute2mac`, aga alguses ei ole see vajalik.

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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine ning shelli enda kohta jaotisest Shellid ja põhikäsud.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: erista süsteemi, shelli ja käsureatööriistu; sama terminaliaken ei tähenda veel, et Linux, macOS ja Windows käituksid ühtemoodi

Kontrollkäsud

- `uname -a` — süsteemi ja kerneli rida
- `echo "$SHELL"` — praegune shell
- `ps -p $$` — käivitud shelliprotsess

- `command -v ls` — kust käsk leitakse
- `sw_vers` — macOS-i versioon
- `wsl -l -v` — WSL-i jaotused

Olulised mõisted

- `Linux` — kernel ehk tuum
- `GNU` — paljud põhikäsud
- `shell` — tõlgendab käske
- `WSL` — Linux Windowsis
- `BSD` — macOS-i tööriistapere
- `PowerShell` — Windowsi käsukeskkond

Pane tähele: macOS-is ja Windowsis võivad sama nimega käsud käituda teistmoodi kui Linuxis.

Edasi: Järgmine loomulik samm: Failisüsteemi kaart.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Failisüsteemi kaart

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.

Kiirülevaade

Eesmärk on saada süsteemist kaart, et teaksid, kus on ohutu töötada ja millised kohad on süsteemi omad.

Tee või käsk	Milleks	Mida tavaliselt näed
`/`	kogu failipuu juur	süsteemi ülemine tase
`~`	sinu kodukataloog	sinu failide töökoht
`/bin`, `/usr/bin`	käivitatavad programmid	käsud nagu `ls`, `cat`, `grep`
`/etc`	seadistused	palju süsteemifaile
`/var`	muutuv sisu, näiteks logid	logid, vaheandmed ja teenuste failid
`/tmp`	ajutised failid	ajutine tööala

Tee või käsk	Milleks	Mida tavaliselt näed
`pwd`	näita praegust asukohta	üks täistee
`echo "$HOME"`	näita kodukataloogi	tee sinu kodukataloogi

Tüüpilised algaja vead

- arvatakse, et `~` ja `/` on lihtsalt kaks erinevat märki sama asja jaoks
- minnakse süsteemikaustadesse midagi muutma enne, kui saadakse aru, mis seal elab
- eeldatakse, et Linux ja macOS kasutavad täpselt samu teid

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
- `/bin` ja `/usr/bin` sisaldavad käivitatavaid programme
- `/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 juurkataloogi, mitte oma isiklikku töökataloogi.

Oluline vahe:

- `cd /` viib juurkataloogi
- `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/kasutaja`

macOS-is sageli midagi sellist:

`/Users/kasutaja`

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`

Nimi `bin` tuleb ajalooliselt sõnast `binary`. Unix-laadsetes süsteemides tähendab see tavaliselt kataloogi, kus on käivitatavad programmid ehk käsud. Näiteks:

- `/bin` sisaldab süsteemi põhilisi käske
- `/usr/bin` sisaldab suurt hulka tavakasutaja käske
- `~/bin` on levinud koht kasutaja enda väikeste skriptide jaoks

Kõik `bin`-kataloogis olev ei pea tänapäeval olema masinkoodis binaar. Seal võib olla ka shelliskripte või Pythoni programme, kui neid saab käsuna käivitada.

`/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.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: õpi eristama süsteemi juuri, oma kodukataloogi ja ajutisi ning süsteemseid kaustu; siis on lihtsam aru saada, kus tohid rahulikult katsetada

Põhikujud

- `pwd` — kontrolli asukohta
- `echo "$HOME"` — vaata oma kodu
- `cd /` — mine juurkataloogi
- `cd ~` — mine kodukataloogi
- `ls /etc | head` — piilu seadistusi
- `ls -ld /tmp` — vaata ajutist kausta

Olulised teed

- `/` — kogu failipuu juur
- `~` — sinu kodukataloog
- `/etc` — süsteemi seadistused
- `/var` — logid ja muutuv sisu
- `/tmp` — ajutiste failide koht
- `/Users / /home` — kasutajate kodud

Pane tähele: / ja ~ ei ole sama asi: esimene on süsteemi juur, teine sinu enda tööala.

Edasi: Järgmine loomulik samm: Kettaruum ja süsteemi maht.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Kettaruum ja süsteemi maht

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.

Kiirülevaade

Erista kaht küsimust: kas failisüsteemis on ruumi ja mis konkreetse kausta sees ruumi võtab.

Käsk või töövoog	Milleks	Mida tavaliselt näed
`df -h`	vaata failisüsteemide mahtu	tabel kogu-, kasutatud ja vaba ruumiga
`du -sh .`	mõõda praegust kausta kokku	üks kokkuvõtlik rida
`du -a .`	mõõda kõik kirjed	pikk nimekiri failidest ja kaustadest
`sort`	tõsta suuremad ette	samad read uues järjekorras
`less`	lehitse pikka nimekirja	rahulik lehitsetav vaade

Tüüpilised algaja vead

- arvatakse, et `df` ja `du` annavad sama vastuse
- vaadatakse ainult kogumahtu, mitte seda, milline kaust ruumi võtab
- unustatakse peidetud kaustad nagu `.git`, `.venv` või `.cache`

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 inimloetavad mahud 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 detailsemalt näha, kuhu ruum kaob:

```
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, millisel failisüsteemil praegune töökaust asub:

```
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
```

Siin tähendab `sort -h`, et sorditakse inimloetavaid suurusi nagu `12K`, `450M` ja `3.1G`, mitte ainult paljaid numbreid.

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`

Peidetud kaustade kaasamiseks kasuta:

```
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.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: erista kogu failisüsteemi seisu ja ühe konkreetse kausta mahtu; df ja du vastavad eri küsimustele

Põhikujud

- `df -h` — kõigi failisüsteemide maht
- `df -h .` — siinse failisüsteemi maht
- `du -sh .` — praeguse kausta summa
- `du -sh * 2> /dev/null` — alamkirjed ilma vigadeta
- `du -sh * 2> /dev/null | sort -h` — alamkirjed inimloetava mahu järgi
- `du -a . 2> /dev/null | sort -nr | less` — suurimad kirjed detailsemalt

Olulised võtmed ja vood

- `-h` — inimloetavad ühikud
- `-s` — ainult kokkuvõte
- `-a` — näita ka üksikuid kirjeid
- `2> /dev/null` — peida ligipääsuvead
- `| sort -h` — sordi inimloetavad mahud väiksemast suuremani
- `| sort -nr | less` — suurimad ette

Pane tähele: df ei ütle, milline kaust ruumi sööb; selleks vaata du väljundit.

Edasi: Järgmine loomulik samm: Õigused, omanikud ja täitmisbitid.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Õigused, omanikud ja täitmisbitid

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.

Kiirülevaade

Eesmärk on lugeda õiguste rida, teha fail vajadusel käivitatavaks ja mõista "Permission denied" viga.

Käsk või mõiste	Milleks	Mida tavaliselt näed
`ls -l`	näita õigusi, omanikku ja gruppi	rida nagu `-rw-r--r--`
`chmod`	muuda õigusi	edukal juhul sageli vaikne
`chown`	muuda omanikku või gruppi	edukal juhul sageli vaikne
täitmisbitt `x`	luba faili käivitada	faili saab programmina käivitada
kataloogi `x`	luba kataloogi "siseneda"	nimed ja teed muutuvad kasutatavaks

`drwxr-xr-x` algab `d` tähega, sest tegu on kataloogiga.

Tüüpilised algaja vead

- arvatakse, et `chmod +x` teeb failist automaatselt "programmi"
- muudetakse õigused liiga laiaks, ilma et oleks aru saadud, kellele mida antakse
- aetakse segi faili lugemisõigus ja käivitamisõigus

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

Lühidalt:

- `r` tähendab lugemist
- `w` tähendab kirjutamist
- `x` tähendab faili puhul käivitamist, kataloogi puhul sisenemist

Miks kataloogi x-õigus on eriline

Faili puhul tähendab `x`, et faili võib programmina käivitada.

Kataloogi puhul tähendab `x` midagi natuke muud:

- tohid sinna `cd` abil siseneda
- tohid kasutada selle sees olevaid nimesid
- ilma `x`-õiguseta võib kataloog olemas olla, aga sa ei saa selles normaalselt liikuda

See on üks tavaline koht, kus `permission denied` tundub esmapilgul segane.

Õiguste muutmine

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

Siin on kaks levinud stiili:

- sümboolne kuju nagu `chmod u+x fail`
- numbriline kuju nagu `chmod 644 fail`

Sümboolne kuju sobib hästi siis, kui tahad teha ühe väikese muudatuse:

- `u+x fail` lisa omanikule täitmisõigus
- `g-w fail` võta grupilt kirjutamisõigus ära

Numbriline kuju sobib siis, kui tahad seada terve õiguste rea korraga:

- `644` tähendab tavaliselt tekstifaili
- `755` tähendab tavaliselt käivitatavat faili või kataloogi

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`.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: loe õiguste rida, mõista rwx tähendust ja erista faili käivitatavust kataloogi läbikäigust

Põhikujud

- `ls -l fail.txt` — loe õiguste rida
- `chmod u+x skript.sh` — anna omanikule täitmine
- `chmod 644 naide.txt` — sea tekstifaili õigused
- `chmod 755 skript.sh` — sea käivitatava õigused
- `sudo chown kasutaja:grupp fail.txt` — muuda omanikku

Olulised õigusekujud

- `r` — loe sisu
- `w` — muuda sisu
- `x` — käivita või sisene
- `d` — tegu on kataloogiga
- `644` — tüüpiline tekstifail
- `755` — tüüpiline käivitatav

Pane tähele: Kataloogi x-õigus tähendab läbikäiku; see ei ole sama asi mis faili käivitamine.

Edasi: Järgmine loomulik samm: Kasutajad, grupid ja sudo.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Kasutajad, grupid ja sudo

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.

Kiirülevaade

Eesmärk on teha nähtavaks, kes käsu käivitab ja millal kasutatakse kõrgemaid õigusi.

Käsk	Milleks	Mida tavaliselt näed
`whoami`	näita praegust kasutajat	üks kasutajanimi
`id`	näita kasutaja ja gruppide identiteeti	pikem rida UID/GID ja gruppidega
`groups`	näita gruppe lühemalt	grupinimede loend
`sudo käsk`	käivita üks käsk kõrgemate õigustega	paroolikküsimus või veateade
`sudo -l`	kontrolli lubatud `sudo` käske	lubade loend või keeld

Tüüpilised algaja vead

- arvatakse, et `sudo` tähendab "paranda see käsk võluväel ära"
- jäädakse pikaks ajaks `root`-i või kõrgemate õigustega shelli
- käivitatakse ohtlik käsk `sudo`-ga enne, kui kontrollitakse, kus parajasti ollakse

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:

```
kasutaja@mac proov % whoami
kasutaja
```

```
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`.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: õpi nägema, millise kasutajana sa töötad, millistes gruppides oled ja millal on sudo päriselt vajalik

Põhikujud

- `whoami` — kontrolli kasutajat
- `id` — loe grupid välja
- `groups` — vaata grupinimesid
- `sudo -l` — vaata sudo õigusi
- `sudo apt update` — üks kõrgem käsk
- `su - kasutaja` — vaheta kasutajat

Olulised märgid ja rollid

- `$` — tavaline kasutaja
- `%` — teine levinud tavaviip
- `#` — root või kõrgemad õigused
- `root` — administraatori kasutaja
- `sudo -i` — ava kõrgem shell
- `sudo käsk` — eelista ühte käsku

Pane tähele: Ära kasuta sudo-t lihtsalt harjumusest; enne kontrolli, kas käsk tõesti vajab kõrgemaid õigusi.

Edasi: Järgmine loomulik samm: Muutujad, keskkond, PATH ja aliased.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Muutujad, keskkond, PATH ja aliased

Shell ei käivita ainult programme. Ta hoiab ka väärtusi, otsustab, kust käske otsida, ja võib anda käskudele lühinimesid.

Loogika

Selles peatükis on neli põhimõtet:

1. shellimuutuja on väärtus praeguses shellis
2. keskkonnamuutuja antakse edasi käivitatud programmidele
3. `PATH` on kataloogide järjekord, kust väliseid käske otsitakse
4. alias on lühinimi käsule või käsukujule

Kõige tähtsam on mitte ajada segi väärtust, käsu otsimist ja mugavusnime.

Kiire orientiir

Kuju	Milleks	Mida tavaliselt näed
`nimi='Mari'`	loo shellimuutuja	edukal juhul vaikne
`echo "$nimi"`	näita muutuja sisu	väärtus terminalis
`export NIMI='väärtus'`	anna väärtus alamprotsessidele	edukal juhul vaikne
`env`	vaata keskkonnamuutujaid	ridade loend
`echo "$PATH"`	vaata käsuotsingu teid	koolonitega eraldatud teed
`command -v python3`	vaata, mis käsk leitakse	failitee või tühi väljund
`type cd`	vaata käsu liiki	builtin, alias, failitee või muu
`type -a grep`	näita kõik sama nimega vasted	mitu võimalikku vastet
`alias ll='ls -lah'`	loo lühinimi	edukal juhul vaikne
`source ~/.zshrc`	loe seadistusfail uuesti sisse	edukal juhul vaikne

Tüüpilised algaja vead

- pannakse `nimi=väärtus` juurde tühikud, näiteks `nimi = Mari`
- arvatakse, et shellimuutuja on automaatselt kõigile programmidele nähtav
- muudetakse `PATH`-i nii, et kogemata varjutatakse süsteemi käsk
- usaldatakse ainult `which` käsku ja ei märgata alias't või shelli sisseehitatud käsku
- lisatakse alias valesse seadistusfaili ja imestatakse, miks uus terminaliaken seda ei tunne
- kopeeritakse prompti muutmise näide püsivasse faili enne, kui seda ajutiselt prooviti

Shellimuutuja ja keskkonnamuutuja

Shellimuutuja elab praeguses shellis.

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

Kui käivitad uue programmi, ei pruugi tavaline shellimuutuja sinna kaasa minna.

```
nimi='Mari'
sh -c 'echo "$nimi"'
```

Keskkonnamuutuja tehakse alamprotsessidele nähtavaks käsuga `export`.

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

Põhireegel:

- `nimi='Mari'` sobib väärtuseks praeguses shellis
- `export NIMI='Mari'` sobib siis, kui käivitatav programm peab seda nägema

Hea tava on kirjutada keskkonnamuutujad sageli suurte tähtedega, näiteks `PATH`, `HOME`, `EDITOR`, `SHELL`.

Tühikud ja jutumärgid

Muutuja omistamisel ei panda võrdusmärgi ümber tühikuid.

Õige:

```
nimi='Mari'
```

Vale:

```
nimi = 'Mari'
```

Kui väärtuses on tühik, pane väärtus jutumärkidesse.

```
projekt='Linuxi õpik'
echo "$projekt"
```

Muutuja lugemisel eelista kuju:

```
echo "$projekt"
```

Jutumärgid hoiavad väärtuse koos ka siis, kui sees on tühikuid.

Mis asi käsk on

Terminali kirjutatud nimi ei tähenda alati välist programmi failisüsteemis.

See võib olla:

Liik	Näide	Märkus
alias	`ll`	shell asendab lühinime käsuga
shelli funktsioon	`mycmd`	kasutaja või seadistusfaili määratud funktsioon
builtin	`cd`	shelli sisseehitatud käsk
reserveeritud sõna	`if`, `for`	shellikeele osa
väline programm	`grep`, `python3`	fail kuskil `PATH` kataloogis

Seetõttu on õppimisel väga kasulik:

```
type cd
type grep
type ll
command -v python3
```

`type` ütleb käsu liigi. `command -v` ütleb lühidalt, mida shell käivitaks.

Kui sama nimega vasteid on mitu:

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

See aitab näha, kas eespool on näiteks Homebrew, Conda, süsteemi Python või sinu enda skript.

`PATH`: kust käsk leitakse

Kui kirjutad käsu:

`python3`

siis shell otsib väliseid programme muutujas `PATH` loetletud kataloogidest.

```
echo "$PATH"
```

Tüüpiline kuju on selline:

`/Users/kasutaja/bin:/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin`

Oluline reegel:

- teed on eraldatud kooloniga
- otsitakse vasakult paremale
- esimene sobiv käivitatav fail võidab

Kontrolli alati päris vastet:

```
command -v python3
command -v grep
```

Kui käsk ei leidu, on tavaliselt üks kahest põhjusest:

- programm pole paigaldatud
- programmi kataloog pole `PATH`-is

Oma `bin` kataloog

Oma väikesed skriptid võib panna näiteks kataloogi `~/bin` või `~/.local/bin`.

Loo kataloog:

```
mkdir -p "$HOME/bin"
```

Lisa see praeguses shellis otsingutee ette:

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

Ette lisamine tähendab, et sinu `~/bin` vaste leitakse enne süsteemi sama nimega käsku. See on mugav, aga nõuab tähelepanu.

Ettevaatlikum kuju on lisada oma kataloog lõppu:

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

Sel juhul jäävad süsteemi tavalised käsud eelisjärjekorda.

Väike oma käsk

Tee üks väike käsurea tööriist oma `~/bin` kataloogi.

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

Siin:

- fail `opik-tere` on väike shelliskript
- `chmod +x` teeb selle käivitatavaks
- `PATH` lisamine lubab käivitada seda nimega `opik-tere`
- `command -v opik-tere` kinnitab, kust käsk leitakse

Kui see töötab ainult praeguses aknas, on põhjus lihtne: `export PATH=...` oli ajutine. Püsivaks muutmiseks lisa rida shelli seadistusfaili.

Seadistusfailid

Shell loeb käivitumisel seadistusfaile. Täpne fail sõltub shellist ja sellest, kas tegu on login-shelli või interaktiivse shelliga.

Praktiline algaja jaotus:

Shell	Kuhu panna alias'ed ja mugavused	Kuhu panna sageli `PATH`
`zsh`	`~/.zshrc`	`~/.zprofile` või `~/.zshrc`
`bash`	`~/.bashrc`	`~/.bash_profile` või `~/.bashrc`

Kui sa ei taha veel startup-failide nüanssidesse süveneda, alusta oma aktiivse shelli interaktiivsest failist:

- `zsh`: `~/.zshrc`
- `bash`: `~/.bashrc`

Näide `zsh` jaoks:

```
cat >> ~/.zshrc <<'EOF'
export PATH="$HOME/bin:$PATH"
alias ll='ls -lah'
EOF
source ~/.zshrc
```

Näide `bash` jaoks:

```
cat >> ~/.bashrc <<'EOF'
export PATH="$HOME/bin:$PATH"
alias ll='ls -lah'
EOF
source ~/.bashrc
```

Shelli seadistusfailide põhjalikumad näited on lisas Lisa F: shelli seadistusfailid bash ja zsh jaoks.

Alias'ed

Alias on lühinimi sagedase käsu jaoks.

```
alias ll='ls -lah'
alias la='ls -A'
alias gs='git status'
alias h='history | tail -n 20'
```

Alias sobib hästi siis, kui ta lühendab tuttavat käsku, aga ei peida olulist loogikat.

Vaata olemasolevaid alias'eid:

```
alias
type ll
```

Eemalda alias praeguses shellis:

```
unalias ll
```

Väldi alias'eid, mis teevad ohtliku käsu liiga "mugavaks". Näiteks `rm` ümberkirjutamine võib aidata ühes shellis, aga tekitada vale turvatunde teises masinas.

Miks `which` võib petta

Paljud kasutavad käsku:

```
which grep
```

Õppimiseks eelista:

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

Põhjus:

- `which` keskendub sageli välistele programmidele
- `type` näitab ka alias'ed, built-in käsud ja shelli vaatenurga
- `command -v` on lühike ning sobib hästi kontrolliks

Kui "käsk käitub imelikult", alusta kontrollist:

```
type käsk
type -a käsk
```

Prompt ja ajalugu lühidalt

Prompt on tekst enne käsu sisestamist. See ei ole käsu osa.

Näide:

```
~/projekt %
```

Kui prompt näitab üht asja ja oled ebakindel, kontrolli käsuga:

```
pwd
```

Ajutine lihtne prompt `zsh`-is:

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

Ajutine lihtne prompt `bash`-is:

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

Proovi prompti muutmist alguses ainult jooksvas shellis. Püsiv prompt kuulub hiljem `~/.zshrc` või `~/.bashrc` faili.

Ajaloo vaatamiseks piisab alguses:

```
history
```

Kui torud ja filtrid on juba tuttavad:

```
history | tail -n 20
```

Mõned sagedased keskkonnamuutujad

Muutuja	Tähendus
`PATH`	kust väliseid käske otsitakse
`HOME`	sinu kodukataloog
`SHELL`	sinu shelli programm
`EDITOR`	vaikimisi tekstiredaktor paljude tööriistade jaoks
`LANG`	keele- ja lokaadiseadistus

Näited:

```
echo "$HOME"
echo "$SHELL"
echo "$LANG"
export EDITOR=nano
```

Kui mõni programm avab ootamatult teise redaktori, kontrolli `EDITOR` muutujat.

Minitest

1. Loo shellimuutuja `nimi='Mari'` ja kuva see.
2. Loo keskkonnamuutuja `DEMO=1` ning kontrolli seda käsuga `env`.
3. Vaata oma `PATH` muutujat.
4. Kontrolli, kust leitakse `python3` või `bash`.
5. Võrdle käske `type cd` ja `command -v cd`.
6. Loo alias `ll='ls -lah'` ja kontrolli seda käsuga `type ll`.
7. Lisa `~/bin` ajutiselt `PATH`-i ette ja kontrolli, et rida muutus.
8. Selgita ühe lausega, miks `PATH` järjekord võib olla ohtlik.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: erista shellimuutujat, keskkonnamuutujat, käsuotsingu teed ja aliast; PATH-i järjekord otsustab, milline väline käsk esimesena leitakse.

Põhikujud

- `nimi='Mari'; echo "$nimi"` — shellimuutuja kehtib selles shellis
- `export DEMO=1` — anna muutuja edasi käivitatavatele programmidele
- `env | grep DEMO` — kontrolli, kas muutuja on keskkonnas
- `echo "$PATH"` — vaata käskude otsinguteed
- `command -v python3` — leia esimene väline vaste
- `type cd` — vaata käsu liiki
- `type -a grep` — näita kõik vasted
- `export PATH="$HOME/bin:$PATH"` — pane oma käskude kataloog otsinguteе ette
- `alias ll='ls -lah'` — loo lühinimi sagedasele käsule
- `source ~/.zshrc` — loe zsh seadistus uuesti

Olulised mõisted

- `nimi='Mari'` — shellimuutuja
- `export NIMI='Mari'` — keskkonnamuutuja
- `PATH` — otsinguteede järjekord
- `$HOME/bin` — oma käskude koht
- `type` — alias, builtin või programm

- `~/.zshrc` — zsh seadistusfail
- `~/.bashrc` — bash seadistusfail

Pane tähele: PATH-i järjekord loeb: vasakul olev vaste leitakse enne; ära lisa tundmatut kataloogi ettepoole lihtsalt sellepärast, et juhend nii ütleb.

Edasi: Järgmine loomulik samm: Paketihaldus: apt, dnf, pacman, brew.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Paketihaldus: apt, dnf, pacman, brew

Paketihaldur paigaldab tarkvara koos vajalike sõltuvustega ja hoiab süsteemi seisundi jälgitavana. Peamine oskus on valida õige tase: kas paigaldad tööriista kogu süsteemile, oma kasutajale või ainult ühele projektile.

Loogika

Paketihalduses on kolm taset:

1. süsteemi paketihaldur paigaldab käsureatööriistu ja süsteemiteeke
2. keele paketihaldur paigaldab ühe programmeerimiskeele teeke
3. projekti sõltuvused kuuluvad võimalusel projekti kataloogi või virtuaalkeskkonda

Kõige sagedasem algaja viga on paigaldada kõik ühe käsuga "kuhugi süsteemi", mõtlemata, kes seda hiljem kasutab ja kust see leitakse.

Kiire orientiir

Tööriist	Kus tavaliselt	Milleks
`apt`	Debian, Ubuntu	süsteemi paketid
`dnf`	Fedora	süsteemi paketid
`pacman`	Arch Linux	süsteemi paketid
`brew`	macOS, vahel Linux	kasutajataseme tööriistad
`python3 -m pip`	Python	Pythoni paketid
`npm`	Node.js	JavaScripti projektid

Vali korraga ainult oma süsteemile või projektile sobiv rida. Ubuntu kasutaja ei käivita Fedora käske ja macOS-i kasutaja ei paigalda Homebrew pakette käsuga `sudo apt`.

Tüüpilised algaja vead

- käivitatakse mitu eri distributsiooni paigalduskäsku järjest

- kasutatakse `sudo`-t seal, kus paketihaldur seda ei vaja, näiteks Homebrew puhul
- paigaldatakse Pythoni projektiteek süsteemi, kuigi õigem oleks `venv`
- arvatakse, et Homebrew on macOS-is juba olemas
- ei loeta paigalduslogi lõppu, kus sageli on tegelik veateade või järgmine juhis

Vali oma süsteemi käsk

Otsi paketti enne paigaldamist:

```
apt search ripgrep
dnf search ripgrep
pacman -Ss ripgrep
brew search ripgrep
```

Paigalda ainult oma süsteemi rida:

```
sudo apt install ripgrep
sudo dnf install ripgrep
sudo pacman -S ripgrep
brew install ripgrep
```

Kontrolli pärast paigaldust:

```
rg --version
command -v rg
```

Kui paigaldus õnnestub, näed tavaliselt logi, allalaadimist, sõltuvusi ja lõpuks vaikset käsuviipa. Kui käsk ei ilmu pärast paigaldust nähtavale, kontrolli `PATH`-i ja ava vajadusel uus terminaliaken.

Miks mõnikord on vaja `sudo`

Linuxi süsteemipaketid muudavad tavaliselt süsteemi katalooge. Seetõttu kasutatakse tihti:

```
sudo apt install htop
```

Homebrew töötab enamasti kasutaja õigustes:

```
brew install htop
```

Ära lisa `sudo`-t lihtsalt harjumusest. Kui paketihaldur seda ei küsi, on parem jätta see kasutamata.

`update` ja `upgrade`

Nimed on sarnased, aga tähendus on erinev.

Kuju	Mida teeb
`sudo apt update`	värskendab paketinimekirja
`sudo apt upgrade`	uuendab juba paigaldatud paketid
`brew update`	värskendab Homebrew pakiretsepte
`brew upgrade`	uuendab Homebrew paketid

Ära käivita suurt `upgrade`-i siis, kui sul on kohe vaja arvutit tööks või esitluseks. Uuendamine võib võtta aega ja mõnikord küsida lisavalikuid.

Uuendamine, info ja eemaldamine

Debianis ja Ubuntus värskendab `apt update` paketinimekirja, mitte veel kõiki programme.

```
sudo apt update
apt show ripgrep
sudo apt install ripgrep
sudo apt remove ripgrep
```

Fedoras on tavaline kuju:

```
dnf info ripgrep
sudo dnf install ripgrep
sudo dnf remove ripgrep
```

Arch Linuxis otsitakse ja paigaldatakse nii:

```
pacman -Ss ripgrep
sudo pacman -S ripgrep
sudo pacman -R ripgrep
```

Homebrew puhul:

```
brew info ripgrep
brew install ripgrep
brew uninstall ripgrep
```

Eemaldamise näited on siin väikese tööriista peal. Ära eemalda juhuslikult süsteemiteeke ega pakette, mille rollist sa aru ei saa.

Süsteemi paketihaldur ja keele paketihaldur

`apt install python3-requests` ja `python3 -m pip install requests` ei ole sama asi.

Kuju	Tase	Millal kasutada
`sudo apt install python3-requests`	süsteem	kui distributsioonipakett on teadlik valik
`python3 -m pip install requests`	Python	kui paigaldad Pythoni paketi aktiivsesse keskkonda
`npm install`	projekt	kui projektis on `package.json`

Pythoni projektis on tavaliselt parem kasutada virtuaalkeskkonda:

```
python3 -m venv .venv
source .venv/bin/activate
python3 -m pip install requests
python3 -m pip show requests
```

Node.js projektis paigaldatakse sõltuvused projekti kataloogis:

```
npm install
npm list --depth=0
npm run dev
```

Kui käsk `npm run dev` ei tööta, vaata kõigepealt `package.json` faili. Seal on kirjas, millised `scripts` nimed on üldse olemas.

`pip` kuju

Pythoni puhul eelista kuju:

```
python3 -m pip install requests
```

mitte lihtsalt:

```
pip install requests
```

Nii on selgem, millise Pythoni tõlgendiga pakette paigaldatakse. See muutub eriti tähtsaks siis, kui arvutis on mitu Pythoni versiooni või aktiivne virtuaalkeskkond.

macOS ja Homebrew

macOS-is on osa Unixi käske juba olemas, aga Homebrew tuleb tavaliselt ise paigaldada. Seda ei pea tegema enne, kui vajad tööriista, mida süsteemis pole või mille uuemat versiooni on vaja.

Näiteks:

```
brew install ripgrep jq tmux gh
```

Graafilisi rakendusi paigaldatakse Homebrew's sageli `--cask` võtmega:

```
brew install --cask docker-desktop
```

Kui vajad macOS-is Linuxi võrgukäskude sarnaseid nimesid, on üks võimalus:

```
brew install iproute2mac
```

See on valikuline lisakiht. Enamasti piisab macOS-is sisseehitatud käskudest `ifconfig`, `lsof` ja `netstat`.

Kontrollkäigud pärast paigaldust

Kontrolli tööriista olemasolu ja versiooni:

```
command -v python3
python3 --version
python3 -m pip --version
command -v node
node --version
npm --version
```

Kui käsk on paigaldatud, aga shell seda ei leia, kontrolli:

```
echo "$PATH"
command -v käsk
type -a käsk
```

Kui paigaldus ebaõnnestub

Kõigepealt loe veateate viimaseid ridu. Seal on tavaliselt põhjus.

Sümptom	Tüüpiline põhjus	Esimene kontroll
`package not found`	nimi on vale või nimekiri vana	otsi paketti ja tee vajadusel `update`
`permission denied`	käsk vajab kõrgemaid õigusi	kasuta süsteemipaketiga `sudo`, aga mitte pimesi
`lock` või `database is locked`	teine paketihaldur töötab	oota, ära kustuta lukufaile käsitsi
`command not found` pärast paigaldust	käsk pole `PATH`-is või nimi on teine	`command -v nimi`, `type -a nimi`
`pip` või `npm` õiguste viga	paigaldad valesse tasemesse	kasuta projekti virtuaalkeskkonda või projekti kataloogi

Minitest

1. Ütle, milline süsteemi paketihaldur sobib sinu masinale.

2. Otsi sellega paketti `ripgrep` või `htop`.
3. Loe ühe paketi infot ilma seda paigaldamata.
4. Selgita, miks `pip install` ja `apt install` ei ole sama tase.
5. Kontrolli, kas `python3`, `pip`, `node` ja `npm` on sinu masinas olemas.
6. Kui kasutad macOS-i, kontrolli, kas `brew` on olemas käsuga `command -v brew`.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: vali õige tase: süsteemi paketihaldur tööriistadele, keele paketihaldur teekidele ja projektisõltuvused võimalusel projekti sisse

Põhikujud

- `apt search ripgrep` — otsi Debianis või Ubuntus
- `sudo apt install ripgrep` — paigalda Debianis või Ubuntus
- `sudo dnf install ripgrep` — paigalda Fedoras
- `sudo pacman -S ripgrep` — paigalda Archis
- `brew install ripgrep` — paigalda Homebrew'ga
- `python3 -m pip install requests` — paigalda Pythoni pakett aktiivsesse keskkonda
- `npm install` — paigalda projekti Node.js sõltuvused
- `command -v rg` — kontrolli, kas käsk leitakse

Tööriistad ja tasemed

- `apt, dnf, pacman` — Linuxi süsteemipaketid
- `brew` — macOS-i kasutajataseme tööriistad
- `update` — värskenda nimekirja või retsepte
- `upgrade` — uuenda paigaldatud paketid
- `--cask` — Homebrew graafiline rakendus
- `venv` — Pythoni projekti eraldi keskkond
- `package.json` — Node.js projekti sõltuvused

Pane tähele: Ära aja segi süsteemi paketihaldurit ja pip/npm-i; need lahendavad eri taseme probleeme.

Edasi: Järgmine loomulik samm: Lihtne veaotsing käsureal.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Lihtne veaotsing käsureal

Veaotsing ei alga uue käsu proovimisest, vaid olukorra kitsendamisest. Enamiku algaja vigade puhul aitab lühike kontrollrada: mida täpselt käivitati, mida terminal vastas, kus sa oled, kas fail või käsk on olemas ja kas õigused lubavad.

Loogika

Hea käsurea veaotsing liigub kitsamaks:

1. kirjuta üles täpne käsk ja täpne veateade
2. kontrolli asukohta ja failinime
3. kontrolli, kas käsk on leitav
4. kontrolli õigusi ja skripti esimest rida
5. kontrolli, kas probleem on paketis, shellis või teekonnas

Ära alusta juhusliku "paranduskäsu" kopeerimisest. Kui esimene oletus on vale, võib järgmine käsk teha olukorra segasemaks.

Kiire orientiir

Kontroll	Käsk	Mida näed
kus olen	`pwd`	praegune täistee
mis siin on	`ls -lah`	failid, kaustad, õigused, peidetud kirjed
kas käsk leitakse	`command -v rg`	failitee või tühi väljund
mis liiki käsk see on	`type -a rg`	alias, builtin või failitee
kas fail lubab	`ls -l skript.sh`	õigused ja omanik
mis shelli skript küsib	`head -n 1 skript.sh`	näiteks `#!/usr/bin/env bash`
kas süntaks on õige	`grep --help`, `man grep`	lühike abi või käsiraamat

Kõige väärtuslikum info on täpne veateade. "Ei tööta" ei ole veel diagnoos; `permission denied`, `command not found` või `No such file or directory` on juba diagnoosi algus.

Tüüpilised algaja vead

- veateadet ei kopeerita täpselt
- eeldatakse, et ollakse õiges kataloogis, aga `pwd` ütleb midagi muud
- aetakse segi puuduv fail ja puuduv õigus
- kasutatakse `sudo`-t enne, kui on selge, miks õigused ei luba
- parandatakse `PATH`-i, kuigi käsu nimi oli lihtsalt valesti kirjutatud
- käivitatakse Bashi skript `sh`-iga ja saadakse süntaksiviga

Kolme minuti kontrollrada

Kui käsk ei tööta, tee kõigepealt see väike rada.

```
pwd
ls -lah
```

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

Kui probleem on skriptiga:

```
ls -l skript.sh
head -n 1 skript.sh
bash skript.sh
```

Need käsud ei paranda veel midagi. Nad annavad pildi: asukoht, failid, käsu leidmine, shell ja skripti käivitusviis.

Levinud veateated

Veateade	Tõenäoline põhjus	Esimene kontroll
`command not found`	käsk on valesti kirjutatud, paigaldamata või pole `PATH`-is	`command -v rg`, `echo "$PATH"`
`No such file or directory`	failinimi, tee või praegune kataloog on vale	`pwd`, `ls`, `ls tee/fail`
`Permission denied`	puudub lugemis-, kirjutamis- või täitmisõigus	`ls -l fail`, `ls -ld kaust`
`syntax error`	käsku tõlgendab vale shell või süntaks on vale	`echo "$SHELL"`, `head -n 1 skript.sh`
`package not found`	paketihalduri nimi on vale või nimekiri vana	paketihalduse otsing ja vajadusel `update`
`database is locked`	teine paketihaldur töötab	oota; ära kustuta lukufaile käsitsi

`command not found`

See viga tähendab, et shell ei leidnud käivitatavat käsku selle nime järgi.

Kontrolli:

```
command -v rg
type -a rg
echo "$PATH"
```

Kui `command -v` ei näita midagi, on järgmised küsimused:

- kas käsu nimi on õige
- kas tööriist on paigaldatud
- kas paigaldatud tööriista kataloog on `PATH`-is

Kui nimi on õige ja tööriist puudub, mine peatükki Paketihaldus: apt, dnf, pacman, brew.

`No such file or directory`

See viga tähendab enamasti, et viitad failile või kataloogile, mida selle tee järgi ei leita.

Kontrolli:

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

Pööra tähelepanu kolmele asjale:

- suur- ja väiketähed peavad klappima
- tühikutega failinimed vajavad jutumärke
- suhteline tee sõltub praegusest kataloogist

Näiteks:

```
ls "Minu fail.txt"
ls ./skript.sh
ls ../andmed/sisend.txt
```

`Permission denied`

See viga ei tähenda automaatselt, et pead kasutama `sudo`-t. Kõigepealt vaata, milline õigus puudu on.

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

Kui skript ei käivitu, kontrolli õigusi ja proovi seda teadlikult shelliga:

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

Kui skript peab olema otse käivitatav, siis täitmisõigus lisatakse näiteks nii:

```
chmod u+x skript.sh
./skript.sh
```

Kui probleem on süsteemikaustas, peatu korraks. Küsimus ei ole ainult käsus, vaid selles, kas sul peaks üldse olema õigus seda kohta muuta.

Vale shell või vale süntaks

Shellid on sarnased, kuid mitte samad. Bashi skript võib anda vea, kui käivitad selle `sh`-iga.

Kontrolli aktiivset shelli:

```
echo "$SHELL"
```

Kontrolli skripti esimest rida:

```
head -n 1 skript.sh
```

Bashi skripti tavaline esimene rida on:

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

Kui tahad olla kindel, et skripti käivitab Bash:

```
bash skript.sh
```

Kui käsk ise on võõras, kinnita süntaks enne:

```
grep --help
man grep
```

Kui probleem tuli pärast paigaldust

Paigaldus võib õnnestuda, aga uus käsk ei ilmu kohe sinu shelli nähtavale.

Kontrolli:

```
command -v htop
type -a htop
echo "$PATH"
```

Kui pakett paigaldati Homebrew, `pip` või mõne projekti tööriistaga, võib vaja olla:

- avada uus terminaliaken
- aktiveerida õige virtuaalkeskkond
- kontrollida, mis nimega käivitatav käsk päriselt paigaldati

Hea veakirjeldus

Kui küsid abi inimeselt või foorumist, pane kaasa vähemalt:

1. mida tahtsid teha
2. täpne käsk
3. täpne veateade
4. `pwd` väljund
5. süsteem või keskkond, näiteks macOS, Ubuntu, WSL või terminal.cs.ut.ee

Hea veakirjeldus säästab aega ja vähendab juhuslike oletuste hulka.

Minitest

1. Kontrolli, kas sinu masinas leitakse `python3`.

2. Tee meelega üks vale failinimi ja vaata täpset veateadet.
3. Selgita, mis vahe on vigadel `command not found` ja `No such file or directory`.
4. Vaata ühe skripti õigusi käsuga `ls -l`.
5. Kontrolli ühe skripti esimest rida käsuga `head -n 1`.
6. Kirjuta näidis-veakirjeldus, kus on käsk, veateade ja `pwd` väljund.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: kitsenda viga enne parandamist: täpne veateade, asukoht, olemasolu, käsu leitavus, õigused ja shelli/süntaksi kontroll

Kontrollrada

- `pwd` — kontrolli, kus oled
- `ls -lah` — vaata faile, õigusi ja peidetud kirjeid
- `command -v rg` — kas käsk leitakse
- `type -a rg` — näita kõik sama nimega vasted
- `ls -l skript.sh` — vaata skripti õigusi
- `head -n 1 skript.sh` — vaata skripti shebang'i
- `bash skript.sh` — käivita teadlikult Bashiga
- `grep --help` — kinnita käsu süntaks

Levinud veateated

- `command not found` — käsk puudub, nimi vale või pole PATH-is
- `No such file or directory` — tee, failinimi või asukoht on vale
- `Permission denied` — õigus ei luba lugeda, kirjutada või käivitada
- `syntax error` — vale shell või vale käsukuju
- `package not found` — paketinimi või nimekiri on vale
- `database is locked` — teine paketihaldur töötab

Pane tähele: Pane kõigepealt kirja täpne käsk ja täpne veateade; ilma nendeta muutub veaotsing oletamiseks.

Edasi: Järgmine loomulik samm: Võrgu põhitööriistad.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Võrgu põhitööriistad

Võrgu veaotsingus on tähtis eristada mitut eri küsimust. "Internet ei tööta", "veebileht ei vasta" ja "minu programm ei kuula porti" võivad välja näha sarnased, aga kontrollitakse neid eri käskudega.

Loogika

Esimene võrgukontroll liigub tavaliselt nii:

1. kas nimi või host vastab üldse
2. kas HTTP/HTTPS teenus vastab
3. millised võrguliidesed ja aadressid sinu masinal on
4. kas kohalik teenus kuulab õiget porti

`ping` ei kontrolli veebilehte. `curl -I` ei näita sinu võrguliideseid. `ss` ja `lsof` ei ütle, kas DNS-nimi laheneb. Iga tööriist vastab oma kitsale küsimusele.

Kiire orientiir

Küsimus	Linux	macOS	Mida tavaliselt näed
kas host vastab	`ping -c 4 example.com`	`ping -c 4 example.com`	vastuseread või paketikadu
kas veeb vastab	`curl -I https://example.com/`	`curl -I https://example.com/`	HTTP staatus ja päised
millised liidesed on	`ip a`	`ifconfig`	liidesed ja IP-aadressid
mis pordid kuulavad	`ss -ltn`	`lsof -iTCP -sTCP:LISTEN -n -P`	kuulavad TCP-pordid

Kui kasutad macOS-i ja `ip a` või `ss` annab `command not found`, siis see on normaalne. Need on Linuxi tööriistad; macOS-is kasuta alguses `ifconfig` ja `lsof`.

Tüüpilised algaja vead

- peetakse `ping` tulemust veebiteenuse lõplikuks kontrolliks
- unustatakse, et mõni server ei vasta `ping`-ile, kuigi veeb töötab
- kasutatakse Linuxi käske `ip` ja `ss` macOS-is ning arvatakse, et midagi on katki
- vaadatakse kuulavat porti, aga ei eristata `127.0.0.1`, `0.0.0.0` ja päris võrguaadressi
- kontrollitakse kauget teenust, kuigi probleem on enda arvutis käivitatud programmis

Esimene kontrollrada

Veebiteenuse puhul alusta HTTP-st:

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

Kui tahad näha, kas host vastab ICMP-le:

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

Kui uurid oma masina võrku, vali süsteemi järgi üks rida:

```
ip a
ifconfig
```

Kui uurid, kas kohalik teenus kuulab porti, vali süsteemi järgi üks rida:

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

Ära käivita Linuxi ja macOS-i variante läbisegi. Vali see, mis sinu süsteemis olemas on.

`ping`: hosti esimene signaal

`ping` saadab ICMP-pakette ja mõõdab, kas host vastab.

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

Tulemuse lugemine:

- vastuseread tähendavad, et host vastas ICMP-le
- paketikadu tähendab, et osa päringuid ei saanud vastust
- vastuse puudumine ei tõesta veel, et veebiteenus on maas

Paljud serverid piiravad või blokeerivad `ping` vastuseid. Seetõttu on veebiteenuse kontrolliks parem järgmine käsk.

`curl -I`: veebiteenuse kontroll

`curl -I` küsib ainult HTTP päised, mitte kogu lehe sisu.

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

Olulised read:

Rida	Tähendus
`HTTP/2 200` või `HTTP/1.1 200`	server vastas edukalt
`301`, `302`, `308`	server suunab teisele aadressile
`404`	aadressi ei leitud
`500` seeria	serveripoolne viga

Kui tahad suunamist jälgida, lisa `-L`:

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

Võrguliidesed: `ip a` ja `ifconfig`

Liides on võrguühendus sinu arvutis: Wi-Fi, kaabel, VPN, loopback või virtuaalne adapter.

Linuxis:

```
ip a
```

macOS-is:

```
ifconfig
```

Alguses otsi ainult kahte asja:

- kas mõnel liidesel on IP-aadress
- kas näed loopback-aadressi `127.0.0.1`

`127.0.0.1` tähendab sinu enda masinat. See on kasulik kohalike teenuste testimisel, aga ei tähenda, et teenus oleks teistele arvutitele nähtav.

Kuulavad pordid

Kui programm peab teenust pakkuma, peab ta tavaliselt kuulama mingit porti. Näiteks:

- `22` SSH
- `80` HTTP
- `443` HTTPS
- `3000`, `5173`, `8000`, `8080` arenduse kohalikud serverid

Linuxis:

```
ss -ltn
```

macOS-is:

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

Teine macOS-i sisseehitatud vaade:

```
netstat -anv -p tcp | grep LISTEN
```

Vaata eriti aadressi osa:

Kuju	Tähendus
`127.0.0.1:3000`	teenus kuulab ainult sinu arvutis
`0.0.0.0:3000`	teenus kuulab kõigil IPv4 liidestel
`[::]:3000`	teenus kuulab IPv6 kaudu kõigil liidestel

Kui brauseris `http://localhost:3000` töötab, aga teiselt masinalt ligi ei saa, võib põhjus olla just selles, et teenus kuulab ainult `127.0.0.1` peal või tulemüür ei luba ühendust.

Kas port on kaugelt avatud

Kui tahad kontrollida konkreetset hosti ja porti, sobib sageli `nc`.

```
nc -vz example.com 443
```

See ei tõenda, et rakendus töötab õigesti, aga näitab, kas TCP-ühendus porti õnnestub.

Kui `nc` pole olemas, kasuta veebiteenuse puhul `curl -I`-d. HTTP jaoks annab see tavaliselt rohkem infot.

macOS: kas pean midagi paigaldama?

Alguses ei pea. macOS-is on võrgukontrolliks piisavalt sisseehitatud tööriistu:

- `ping`
- `curl`
- `ifconfig`
- `lsof`
- `netstat`
- sageli ka `nc`

Kui tahad hiljem Linuxi `ip`-stiilis käsku, saab lisada:

```
brew install iproute2mac
```

See on mugav lisakiht, mitte õpiku eeldus.

Minitest

1. Tee `curl -I` päring aadressile `https://example.com/`.
2. Tee `ping -c 4 example.com` ja võrdle, kas see vastab samale küsimusele.
3. Vaata oma süsteemis võrguliideseid: Linuxis `ip a`, macOS-is `ifconfig`.
4. Vaata, kas sinu masinas kuulab mõni TCP-port.
5. Selgita ühe lausega, mis vahe on aadressidel `127.0.0.1:3000` ja `0.0.0.0:3000`.

Peatüki täisspikker

Tase: **Baas ja süsteemipilt**

Eesmärk: erista nelja küsimust: kas host vastab, kas veeb vastab, millised liidesed on ja kas kohalik teenus kuulab porti

Kontrollrada

- `curl -I https://example.com/` — kontrolli veebiteenust
- `ping -c 4 example.com` — kontrolli hosti ICMP vastust
- `ip a` — vaata Linuxi liideseid
- `ifconfig` — vaata macOS-i liideseid

- `ss -ltn` — vaata Linuxi kuulavaid TCP-porte
- `lsof -iTCP -sTCP:LISTEN -n -P` — vaata macOS-i kuulavaid porte
- `nc -vz example.com 443` — kontrolli TCP ühendust porti

Olulised võtmed ja vaated

- `-I` — küsi ainult HTTP päised
- `-L` — järgi HTTP suunamisi
- `LISTEN` — port kuulab
- `127.0.0.1` — ainult oma masin
- `0.0.0.0` — kõik IPv4 liidesed
- `ip/ss` — Linuxi tööriistad
- `ifconfig/lsof` — macOS-i põhivariandid

Pane tähele: ping ja HTTP ei ole sama asi; veeb võib töötada ka siis, kui host pingile ei vasta.

Edasi: Järgmine loomulik samm: Failide kopeerimine ja sünkroonimine.

Osa PDF: ./spikrid/osa-ii-susteemi-pilt-ja-haldus-spikker.pdf

Failide kopeerimine ja sünkroonimine

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

Kiirülevaade

Eesmärk on valida kopeerimiseks õige tööriist ja enne suuremat sünkroonimist kontrollida, mida tehtaks.

Käsk	Milleks	Mida tavaliselt näed
`cp`	kopeeri samas masinas	edukal juhul sageli vaikne
`scp`	kopeeri üle SSH	ülekande edenemine

Käsk	Milleks	Mida tavaliselt näed
`rsync`	sünkrooni muutused	`-v` näitab tehtut, `-n` näitab plaani
`wget`	laadi URL-ist fail	edenemine ja salvestatud nimi
`curl`	tee HTTP-päring või laadi fail	päised, sisu või edenemine

`rsync -n` on enne suuremat kopeerimist hea turvapidur: ta näitab, mida tehtaks, aga ei muuda veel midagi.

Tüüpilised algaja vead

- valitakse vale tööriist: `scp` korduva sünkroonimise jaoks või `curl` seal, kus vaja oleks lihtsalt `wget`
- unustatakse `rsync` allika lõpu kaldkriips
- jäetakse `--dry-run` või `-n` vahele enne suurt sünkroonimist
- eeldatakse, et kopeeriv käsk peab alati midagi ekraanile kirjutama

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 siht-kausta 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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: vali tööriist allika järgi: cp samas masinas, scp üle SSH, rsync korduvaks sünkroonimiseks ja curl/wget veebist toomiseks

Põhikujud

- `cp fail.txt koopia.txt` — tee koopia
- `cp -R kaust kaust-koopia` — kopeeri kaust
- `scp fail.txt kasutaja@server:/tmp/` — saada serverisse
- `rsync -avn kaust/ kasutaja@server:/tmp/kaust/` — kontrolli enne
- `rsync -av kaust/ kasutaja@server:/tmp/kaust/` — sünkrooni sisu
- `wget https://example.com/arhiiv.tar.gz` — laadi fail
- `curl -L -O https://example.com/arhiiv.tar.gz` — järgi ja salvesta

Olulisemad lipud, märgid ja kiirnupud

- `cp -R` — kopeeri kaust
- `scp -r` — saada kaust
- `rsync -a` — säilita struktuur
- `rsync -n` — proovi enne
- `curl -L` — järgi suunamist
- `curl -O` — salvesta nimega

Pane tähele: Vali tööriist allika järgi: cp samas masinas, scp üle SSH, rsync korduvaks sünkroonimiseks, curl/wget veebist toomiseks.

Edasi: Järgmine loomulik samm: Kauglogimine ja SSH.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Kauglogimine ja SSH

SSH on turvaline viis logida teise masinasse, käivitada seal käske ja liigutada faile. Sama ühendusloogikat kasutavad sageli ka `scp`, `rsync` ja Git.

Loogika

SSH töövoog liigub tavaliselt nii:

1. proovi parooli või olemasoleva võtmega tavalist sisselogimist
2. kontrolli, kas sul on võtmepaar juba olemas
3. loo uus võti ainult siis, kui seda on vaja
4. lisa avalik võti serverisse
5. pane korduv ühendus `~/.ssh/config` faili alias'e alla
6. macOS-is lase Keychainil võtme paroolifraasi meeles pidada

Kõige tähtsam piir on lihtne: privaatvõti jääb sinu arvutisse, avalik võti läheb serverisse.

Kiire orientiir

Küsimus	Käsk või fail	Mida näed või teed
kas saan sisse	`ssh kasutaja@server.example.org`	kaugmasina viip või veateade
mis võtmed failina olemas on	`ls -la ~/.ssh`	`id_ed25519`, `id_ed25519.pub` jms
mis võtmed agendis on	`ssh-add -l`	võtmete sõrmejäljed või teade, et agent on tühi
loo uus võti	`ssh-keygen -t ed25519`	privaat- ja avaliku võtme fail
lisa võti macOS Keychaini	`ssh-add --apple-use-keychain ~/.ssh/id_ed25519`	paroolifraas salvestatakse Keychaini
kasuta alias't	`ssh opik-server`	ühendus `~/.ssh/config` kirje järgi

Esmakordsel ühendusel küsib SSH sageli serveri sõrmejälje kinnitamist. See on serveri tuvastus, mitte sinu kasutajavõti. Tundmatu serveri puhul kontrolli sõrmejälge usaldusväärsest allikast.

Tüüpilised algaja vead

- aetakse segi privaatvõti ja avalik võti
- kirjutatakse olemasolev võti kogemata üle

- arvatakse, et Apple Keychain asendab `~/.ssh` võtmefailid
- lisatakse serverisse privaatvõti, kuigi lisada tuleb ainult `.pub` fail
- tehakse keeruline `ssh -L` või agent forwarding enne, kui tavaline sisselogimine töötab
- unustatakse, et `~/.ssh/config` alias'e saavad kasutada ka `scp`, `rsync` ja Git

Esimene ühendus

Kõigepealt proovi kõige tavalisemat kuju:

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

Kui port ei ole tavaline 22, lisa port:

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

Ühekordse kaugkäsu saab anda nii:

```
ssh kasutaja@server.example.org 'hostname && uptime'
```

Kui see veel ei tööta, ära alusta võtmete ümbertegemisest. Enne kontrolli kasutajanime, hostinime, porti ja veateadet.

Kas sul on võtmed juba olemas?

Vaata kõigepealt oma `~/.ssh` kataloogi.

```
ls -la ~/.ssh
```

Tüüpilised failid:

Fail	Tähendus
`id_ed25519`	privaatvõti; ära saada kellelegi
`id_ed25519.pub`	avalik võti; see lisatakse serverisse
`config`	ühenduste alias'ed ja seaded
`known_hosts`	serverite sõrmejäljed
`authorized_keys`	serveris lubatud avalikud võtmed

Kontrolli võtme sõrmejälge:

```
ssh-keygen -lf ~/.ssh/id_ed25519.pub
```

Kontrolli, mis võtmed on praegu `ssh-agent`-is:

```
ssh-add -l
ssh-add -L
```

`ssh-add -l` näitab sõrmejälgi. `ssh-add -L` näitab agendis olevate võtmete avalikke ridu. Kui agent on tühi, ei tähenda see, et võtmefaile pole olemas; see tähendab ainult, et agent ei hoia neid praegu mälus.

Uue võtme loomine

Ära kirjuta olemasolevat võtit üle. Kui `~/.ssh/id_ed25519` on juba olemas ja kasutusel, kasuta seda või loo eraldi nimega võti.

Tavaline uus võti:

```
ssh-keygen -t ed25519 -C 'kasutaja@arvuti'
```

Kui tahad eraldi õpiku või serveri võtit:

```
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519_opik -C 'opik-server'
```

Soovitus: pane privaatvõtmele paroolifraas. Siis kaitseb võtmefaili ka see, kui fail satub kogemata valesse kohta.

Õigused peavad olema kitsad:

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/id_ed25519
chmod 644 ~/.ssh/id_ed25519.pub
```

Kui kasutasid teise nimega võtit, asenda käsus failinimi vastavaks.

macOS: Keychain ja ssh-agent

macOS-is on kaks eri asja:

- `ssh-agent` hoiab võtmeid jooksva kasutusseansi ajal mälus
- Apple Keychain saab hoida privaatvõtme paroolifraasi, et sa ei peaks seda iga kord uuesti sisestama

Keychain ei asenda võtmefaile. Privaatvõti on endiselt näiteks `~/.ssh/id_ed25519`; Keychain aitab selle paroolifraasi hallata.

Lisa võti agendile ja salvesta paroolifraas Keychaini:

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

Kui paroolifraas on juba Keychainis ja tahad võtmed agendile laadida:

```
ssh-add --apple-load-keychain
```

Kui tahad võtme ainult jooksvast agendist eemaldada:

```
ssh-add -d ~/.ssh/id_ed25519
```

Kui tahad eemaldada ka Keychaini salvestatud paroolifraasi, lisa eemaldamisel Apple'i Keychaini valik:

```
ssh-add --apple-use-keychain -d ~/.ssh/id_ed25519
```

Vanemates macOS-i juhendites näeb vahel käske `ssh-add -K` ja `ssh-add -A`. Tänapäevasem ja selgem kuju on kasutada `--apple-use-keychain` ja `--apple-load-keychain`.

Kontrolli pärast:

```
ssh-add -l
```

Kui `ssh-add` ütleb, et agenti ei saa ühendada, ava uus terminaliaken ja proovi uuesti. macOS-is käivitab graafiline kasutajaseanss agendi tavaliselt ise.

`~/.ssh/config`: alias'ed ja püsiseaded

Kui ühendud sama serveriga korduvalt, tee config-faili alias.

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

Pärast seda töötavad lühemad käsud:

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

`IdentitiesOnly yes` on kasulik siis, kui agendis on palju võtmeid. See ütleb SSH-le, et selle hosti puhul kasuta eelkõige siin nimetatud võtit.

macOS-i mugav plokk võib olla selline:

```
Host *
    IgnoreUnknown UseKeychain
    AddKeysToAgent yes
    UseKeychain yes

Host opik-server
    HostName server.example.org
    User kasutaja
    IdentityFile ~/.ssh/id_ed25519
    IdentitiesOnly yes
```

`IgnoreUnknown UseKeychain` peab olema enne `UseKeychain` rida, sest Linuxi OpenSSH ei pruugi Apple'i `UseKeychain` valikut tunda. Nii saab sama `config` fail olla talutavam mitmes süsteemis.

Config-faili õigused:

```
chmod 600 ~/.ssh/config
```

Avaliku võtme lisamine serverisse

Serverisse lisatakse avalik võti, mitte privaatvõti.

Kui `ssh-copy-id` on olemas:

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub kasutaja@server.example.org
```

Kui `ssh-copy-id` puudub, kuva avalik võti:

```
cat ~/.ssh/id_ed25519.pub
```

macOS-is saad selle lõikelauale panna nii:

```
pbcopy < ~/.ssh/id_ed25519.pub
```

Seejärel lisa see avalik rida serveris faili `~/.ssh/authorized_keys`. Serveris peavad õigused olema tavaliselt:

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Failide kopeerimine sama alias'ega

Kui `~/.ssh/config` alias töötab `ssh`-iga, töötab see sageli ka kopeerimisel.

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

Korduva töö puhul eelista `rsync`-i ja proovi enne:

```
rsync -avn kaust/ opik-server:/tmp/kaust/
```

Veaotsing

Kõige kasulikum esimene diagnostikakäsk on:

```
ssh -v opik-server
```

See näitab:

- millist config-kirjet kasutatakse
- millist hosti ja porti proovitakse
- milliseid võtmeid pakutakse
- kas viga on ühenduses, serveri sõrmejäljes või autentimises

Kui väljund on liiga pikk, otsi sealt ridu `Offering public key`, `Authentications that can continue` ja `Permission denied`.

Mis jääb hilisemaks

SSH-l on ka võimsamad töövõtted:

- port forwarding: `ssh -L ...`
- agent forwarding: `ssh -A ...`
- hüppemasinad: `ProxyJump`
- püsivad ühendused: `ControlMaster`

Need on kasulikud, aga tulevad pärast seda, kui tavaline sisselogimine, võtmed ja `~/.ssh/config` alias'ed on selged.

Minitest

1. Vaata, kas sul on `~/.ssh` kataloog ja millised võtmefailid seal on.
2. Selgita, kumb on privaatvõti ja kumb avalik võti.
3. Kontrolli, kas `ssh-agent` hoiab praegu mõnd võtit.
4. Kirjuta näidis `Host opik-server` plokk `~/.ssh/config` jaoks.
5. Selgita, mida teeb macOS-is `ssh-add --apple-use-keychain`.
6. Kirjuta ühe lausega, miks `UseKeychain` ei tähenda, et võti oleks serverisse lisatud.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: tee SSH korduvkasutatavaks: kontrolli olemasolevaid võtmeid, lisa avalik võti serverisse, kasuta ~/.ssh/config alias't ja macOS-is Keychaini

Põhirada

- `ssh kasutaja@server.example.org` — proovi tavalist sisselogimist
- `ls -la ~/.ssh` — vaata olemasolevaid võtmefaile
- `ssh-add -l` — vaata agendis olevaid võtmeid
- `ssh-keygen -t ed25519 -C 'kasutaja@arvuti'` — loo võti ainult vajadusel
- `ssh-add --apple-use-keychain ~/.ssh/id_ed25519` — macOS: salvesta paroolifraas Keychaini
- `ssh-copy-id -i ~/.ssh/id_ed25519.pub kasutaja@server.example.org` — lisa avalik võti serverisse
- `ssh opik-server` — kasuta config-alias't
- `ssh -v opik-server` — diagnostika

Olulised failid ja mõisted

- `id_ed25519` — privaatvõti; ära jaga
- `id_ed25519.pub` — avalik võti serverisse
- `~/.ssh/config` — ühenduse alias
- `~/.ssh/authorized_keys` — lubatud võtmed
- `known_hosts` — serverite sõrmejäljed
- `AddKeysToAgent` — lisa võtmed agenti

- `UseKeychain` — macOS paroolifraasi hoidmine
- `IdentitiesOnly yes` — kasuta nimetatud võtit

Pane tähele: Alusta sisselogimisest ja võtmetest; port forwarding nagu ssh -L … on järgmine, mitte esimene samm.

Edasi: Järgmine loomulik samm: Veebist sisu toomine ja tekstivaade: curl, wget, lynx.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Veebist sisu toomine ja tekstivaade: curl, wget, lynx

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.

Kiirülevaade

Eesmärk on veebist sisu tuua kontrollitult: alusta väikesest proovist ja vaata enne, mida server vastab.

Käsk	Milleks	Mida tavaliselt näed
`curl -I URL`	küsi ainult päised	HTTP staatus ja päised
`curl URL`	tee üks päring	vastuse sisu terminalis
`wget URL`	laadi fail alla	edenemine ja faili nimi
`lynx -dump URL`	vaata HTML-i tekstina	lihtsustatud tekstivaade

Päiste vaatamine aitab enne suuremat tõmmet aru saada, kas aadress töötab ja kas server suunab sind mujale.

Tüüpilised algaja vead

- minnakse kohe suure crawl'i peale, enne kui üks URL on kontrollitud
- aetakse segi "too üks vastus" ja "laadi fail alla"
- unustatakse, et mõistlik on eelistada API-t või andmefaili HTML-i kraapimisele

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.

Vali ainult oma süsteemile sobiv paigalduskäsk:

Debianis või Ubuntus:

```
sudo apt install wget lynx
```

Fedoras:

```
sudo dnf install wget lynx
```

macOS-is Homebrew kaudu:

```
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`.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: vali curl ühe vastuse või päiste jaoks, wget allalaadimise jaoks ja lynx HTML-i rahulikuks tekstivaateks

Põhikujud

- `curl -I https://example.com/` — kontrolli URL-i
- `curl -L -o leht.html https://example.com/` — salvesta HTML
- `wget -O leht.html https://example.com/` — laadi faili
- `curl -sL https://example.com/ | lynx -stdin -dump` — loe teksti-na
- `lynx -dump -listonly https://example.com/` — kogu lingid

Olulisemad lipud, märgid ja kiirnupud

- `curl -I` — vaata päiseid
- `curl -L` — järgi suunamist
- `curl -O` — salvesta nimega
- `wget -c` — jätka tõmmet
- `lynx -dump` — tekstvaade
- `lynx -listonly` — ainult lingid

Pane tähele: Alusta alati ühest URL-ist; suur crawl ja automaatne kraapimine olgu järgmine, mitte esimene samm.

Edasi: Järgmine loomulik samm: Arhiivid ja pakkimine.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Arhiivid ja pakkimine

Loogika

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

Kiirülevaade

Erista kaht sammu: failide koondamine üheks arhiiviks ja selle arhiivi tihendamine väiksemaks.

Käsk või formaat	Milleks	Mida tavaliselt näed
`tar`	koonda failid arhiiviks	edukal loomisel võib olla vaikne
`gzip`, `xz`	tihenda arhiiv väiksemaks	tekib tihendatud fail
`.tar.gz`	`tar` + `gzip` koos	üks kokkupakitud arhiiv
`zip`	arhiiv ja tihendus ühes vormingus	tekib `.zip` fail
`tar -tf`	vaata arhiivi sisu	failide loend
`tar -xf`, `unzip`	paki lahti	`tar` sageli vaikne, `unzip` näitab faile

Tüüpilised algaja vead

- arvatakse, et `.tar` tähendab juba tihendatud faili
- pakitakse arhiiv lahti vales kataloogis
- ei vaadata enne lahtipakkimist, mis arhiivi sees on

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, lehitse sisuloendit:

```
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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: arhiiv koondab faili- või kaustapuu üheks tervikuks ja pakkimine teeb selle terviku väiksemaks või lihtsamini jagatavaks

Põhikujud

- `tar -cf proov.tar kaust/` — loo arhiiv
- `tar -tf proov.tar` — vaata sisu
- `tar -czf proov.tar.gz kaust/` — loo gzip-arhiiv
- `tar -xzf proov.tar.gz -C siht/` — paki lahti sihtkohta
- `zip -r proov.zip kaust/` — loo zip

- `unzip -l proov.zip` — vaata zip sisu

Olulisemad lipud, märgid ja kiirnupud

- `tar -cf` — loo arhiiv
- `tar -tf` — vaata sisu
- `tar -czf` — gzip-iga kokku
- `tar -xzf` — paki gzip lahti
- `zip -r` — zip kaust
- `unzip -l` — vaata zip sisu

Pane tähele: Enne lahtipakkimist vaata võimalusel arhiivi sisu; see aitab näha, kas sees on oodatud juurkaust ja failipuud.

Edasi: Järgmine loomulik samm: Tervete kataloogipuude haldus ja jagamine.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

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.

Kiirülevaade

Eesmärk on valida tööriist terve projektipuu jaoks: koopia, sünkroon, arhiiv või muudatusajalugu.

Tööriist	Milleks	Mida tavaliselt näed
`cp -R`	lihtne kohalik koopia	edukal juhul sageli vaikne
`rsync`	võrdle ja kopeeri erinevused	`-av` näitab ülekantud faile

Tööriist	Milleks	Mida tavaliselt näed
`scp -r`	saada puu üle SSH	ülekande edenemine
`tar`, `zip`	tee jagatav pakend	arhiivifail; `tar -tf` näitab sisu
Git	jälgi muutusi ajas	diffid, commit'id ja ajalugu

Tüüpilised algaja vead

- kasutatakse sama töö jaoks juhuslikult eri tööriistu, mõtlemata eesmärgile
- arvatakse, et `scp -r` ja `rsync` on alati sisuliselt sama
- ei kontrollita, kas koopia sisaldab ka peidetud faile või õigeid metaandmeid

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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: terve projekti või kaustapuu puhul vali tööriist selle järgi, kas tahad koopiat, korduvat sünkroonimist, arhiivi või versiooniajalugu

Põhikujud

- `cp -R projekt projekt-koopia` — tee kohalik koopia
- `rsync -avn projekt/ projekt-varu/` — kontrolli enne
- `rsync -av projekt/ kasutaja@server:/srv/projekt/` — sünkrooni serverisse
- `scp -r projekt kasutaja@server:/tmp/` — saada kiirelt
- `tar -czf projekt.tgz projekt/` — tee hetkearhiiv
- `git` — jälgi ajalugu

Olulisemad lipud, märgid ja kiirnupud

- `projekt/` — sisu sees
- `projekt` — kaust tervikuna
- `rsync -n` — proovi enne
- `cp -a` — säilita metaandmed

Pane tähele: Kui vajad korduvat sünkroonimist, eelista tavaliselt rsync-i; scp -r sobib pigem üheks kiireks ülekandeks.

Edasi: Järgmine loomulik samm: Protsessid, tööd ja signaalid.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Protsessid, tööd ja signaalid

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

Kiirülevaade

Eesmärk on saada kontroll olukorra üle, kus midagi jookseb kaua, kasutab palju ressursse või peab liikuma taustale.

Käsk või mõiste	Milleks	Mida tavaliselt näed
`ps`	vaata protsesside hetkeolukorda	protsesside tabel
`top`, `htop`	jälgi reaalajas	pidevalt uuenev vaade
`kill PID`	saada protsessile signaal	edukal juhul sageli vaikne
`jobs`	näita selle shelli taustatöid	töö numbrid nagu `%1`
`bg`	jätka peatatud tööd taustal	töö jätkub taustal
`fg`	too töö esiplaanile	käsk tuleb terminaliaknasse tagasi
`wait`	oota taustatöö lõppu	lõpeb siis, kui töö lõpeb

PID on süsteemi protsessinumber; `%1` ja `%2` on ainult selle shelli töö numbrid.

Tüüpilised algaja vead

- kasutatakse kohe `kill -9`, kuigi tavalisest `kill`-ist piisaks
- aetakse segi PID ja `jobs`-i töö number
- unustatakse, et taustatöö ei pruugi akna sulgemisel alles jääda
- vaadatakse ainult protsesside nimekirja, ilma et küsitaks, kas töö kuulub üldse samale shellile

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 programmi töö
- `Ctrl-z` peatab programmi 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

Need käsud vastavad kahele tavalisele küsimusele:

- 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`.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: erista protsessi, shelli tööd ja signaali: vaata, mis jookseb, peata vajadusel ja too pikk töö taustale

Põhikujud

- `ps aux | grep python` — otsi protsess
- `ps aux | sort -nrk 3 | head` — CPU sööjad
- `ps aux | sort -nrk 4 | head` — mälusööjad
- `sleep 300 &` — käivita taustal
- `jobs` — vaata töid
- `fg %1` — too töö ette
- `top` — jälgi reaalajas
- `kill` — saada signaal
- `bg` — jätka taustal

Olulisemad lipud, märgid ja kiirnupud

- `Ctrl-c` — katkesta programmi töö
- `Ctrl-z` — peata ajutiselt; fg ette, bg taustale
- `&` — käivita taustal
- `%1` — töö number
- `kill -15` — viisakas lõpp
- `kill -9` — jõuga lõpp

Pane tähele: Kasuta kill -9 alles siis, kui tavalisem kill või kill -15 ei lõpeta protsessi viisakalt.

Edasi: Järgmine loomulik samm: Logid ja teenused.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Logid ja teenused

Loogika

Kui mõni teenus ei tööta, alusta kahest küsimusest:

1. kas teenus üldse töötab
2. mida logid selle kohta ütlevad

Teenuse seisu vaatad tavaliselt `systemctl` abil. Vea põhjuse otsimiseks liigud edasi `journalctl` või logifailide juurde.

Kiirülevaade

Eesmärk on muuta “miski ei tööta” kontrollitavaks küsimuseks: kas teenus töötab ja mida logi ütleb.

Käsk	Milleks	Mida tavaliselt näed
`systemctl status nimi`	küsi teenuse seisu	aktiivsus, käivitus ja viimased teated

Käsk	Milleks	Mida tavaliselt näed
`journalctl -u nimi`	vaata teenuse journali	järjest logiridu
`journalctl -n 50`	vaata viimaseid kirjeid	viimased 50 logirida
`tail -n 50 fail`	vaata logifaili lõppu	faili viimased read
`tail -f fail`	jälgi logi reaalajas	käsk jääb uusi ridu ootama

Logides on sageli kõige kasulikumad sõnad `ERROR`, `WARN`, `failed`, `denied` ja `timeout`.

Tüüpilised algaja vead

- loetakse kohe logi, ilma et teenuse seis üle kontrollitaks
- vaadatakse vale teenuse nime, näiteks `ssh` vs `sshd`
- unustatakse, et `tail -f` lõpetatakse tavaliselt `Ctrl-c`-ga
- otsitakse "üht õiget faili", kuigi süsteem võib kasutada kas journal'it või tavapäraseid logifaile

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 lehitsetavas vaates sirvida:

```
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, vaata viimaseid selle teenuse kirjeid:

```
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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: kui teenus ei tööta, alusta kahest küsimusest: kas teenus on üldse aktiivne ja mida logid selle kohta ütlevad

Põhikujud

- `systemctl status ssh` — vaata seisu
- `journalctl -u ssh -n 50` — loe viimast logi
- `journalctl -u ssh -f` — jälgi teenust
- `tail -n 50 /var/log/syslog` — loe faili lõppu
- `tail -f /var/log/syslog` — jälgi faili
- `tail -n 50 /var/log/syslog | less` — sirvi viimaseid ridu

Olulisemad lipud, märgid ja kiirnupud

- `status` — teenuse seis
- `-u nimi` — üks teenus
- `-n 50` — viimased read
- `-f` — jälgi reaalajas
- `--since today` — ainult tänane

Pane tähele: Alusta teenuse puhul tavaliselt systemctl status-est ja alles siis mine sügavamale journalctl või logifailide juurde.

Edasi: Järgmine loomulik samm: Püsivad terminalisessioonid: tmux ja screen.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Püsivad terminalisessioonid: tmux ja screen

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.

Kiirülevaade

Eesmärk on hoida pikk töö elus ka siis, kui terminaliaken või SSH-ühendus katkeb.

Käsk või mõiste	Milleks	Mida tavaliselt näed
`tmux`, `screen`	loo püsiv sessioon	tavaline shell püsiva kihi sees
detach ehk eraldumine	jäta töö sessiooni edasi käima	naased tavalisse terminali
attach ehk taasühendumine	tule samasse sessiooni tagasi	sama aken ja samad protsessid
`tmux ls`, `screen -ls`	loetle sessioonid	olemasolevate sessioonide nimed

Tüüpilised algaja vead

- aetakse segi taustatöö `&` ja püsiv sessioon
- pannakse aken kinni ilma, et mõistetaks, kas töö jäi sessiooni sisse jooksma
- unustatakse sessioonile nimi anda, kuigi see teeks tagasimineku lihtsamaks

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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: kasuta püsivat sessiooni siis, kui SSH võib katkeda või kui pikk töö peab jätkuma ka pärast akna sulgemist

Põhikujud

- `tmux new -s opik` — loo tmux
- `tmux attach -t opik` — naase tmux-i
- `screen -S opik` — loo screen
- `screen -r opik` — naase screeni
- `nohup pikk-kaik > logi 2>&1 &` — jäta töö käima
- `disown` — seo shellist lahti

Olulisemad lipud, märgid ja kiirnupud

- `Ctrl-b d` — eraldu tmux-ist
- `Ctrl-a d` — eraldu screenist
- `tmux ls` — sessioonide loend
- `screen -ls` — sessioonide loend

Pane tähele: Kui pead valima ühe harjumuse, vali tmux: see jätab alles terve sessiooni, mitte ainult ühe käsu.

Edasi: Järgmine loomulik samm: Graafilised rakendused kaugmasinast.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

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.

Kiirülevaade

Eesmärk on otsustada, kas graafilist programmi on vaja üle võrgu näidata või sobib paremini veebiliides või IDE kaugühendus.

Lahendus	Milleks	Mida tavaliselt näed
`ssh -X`	saada X11-rakenduse aken sinu masinasse	terminal on vaikne, aken võib avaneda eraldi
`ssh -L`	too serveri port oma `localhost` aadressile	SSH-seanss jääb lahti, brauser kasutab kohalikku aadressi
Remote SSH IDE-s	redigeeri oma arvutis, jooksuta serveris	IDE-s serveri failipuu ja terminal
veebiliides	jäta graafika brauseri tööks	HTTP-aadress brauseris

Tüüpilised algaja vead

- proovitakse X11 forwardingut enne, kui tavaline `ssh` töötab
- unustatakse, et `localhost` tähendab pärast port forwardingut sinu enda arvutit, mitte tingimata serverit
- arvatakse, et GUI-edastus on alati parim tee, kuigi veebiliides või Remote SSH on sageli kiirem ja töökindlam
- jäetakse port forwarding liiga laialt avatuks; alguses hoia seos ainult oma masinaga

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`.

Siin tähendavad kaks `localhost`-i eri vaates peaaegu sama asja:

- käsus olev `localhost:8888` on serveri enda vaade teenusele
- brauseris avatav `http://localhost:8888` on sinu arvuti vaade edasisuunatud pordile

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.

Peatüki täisspikker

Tase: **Praktiline kasutus**

Eesmärk: kaugelt graafika kasutamisel eelista lihtsaimat toimivat teed: veebiliides enne X11-edastust, port forwarding enne toorest kaugtöölauda.

Põhikujud

- `ssh -L 8888:localhost:8888 kasutaja@server` — too veebiliides kohale
- `http://localhost:8888` — ava edasi suunatud teenus
- `ssh -X kasutaja@server` — proovi lihtsat X11
- `code --remote ssh-remote+server /tee/projektini` — IDE üle SSH

Valiku rusikareeglid

- `veebiliides` — tavaliselt mugavaim
- `ssh -L` — port edasi
- `ssh -X` — X11 erijuht
- `Remote SSH` — arenduseks parem

Pane tähele: Kui sul on valida, eelista brauserit või Remote SSH-d; X11 forwarding olgu pigem varuplaan, mitte esimene mõte.

Edasi: Järgmine loomulik samm: Teksti otsimine: grep ja sugulased.

Osa PDF: ./spikrid/osa-iii-failid-vork-ja-susteemitoo-spikker.pdf

Teksti otsimine: grep ja sugulased

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.

Kiirülevaade

Eesmärk on suurest tekstihulgast kiiresti õiged read välja valida. `grep` on sageli esimene filter, mitte kogu lahendus.

Käsk või lipp	Milleks	Mida tavaliselt näed
`grep muster fail`	jäta alles sobivad read	ainult vastega read
`grep -v`	jäta alles mittesobivad read	read, kus mustrit ei ole

Käsk või lipp	Milleks	Mida tavaliselt näed
`grep -r`	otsi failipuust	failinimed ja vastega read
`grep -n`	lisa reanumbrid	number rea ees
vasteid pole	tulemus puudub	tühi väljund

Tüüpilised algaja vead

- arvatakse, et `grep` otsib "sõnu", kuigi ta töötab ridade kaupa
- unustatakse tõstutundlikkus ja imestatakse, miks vasteid ei leita
- aetakse segi lihtne tekst ja regulaaravaldis

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

Selles näites ei anna esimene otsing vastet. Järgmine käsk teeb sisendi enne väiketäheliseks ja vaste muutub nähtavaks.

```
kasutaja@mac tmp % wget https://raw.githubusercontent.com/dwyl/english-words/refs/heads/mast
...
'words.txt' saved

kasutaja@mac tmp % cat words.txt | grep -x 'a..y..l.e'

kasutaja@mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a..y..l.e'
abbyville

kasutaja@mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a.t.l'
antal
aptal
```

```
artal
artel
astel
attal
axtel

kasutaja@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

kasutaja@mac tmp % cat words.txt | tr 'A-Z' 'a-z' | grep -x 'a.t.l' | grep -o . | sort | uni
  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

Sama töövoog ühendab mitu tuttavat sammu:

- `grep` otsib mustri järgi
- `tr` muudab teksti kuju
- `sort` ja `uniq -c` koondavad tulemuse statistikaks

Kui eesmärk on ainult mustriotsing, võib toru `cat words.txt | ...` asemel kirjutada lühemalt:

```
tr 'A-Z' 'a-z' < words.txt | grep -x 'a..y..l.e'
```

Pikem kuju sobib siis, kui tahad eraldi näha, kuidas tekst liigub käsust järgmisse.

Edasijõudnule: tagasiviited ja korduv muster

GNU `grep` toetab ka tagasiviiteid. See tähendab, et saad öelda: “otsi midagi, kus seesama eelnevalt leitud tükk kordub uuesti”.

Näide:

```
kasutaja@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

Viimane näide näitab ka `-x` mõju:

- 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.

Tagasiviited sobivad katsetamiseks ja mõneks erijuhtumiks. Igapäevases tekstifiltreerimises eelista võimalusel lihtsamat mustrit, sest tagasiviited võivad olla aeglasemad ja raskemini loetavad.

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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: grep valib sisendist välja ainult need read, mis sobivad mustriga; see on filtritööriist, mitte tekstiredaktor

Põhikujud

- `grep 'kass' loomad.txt` — otsi ühest failist
- `grep -i 'kass' loomad.txt` — otsi tõstuta
- `grep -n 'koer' loomad.txt` — näita reanr
- `grep -r 'TODO' .` — otsi puust
- `grep -F 'https://example.com' fail.txt` — otsi täpset sõnet
- `grep -v '^#' seadistus.conf` — jäta kommentaarid välja

Olulisemad lipud, märgid ja kiirnupud

- `-i` — tõstutundetu
- `-n` — reanumbrid
- `-r` — rekursiivne
- `-v` — jäta vasted välja
- `-F` — täpne sõne
- `-E` — laiendatud regex

Pane tähele: Kui sa ei ole mustris kindel, alusta grep -F või lihtsa sõnega; regulaaravaldise keerukus olgu järgmine samm, mitte esimene.

Edasi: Järgmine loomulik samm: Teksti teisendamine: tr, cut, paste, column, strings.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

Teksti teisendamine: tr, cut, paste, column, strings

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.

Kiirülevaade

Eesmärk on muuta teksti kuju väikeste tööriistadega, kus iga käsk teeb ühe konkreetse sammu.

Käsk	Milleks	Mida tavaliselt näed
`tr`	muuda või kustuta märke	muutunud tekstivoog

Käsk	Milleks	Mida tavaliselt näed
`cut`	võta välja välju või veerge	ainult valitud osad
`paste`	pane read kõrvuti	ühendatud read
`column -t`	joonda tabelilaadselt	loetavamad veerud
`strings`	otsi binaarfailist teksti	loetavad tekstijupid ridadena

Tüüpilised algaja vead

- aetakse segi märkide muutmine ja väljade lõikamine
- kasutatakse `cut`-i keeruka CSV peal, kus lihtne eraldaja ei pruugi piisata
- oodatakse, et `column` muudab faili, kuigi ta ainult vormib väljundi ilusamaks

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

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

Nii muutub käsuväljund kiirelt tabelina loetavaks.

Päris näide: `strings`

`strings` on kasulik 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.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: need väikesed filtrid muudavad tekstivoo kuju kiiresti: märgid, väljad, veerud ja tabelid ilma eraldi skriptita

Põhikujud

- `echo 'tere maailm' | tr '[:lower:]' '[:upper:]'` — muuda suurtäheks
- `echo 'a,b,c' | tr ',' '\n'` — tee ridadeks
- `cut -d ':' -f 1 andmed.txt` — võta esimene väli
- `cut -d ':' -f 1,3 andmed.txt` — võta mitu välja
- `printf 'nimi vanus\nMari 20\n' | column -t` — joonda tabel
- `strings /bin/ls | head -n 20` — loe binaarist tekst
- `paste` — kleebi veerge

Olulisemad lipud, märgid ja kiirnupud

- `[:lower:]` — väiketähed
- `[:upper:]` — suurtähed
- `cut -d` — vali eraldaja
- `cut -f` — vali väljad
- `column -t` — joonda tabel
- `strings` — tekstijupid

Pane tähele: Kui väljund muutub imelikuks, kontrolli kõigepealt eraldajat ja sisendi kuju; need tööriistad on väikesed, aga väga sõna-sõnalt loetavad.

Edasi: Järgmine loomulik samm: Vood ja tabelid: sort, uniq, wc, pr, join.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

Vood ja tabelid: sort, uniq, wc, pr, join

Loogika

Kui sul on palju ridu, aitab see peatükk neist kokkuvõtte teha. Tüüpiline töövoog on: sorteeri, koonda, loenda.

Kiirülevaade

Eesmärk on muuta suur hulk ridu kokkuvõtteks: sortida, loendada, rühmitada ja vajadusel tabelina siduda.

Käsk	Milleks	Mida tavaliselt näed
`sort`	sea read järjekorda	samad read uues järjekorras
`uniq`	koonda kõrvuti duplikaadid	korduvad read kaovad või saavad loenduri
`wc`	loenda ridu, sõnu või märke	lühike arvuline kokkuvõte
`join`	ühenda kaks faili ühise välja järgi	seotud read ühises väljundis
`pr`	vormi veergudesse või printimiseks	lehekülje- või veerupaigutus

Tüüpilised algaja vead

- kasutatakse `uniq`-i sortimata sisendi peal ja oodatakse kogu duplikaatide kadumist
- aetakse segi ridade loendamine ja sõnade loendamine
- unustatakse, et `join` eeldab tavaliselt sobivalt ette valmistatud sisendit

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

`pr` ei ole tänapäeval kõige sagedasem tööriist, kuid sobib kiireks veergudesse vormindamiseks.

Näiteks saab nummerdatud loendi panna 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.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: tüüpiline töövoog on: sorteeri read, koonda kordused ja loe tulemused kokku

Põhikujud

- `sort viljad.txt` — sordi read
- `sort viljad.txt | uniq` — eemalda kordused

- `sort viljad.txt | uniq -c` — loe kordused
- `wc -l data/sample-text.txt` — loe read
- `wc -w data/sample-text.txt` — loe sõnad
- `join nimed.txt linnad.txt` — ühenda võtme järgi
- `pr` — jaga veergudeks

Olulisemad lipud, märgid ja kiirnupud

- `sort -n` — numbrid
- `sort -r` — tagurpidi
- `uniq -c` — loenda kordused
- `wc -l` — ridade arv
- `wc -w` — sõnade arv
- `pr -2` — kaks veergu

Pane tähele: uniq töötab loogiliselt alles pärast sort-i; vastasel juhul loendab ta ainult järjestikuseid kordusi.

Edasi: Järgmine loomulik samm: sed, awk ja perl praktiliselt.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

sed, awk ja perl praktiliselt

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

Kiirülevaade

Eesmärk on minna edasi siis, kui `grep`, `cut` ja `tr` enam ei piisa: muuta ridu, arvutada väljade kaupa ja kirjutada lühike tekstiloogika.

Käsk	Milleks	Mida tavaliselt näed
`sed`	muuda ridu mustri järgi	ümber kujundatud tekstivoog
`awk`	töötle välju ja veerge	valitud või arvutatud väljad

Käsk	Milleks	Mida tavaliselt näed
`perl`	tee keerukam mustri- ja tekstiloogika	skripti või ühekordse käsu väljund
faili muutmine	nõuab eraldi kuju	sisendfail ei muutu vaikimisi
süntaksiviga	käsk ei käivitu ootuspäraselt	mustri- või süntaksiveateade

Tüüpilised algaja vead

- oodatakse, et `sed` või `awk` muudab faili kohe kohapeal
- aetakse segi "esimene vaste real" ja "kõik vasted real"
- kirjutatakse one-liner, mille loogika on endalegi liiga segane

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:

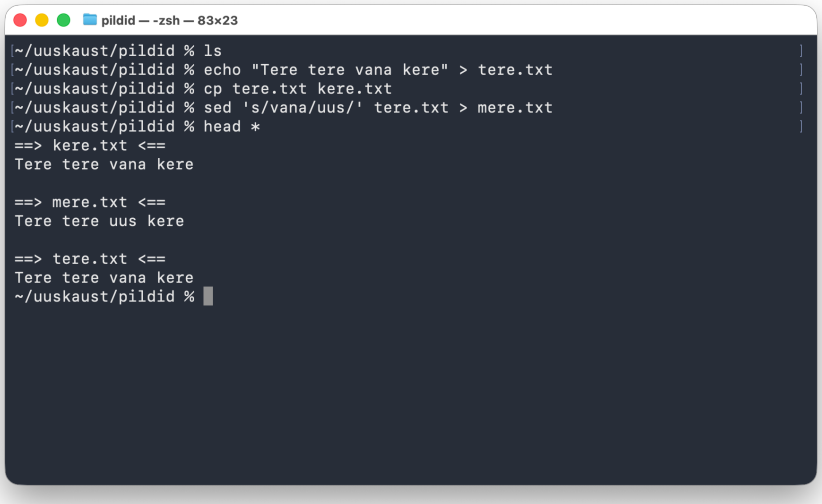
1. `echo "Tere tere vana kere" > tere.txt` loob algse faili
2. `cp tere.txt kere.txt` teeb samast sisust koopia
3. `sed 's/vana/uus/' tere.txt > mere.txt` loeb faili `tere.txt`, asendab esimese vaste ja kirjutab tulemuse uude faili
4. `head *` näitab kolme faili algust kõrvuti

Oluline tähelepanek on see, et `sed` ei muuda siin algset faili kohapeal. Tulemuse saab uude faili suunata märgiga `>`.

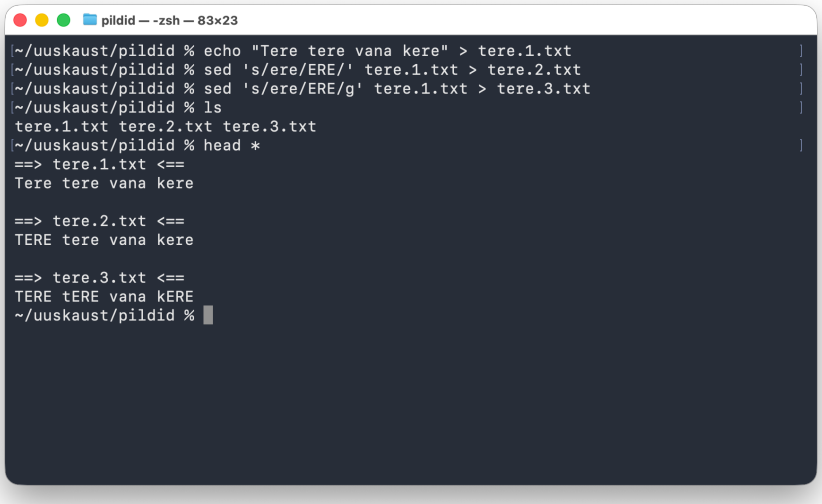
Üks asendus või kõik asendused

`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:



Joonis 7: 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.



Joonis 8: 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.

1. `sed 's/ere/ERE/' tere.1.txt > tere.2.txt` muudab ainult rea esimese vaste
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.

Edasijõudnule: 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.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: kasuta sed lihtsaks asenduseks, awk väljade jaoks ja perl siis, kui vajad tugevamat regulaaravaldiste loogikat

Põhikujud

- `echo 'kass koer kass' | sed 's/kass/rebane/'` — asenda esimene

- `echo 'kass koer kass' | sed 's/kass/rebane/g'` — asenda kõik
- `printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1}'` — võta esimene väli
- `printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1, $2}'` — võta kaks välja
- `echo 'kass koer kass' | perl -pe 's/kass/rebane/g'` — asenda Perliga
- `printf 'Mari\nJaan\n' | perl -ne 'print if /Ja/'` — filtreeri Perliga

Olulisemad lipud, märgid ja kiirnupud

- `s/vana/uus/` — üks asendus
- `/g` — kõik vasted
- `-F:` — koolon väljade vahel
- `$1` — esimene väli
- `$2` — teine väli
- `-pe / -ne` — read läbi

Pane tähele: Kui vajad ainult üht lihtsat asendust, alusta sed-ist; ära hüppa kohe awk või perl juurde enne, kui lihtsam tee on ammendunud.

Edasi: Järgmine loomulik samm: find ja xargs ohutumalt.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

`find` ja `xargs` ohutumalt

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`

Kiirülevaade

Eesmärk on failipuus tegutseda nii, et tühikud ja muud keerulised failinimed tulemust ei lõhuks.

Käsk või kuju	Milleks	Mida tavaliselt näed
`find ...`	leia teed tingimuste järgi	leitud failide ja kaustade teed
`xargs`	anna teed järgmisele käsule	järgmise käsu väljund
`-print0`	eralda nimed nullmärgiga	masinloetav voog, mitte tavalised read
`xargs -0`	loe nullmärgiga nimesid	tühikutega nimed jäävad terveks
`-exec ... {} +`	käivita käsk otse `find` sees	järgmise käsu väljund

Vale eraldusviis lõpeb sageli veaga “No such file or directory”, sest failinimi lõigatakse tükkideks.

Tüüpilised algaja vead

- eeldatakse, et kõik failinimed on lihtsad ühe sõnaga nimed
- kasutatakse `xargs`-it pimesi seal, kus `-exec` oleks lihtsam
- testimata käsk pannakse kohe muutma või kustutama

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

Kui `find` ja `xargs` on tuttavad, on loomulik järgmine samm kasutada neid koos tekstiotsingu või shelliskriptidega.

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`.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: otsi failid find-iga ja anna need ohutult edasi järgmisele käsule, isegi siis, kui nimedes on tühikuid

Põhikujud

- `find . -type f -name '*.txt'` — otsi failid
- `find . -type f -name '*.txt' -print0 | xargs -0 wc -l` — loe read ohutult
- `find . -type f -name '*.log' -exec ls -lh {} +` — käivita otse
- `find data -type f -name '*.txt' -print0 | while IFS= read -r -d '' fail; do head -n 1 "$fail"; done` — töötle ükshaaval

Olulisemad lipud, märgid ja kiirnupud

- `-type f` — ainult failid
- `-name '*.txt'` — nime järgi
- `-print0` — ohutu eraldus
- `xargs -0` — loe ohutult
- `-exec {} +` — käivita otse
- `read -r -d ''` — loe nullmärgini

Pane tähele: Kui failinimes võib olla tühikuid või reavahetusi, eelista paari -print0 ja xargs -0.

Edasi: Järgmine loomulik samm: Esimene shelliskript.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

Esimene shelliskript

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

Kiirülevaade

Eesmärk on panna korduvad käsureasammud tekstifaili, mida saab uuesti käivitada ja jagada.

Mõiste või käsk	Milleks	Mida tavaliselt näed
shebang	vali skripti tõlgend	skript käivitub õige shelli või keelega

Mõiste või käsk	Milleks	Mida tavaliselt näed
`chmod +x fail`	luba faili käivitada	edukal juhul sageli vaikne
`$1`	esimene argument	skript kasutab käsurealt antud väärtust
`"$@"`	kõik argumendid turvaliselt	iga argument jääb eraldi
`exit 0`, `exit 1`	teata õnnestumisest või veast	järgmine käsk saab tulemuskoodi lugeda

Vigase süntaksi, puuduva tõlgendi või vale shebang'i korral tuleb veateade enne, kui töö päriselt õnnestub.

Tüüpilised algaja vead

- unustatakse jutumärgid argumentide ümber
- käivitatakse skript vale shelliga
- eeldatakse, et `chmod +x` üksi ütleb juba, millega faili tõlgendada

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.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: esimese shelliskripti tuum on viis asja: shebang, käivitatavus, argumendid, lihtne tingimus ja lõpetuskood

Põhikujud

- `chmod +x tervita.sh` — tee käivitatavaks
- `./tervita.sh Mari Jaan` — käivita argumentidega
- `./tervita.sh` — näe veajuhtumit
- `echo $?` — vaata lõpetuskoodi
- `bash skript.sh` — käivita Bashiga
- `sh skript.sh` — käivita sh-ga
- `./skript.sh` — kasuta shebang'i

Olulisemad lipud, märgid ja kiirnupud

- `#!/usr/bin/env bash` — vali Bash
- `$1` — esimene argument
- `"$@"` — kõik argumendid
- `if ... fi` — tingimus
- `for ... done` — kordus
- `exit 0` / `exit 1` — edu või viga

Pane tähele: Kõige tähtsam vahe on see: ./skript.sh kasutab shebang-rida, bash skript.sh valib tõlgendaja käsurealt.

Edasi: Järgmine loomulik samm: cron ja ajastatud tööd.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

`cron` ja ajastatud tööd

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

Kiirülevaade

Eesmärk on panna lihtne töö kindlal ajal käima ja jätta maha logi, millest saab hiljem aru, mis juhtus.

Käsk või mõiste	Milleks	Mida tavaliselt näed
`crontab -l`	näita olemasolevaid ajastusi	ridade loend või teade, et ajastusi pole
`crontab -e`	muuda ajastusi	avaneb redaktor
croni ajamuster	käivita käsk kindlal ajal	terminalis ei ilmu tavaliselt midagi
suunamine logifaili	salvesta tulemus ja vead	logifailist näed hiljem väljundit

Cron jookseb teistsuguses keskkonnas kui sinu interaktiivne shell, seega kasuta vajadusel täisteid ja logimist.

Tüüpilised algaja vead

- kasutatakse suhtelisi teid, mis cronis ei tööta ootuspäraselt
- unustatakse, et cronil võib olla teistsugune `PATH`
- eeldatakse, et vea korral ilmub midagi automaatselt ekraanile

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.

Peatüki täisspikker

Tase: **Töövood**

Eesmärk: ajastatud töö tähendab, et käsk jookseb kindlal ajal ilma sinu avatud terminalita; tähtsad on ajaväljad, täisrajad ja logifail

Põhikujud

- `crontab -l` — vaata olemasolevat
- `crontab naide.cron` — paigalda fail
- `/bin/date >> "$HOME"/cron-naide.log 2>&1` — testi käsitsi
- `tail -n 5 "$HOME"/cron-naide.log` — vaata logi

Olulisemad lipud, märgid ja kiirnupud

- `*/15 * * * *` — iga 15 min
- `>>` — lisa logisse
- `2>&1` — vead samasse
- `$HOME` — kasutaja kodu

- `/bin/date` — täisrada

Pane tähele: Kui käsk ei tööta täpselt samal kujul käsitsi, ei tööta see tavaliselt ka cronis.

Edasi: Järgmine loomulik samm: Git, GitHub ja töövoog.

Osa PDF: ./spikrid/osa-iv-tekst-otsing-ja-automatiseerimine-spikker.pdf

Git, GitHub ja töövoog

Loogika

Git ei ole lihtsalt viis faile GitHubi saata. Git hoiab projekti muutuste ajalugu.

Kõige tähtsam töövoog on:

1. vaata, mis seisus repo on
2. vaata, mis täpselt muutus
3. vali järgmise commit'i sisu
4. salvesta loogiline muutus commit'ina
5. sünkrooni vajadusel kaugrepoga

Git ja GitHub ei ole sama asi:

- `Git` on versioonihaldus sinu masinas
- `GitHub` on teenus, kus repot jagada, arutada ja üle vaadata

Kiirülevaade

Käsk	Milleks	Mida tavaliselt näed
`git status`	vaata repo seisu	muutunud, stage'is ja jälgimata failid
`git diff`	vaata tööpuu muutusi	`-` ja `+` read jälgitud failides
`git add fail`	pane fail stage'i	edukal juhul sageli vaikne
`git diff --cached`	vaata stage'i sisu	järgmisse commit'i minev diff
`git commit -m '...'`	salvesta commit	commit'i lühikood ja kokkuvõte
`git log --oneline`	vaata ajalugu	commit'id ühe rea kaupa
`git restore fail`	viska tööpuu muudatus ära	fail läheb tagasi viimase commit'i seisu
`git restore --staged fail`	võta fail stage'ist maha	sisu jääb alles, stage muutub
`git switch -c haru`	loo ja ava uus haru	liigud uuele harule

Käsk	Milleks	Mida tavaliselt näed
`git pull --ff-only`	uuenda puhas haru serverist	fast-forward või selge keeldumine
`git push -u origin haru`	saada haru GitHubi	üleslaadimise kokkuvõte

Tüüpilised algaja vead

- tehakse `git add` enne, kui muudatus on üle vaadatud
- arvatakse, et `git diff` näitab ka täiesti uusi ehk jälgimata faile
- aetakse segi tööpuu, stage ja commit
- tehakse `git pull` määrdunud tööpuuga
- töötatakse otse `main` harus, kuigi muudatus võiks olla eraldi harus
- kasutatakse `git push --force` ilma aru saamata, kelle ajalugu see muudab

Mis on versioonihaldus

Versioonihaldus tähendab, et projekti muutused talletatakse ajalooks.

Praktiliselt annab see neli võimet:

- näha, mis muutus
- salvestada loogilisi vaheetappe
- minna vajadusel eelmise seisu juurde tagasi
- jagada sama projekti teistega

Ilma selle mõtteta jäävad Giti käsud lihtsalt mehaaniliseks loeteluks.

Kolm kohta: tööpuu, stage ja commit

Giti õppimine läheb palju lihtsamaks, kui eristad kolme kohta.

Koht	Tähendus	Kontrollkäsk
tööpuu	sinu failid praegu kettal	`git status`, `git diff`
stage	järgmise commit'i ettevalmistus	`git diff --cached`
commit	salvestatud loogiline muutus	`git log --oneline`

Tavaline rütm on:

```
git status
git diff
git add fail.txt
git diff --cached
git commit -m 'Selge sõnum'
```

`git add` ei tähenda veel “saada GitHubi”. See tähendab ainult: “see muudatus läheb järgmisse commit’i”.

Esimene harjutusrepo

Harjutamiseks tee eraldi ajutine repo. Nii ei puutu sa päris projekti.

```
mkdir -p ~/tmp/git-naide
cd ~/tmp/git-naide
git init
printf 'esimene rida\n' > naide.txt
git status
git add naide.txt
git diff --cached
git commit -m 'Lisa naidefail'
git log --oneline
```

Pane tähele: uus fail `naide.txt` on alguses jälgimata. Seetõttu ei näita tavaline `git diff` veel selle sisu. Pärast `git add naide.txt` näitab `git diff --cached`, mis läheb esimesse commit’i.

Kui `git commit` kaebab autori identiteedi üle, seadista nimi ja e-post:

```
git config --global user.name "Eesnimi Perenimi"
git config --global user.email "nimi@example.com"
```

Seejärel proovi `git commit` uuesti.

Muudatuse ülevaatamine

Kui fail on juba Git-is jälgimisel, näitab `git diff` tööpuu muutust.

```
printf 'teine rida\n' >> naide.txt
git status
git diff
```

Diffi lugemise algreegel:

- `-` tähendab eemaldatud rida
- `+` tähendab lisatud rida
- ülejäänud read on ümbrus ehk kontekst

Enne commit’i kontrolli ka stage’i:

```
git add naide.txt
git diff --cached
git commit -m 'Lisa teine rida'
```

Hea harjumus:

- enne `git add`: `git diff`
- enne `git commit`: `git diff --cached`

- enne pull request'i: vaata kogu muudatus veel kord üle

Igapäevane põhivoog

Kui repo on juba olemas ja töötad GitHubiga, on rahulik põhivoog selline:

```
git status
git switch main
git pull --ff-only
git switch -c parandus
```

Kui sinu repo kasutab põhireana nime `master`, kasuta nendes näidetes `main` asemel `master`.

Seejärel tee muudatused. Enne commit'i:

```
git status
git diff
git add fail1 fail2
git diff --cached
git commit -m 'Paranda näited Git peatükis'
git push -u origin parandus
```

Miks just nii:

- `git status` näitab, kas tööpuu on puhas
- `git pull --ff-only` uuendab põhirea ainult siis, kui seda saab teha sirgelt
- `git switch -c parandus` hoiab muudatuse eraldi harus
- `git diff` ja `git diff --cached` vähendavad juhusliku commit'i riski

Kui `git pull --ff-only` keeldub, ära lisa kohe keerulisemaid lippe. Tee esmalt `git status` ja vaata, kas sul on kohalikke commit'e või pooleliolevaid muudatusi.

Haru ehk branch

`Haru` ehk `branch` on eraldi töölõng sama projekti sees.

Tüüpiline mõtteviis:

- `main` on põhirida
- väike parandus tehakse eraldi harus
- hiljem ühendatakse valmis haru pull request'i kaudu põhireaga

Põhikäsud:

```
git branch
git switch -c parandused-logides
git switch main
git branch -d parandused-logides
```

Siin:

- `git branch` näitab harusid; tärn näitab praegust haru
- `git switch -c nimi` loob uue haru ja liigub sinna
- `git switch main` liigub olemasolevale harule
- `git branch -d nimi` kustutab kohaliku haru, kui see on ühendatud

Vanemates juhendites näed sageli kuju:

```
git checkout -b parandus
```

See tähendab sama, mis `git switch -c parandus`, aga `switch` on algajale selgem: see ütleb otse, et vahetad haru.

Kaugrepo: `clone`, `fetch`, `pull`, `push`

Kaugrepo on sama repo serveris, näiteks GitHubis. Vaikimisi nimi on sageli `origin`.

Kui sul ei ole repot veel kohalikus masinas:

```
git clone git@github.com:kasutaja/projekt.git
cd projekt
git status
```

See kuju kasutab SSH-d. Kui SSH võtmed ei ole veel seadistatud, pakub GitHub sama repo jaoks ka `https://...` algusega klooniaadressi.

Kui tahad ainult teada saada, mis serveris muutus, kasuta:

```
git fetch origin
git log --oneline --graph --decorate --all -n 20
```

`git fetch` uuendab infot kaugrepo kohta, aga ei muuda sinu praegust tööharu.

Kui tahad puhta kohaliku haru serveriga samasse seisu tuua:

```
git pull --ff-only
```

Kui sul on oma haru valmis ja tahad selle GitHubi saata:

```
git push -u origin parandus
```

Kui tööpuu pole puhas

Enne haru vahetamist, pull'i või rebase'i kontrolli:

```
git status
```

Kui tahad tööpuu muudatuse ära visata:

```
git restore fail.txt
```

Kui tahad faili stage'ist maha võtta, aga sisu alles jätta:

```
git restore --staged fail.txt
```

Kui tahad poolelioleva töö korraks kõrvale panna:

```
git stash push -m 'pooleli enne pulli'
git pull --ff-only
git stash pop
```

`stash` on ajutine sahtel, mitte pikaajaline hoiukoht. Kui töö on sisuline, on commit tavaliselt parem kui nädalateks stashi jätmine.

`.gitignore`

Kõiki faile ei tasu Git-i panna.

Tüüpiliselt jäetakse välja:

- virtuaalkeskkonnad, näiteks `.venv/`
- vahemälud, näiteks `__pycache__/`
- buildi väljundid, näiteks `dist/`
- ajutised logid, näiteks `*.log`

Lihtne näide:

```
cat > .gitignore <<'EOF'
.venv/
__pycache__/
dist/
*.log
EOF
git status
```

Kui fail on juba Git-i lisatud, siis ainult `.gitignore` ei eemalda teda automaatselt repost. `.gitignore` takistab eelkõige uute sobivate failide juhuslikku lisamist.

`merge` ja `rebase`

`merge` ja `rebase` panevad kaks ajalugu uuesti kokku, aga teevad seda eri moodi.

Tegevus	Mõte	Millal kasutada
`merge`	säilitab mõlema haru ajaloo	valmis haru ühendamisel
`rebase`	tõstab sinu commit'id teise haru lõppu	oma parandusharu korrastamisel

Näide: oled harus `parandus` ja tahad võtta sisse värske `main` haru:

```
git fetch origin
git switch parandus
git rebase origin/main
```

Näide: tahad valmis haru `main` sisse ühendada:

```
git switch main
git pull --ff-only
git merge parandus
```

Meeskonnareegel:

- oma isiklikku parandusharu võib enne pull request'i sageli rebase'ida
- ära rebase'i haru, mille peale teised juba oma tööd ehitavad, kui see ei ole kokku lepitud
- ühise `main` haru ajalugu tuleb käsitleda eriti ettevaatlikult

Konfliktid

Konflikt tekib siis, kui Git ei oska kahte muudatust ise kokku panna.

Failis võib olla selline koht:

```
    <<<<<<< HEAD
minu praegune tekst
    =======
teisest harust tulnud tekst
    >>>>>>> origin/main
```

See tähendab:

- `<<<<<<< HEAD` all on sinu praeguse haru versioon
- `=======` eraldab kaks varianti
- `>>>>>>> origin/main` all on teise haru versioon

Päris konfliktis algavad need märgid tavaliselt rea algusest. Siin on nad taandatud, et õpiku enda Git ei peaks näidet lahendamata konfliktiks.

Lahendamine:

1. ava fail redaktoris
2. tee valmis õige lõpptekst
3. kustuta konfliktimärgid
4. kontrolli faili sisu
5. lisa fail stage'i

Näiteks:

```
git status
nano fail.txt
git add fail.txt
```

Kui konflikt tekkis merge'i ajal:

```
git commit
```

Kui konflikt tekkis rebase'i ajal:

```
git rebase --continue
```

Kui läksid valesse suunda, katkesta ainult käimasolev tegevus:

```
git merge --abort
git rebase --abort
```

Kasuta neist ainult seda käsku, mis vastab parasjagu käimasolevale tegevusele.

Mida mitte teha pimesi

Need käsud võivad olla õiged, aga neid ei tasu kasutada paanikas:

- `git reset --hard`
- `git clean -fd`
- `git push --force`
- `git branch -D haru`

Rahulikum kontrolljärjekord on:

1. `git status`
2. `git diff`
3. `git diff --cached`
4. vajadusel küsi abi või tee koopia enne hävitavat käsku

Kui force-push on päriselt vajalik, peaks see olema meeskonnas kokku lepitud ja tavaliselt omaenda parandusharu peal, mitte ühisel `main` harus.

GitHub: issue ja pull request

GitHubis seotakse muudatus tavaliselt arutelukoha või ülesandega.

Põhimõisted:

- `issue` on ülesanne, bugi või aruteluteema
- `pull request` on konkreetne muudatusettepanek
- `review` on teiste tagasiside pull request'i kohta
- `checklist` on Markdowni märkeruutudega loend

Lihtne issue kirjeldus:

```
## Probleem

Peatükis 09 on `cat` ja `less` vahe liiga uduselt seletatud.

## Oodatud tulemus

Lugeja saab aru, millal kasutada `cat` ja millal `less`.
```

```
## Kontroll

- [ ] näide on peatükis parandatud
- [ ] build läheb läbi
```

Hea pull request vastab kolmele küsimusele:

- mis probleem lahendati
- kuidas seda lahendati
- kuidas kontrolliti, et lahendus töötab

Tüüpiline GitHubi töövoog:

1. vali issue või kirjuta probleem lühidalt lahti
2. loo selle jaoks haru, näiteks `fix-cat-less`
3. tee väike loogiline muudatus
4. commit'i selge sõnumiga
5. push'i haru GitHubi
6. ava pull request
7. vasta review kommentaaridele uute commit'idega
8. pärast ühendamist kustuta tööharu

Commit'i või pull request'i kirjelduses saab issue'le viidata:

```
Fixes #12
Closes #12
Refs #12
```

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 seotud peatükiga Kauglogimine ja SSH.

Väike lõppharjutus

Tee ajutises repos üks väike töövoog läbi:

```
cd ~/tmp/git-naide
git switch -c kolmas-rida
printf 'kolmas rida\n' >> naide.txt
git diff
git add naide.txt
git diff --cached
git commit -m 'Lisa kolmas rida'
git log --oneline --graph --decorate -n 5
```

Kui tahad harjutusrepo hiljem lihtsalt ära unustada, jäta see `~/tmp` alla või kustuta siis, kui oled kindel, et seal pole vajalikku tööd.

Minitest

1. Selgita, mis vahe on tööpuul, stage'il ja commit'il.
2. Miks ei näita `git diff` uue jälgimata faili sisu?
3. Millal vaatad `git diff --cached`?
4. Miks on `git switch -c parandus` algajale selgem kui `git checkout -b parandus`?
5. Mis vahe on `git fetch` ja `git pull --ff-only` vahel?
6. Millal kasutaksid `git restore --staged fail.txt`?
7. Kirjuta pull request'i lühikirjeldus kujul: probleem, lahendus, kontroll.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: Git hoiab muutuste ajalugu; erista tööpuud, stage'i ja commit'i ning vaata diff enne üle.

Põhikujud

- `git status` — vaata seis
- `git diff` — muutused tööpuus
- `git diff --cached` — muutused stage'is
- `git add fail.txt` — pane stage'i
- `git commit -m '...'` — tee commit
- `git switch -c parandus` — loo haru
- `git pull --ff-only` — uuenda puhtalt
- `git push -u origin parandus` — saada haru
- `git restore --staged fail.txt` — võta stage'ist

Olulisemad lipud, märgid ja kiirnupud

- `HEAD` — praegune tipp
- `main` — põhirida
- `origin` — kaugrepo
- `stage` — järgmine commit

Pane tähele: Uus jälgimata fail ei ilmu tavalisse git diff vaatesse; pärast git add kontrolli esimese commit'i sisu käsuga git diff –cached.

Edasi: Järgmine loomulik samm: Pythoni venv ja eraldatud keskkonnad.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

Pythoni venv ja eraldatud keskkonnad

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

Kiirülevaade

Eesmärk on teada, millise Pythoniga projekt jookseb ja kuhu paketid paigaldatakse.

Käsk või samm	Milleks	Mida tavaliselt näed
`python3 -m venv .venv`	loo eraldatud keskkond	uus `.venv` kataloog
aktiveerimine	suuna `python` ja `pip` sellesse keskkonda	promptis sageli `(.venv)`
`pip install ...`	paigalda pakett aktiivsesse keskkonda	paigalduslogi
`command -v python`	kontrolli interpreteri teed	tee `.venv` alla
`python --version`	kontrolli versiooni	versioon võib jääda samaks, tee muutub

Tüüpilised algaja vead

- paigaldatakse pakette enne keskkonna aktiveerimist
- aetakse segi projekti `venv` ja süsteemi Python
- eeldatakse, et igasse konteinerisse või igasse väikesse proovikausta on alati eraldi `venv` vaja

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) kasutaja@mac 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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: venv hoiab ühe projekti Pythoni paketid eraldi; kõige olulisem on, et terminal, pip ja IDE näeksid sama keskkonda.

Põhikujud

- `python3 -m venv .venv` — loo keskkond
- `source .venv/bin/activate` — aktiveeri shellis
- `python -m pip install -U pip` — uuenda pip
- `python -m pip install requests` — paigalda pakett
- `python -m pip list` — vaata pakke

- `command -v python` — kontrolli tõlgendit
- `deactivate` — välju keskkonnast

Olulised märgid

- `(.venv)` — aktiivne keskkond
- `.venv/` — projekti sees
- `python -m pip` — kindlam kui pip
- `Docker` — terve keskkond

Pane tähele: Kui pip install läks valesse kohta, kontrolli kohe command -v python ja command -v pip; enamasti on probleem just vales aktiivses keskkonnas.

Edasi: Järgmine loomulik samm: Dockeri alused.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

Dockeri alused

Loogika

Docker aitab panna rakenduse koos käivituskeskkonnaga ühte korratavasse pakki.

Kõige tähtsam mõttejärjekord on:

1. vali või ehita `image`
2. käivita sellest `konteiner`
3. jaga vajadusel port või kaust hostmasinaga
4. vaata logisid ja käivita käske konteineri sees
5. kui teenuseid on mitu, kirjelda need `compose.yaml` failis

Git ja `venv` ei kao Dockeri tõttu ära. Tavaliselt elab lähtekood hostmasina Gitrepos, Docker annab sellele koodile ühtse jooksutuskeskkonna.

Kiire orientiir

Mõiste või käsk	Milleks	Mida tavaliselt näed
`image`	käivitusmall	nimi ja tag, näiteks `python:3.13-slim`
`konteiner`	image'ist käivitatud protsess	töötav või peatunud eksemplar
`Dockerfile`	image'i ehitamise retsept	`FROM`, `WORKDIR`, `COPY`, `RUN`, `CMD`
`docker build`	ehita image	buildi kihid ja image'i nimi
`docker run`	käivita konteiner	programmi väljund või konteineri ID

Mõiste või käsk	Milleks	Mida tavaliselt näed
`docker ps`	vaata töötavaid konteinereid	konteinerite tabel
`docker logs`	loe konteineri väljundit	programmi logiread
`docker exec`	käivita käsk töötavas konteineris	käsu väljund konteineri seest
`docker compose`	käivita seotud teenused	teenuste logid ja seis

Tüüpilised algaja vead

- aetakse segi `image` ja konteiner
- arvatakse, et konteineri sees käsitsi muudetud failid jäävad alati alles
- kasutatakse `docker exec` konteineri käivitamiseks, kuigi see töötab ainult juba jooksvas konteineris
- eeldatakse, et `depends_on` tähendab andmebaasi täielikku valmisolekut
- pannakse konteinerisse `venv` ainult harjumusest, kuigi konteiner ise juba isoleerib keskkonda
- kustutatakse `docker compose down -v` käsuga kogemata andmebaasi volume

Kas Docker töötab

Enne esimese näite tegemist kontrolli, kas Docker on masinas olemas ja teenus töötab.

```
docker --version
docker compose version
docker run --rm hello-world
```

`hello-world` on väike test-image. Esimesel käivitamisel võib Docker selle internetist alla laadida.

macOS-is ja Windowsis tähendab Docker tavaliselt Docker Desktopi või muud taustal töötavat virtuaalmasina kihti. Kui käsk ütleb, et Docker daemon ei tööta, ava Docker Desktop ja proovi uuesti.

Image ja konteiner

`image` on valmis käivitusmall. `konteiner` on sellest mallist käivitatud konkreetne eksemplar.

Ühest image'ist võib korraga töötada mitu konteinerit:

```
docker run --rm alpine echo tere
docker run --rm alpine uname -a
```

Siin:

- `alpine` on image
- `echo tere` või `uname -a` on käsk, mis käivitatakse konteineri sees
- `--rm` kustutab konteineri pärast lõpetamist

Oluline algaja kaitsepiire: konteiner on vahetatav. Kui tahad andmeid säilitada, hoia need hostmasina kaustas, named volume'is või Git-repos.

`venv` vs Docker

`venv` ja Docker lahendavad eri probleemi.

Olukord	Tavaliselt sobib
ainult Pythoni paketid peavad olema eraldi	`venv`
vaja on kindlat Linuxi keskkonda või süsteemipakette	Docker
projekt vajab andmebaasi, vahemälu või mitut teenust	Docker Compose
tahad kiiresti väikest lokaalset skripti teha	`venv`
tiimil peab sama rakendus eri masinates sarnaselt käivituma	Docker

Dockeri sees ei ole eraldi `venv` enamasti vajalik. Kui üks konteiner jooksutab üht rakendust, on konteiner ise eralduskiht.

Esimene väike Dockerfile

Kõige selgem esimene näide on väike programm, mille image ise käivitab.

```
mkdir -p ~/tmp/docker-naide
cd ~/tmp/docker-naide
```

`app.py`

```
print("Tere konteinerist")
```

`Dockerfile`

```
FROM python:3.13-slim
WORKDIR /app
COPY app.py .
CMD ["python", "app.py"]
```

Siin:

- `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 vaikimisi käivitatava käsu

Ehita ja käivita:

```
docker build -t tere-rakendus .
docker run --rm tere-rakendus
```

Tulemus peaks olema:

```
Tere konteinerist
```

Punkt `.` käsu lõpus tähendab build context'i: kausta, mille sisu Docker buildi ajal näeb.

`.dockerignore`

Build context ei peaks sisaldama kõike, mis projektikaustas on.

Tüüpiline `.dockerignore`:

```
.git
.venv
__pycache__/
dist/
*.log
```

See teeb buildi kiiremaks ja vähendab riski, et image'isse satub juhuslikke kohalikke faile.

`docker run`: käsk, port ja nimi

`docker run` loob uue konteineri ja käivitab selle.

```
docker run --rm python:3.13-slim python --version
```

Kui konteineri sees jookseb server, tuleb port hostmasinasse siduda.

```
docker run --rm -d --name pyweb -p 8000:8000 python:3.13-slim python -m http.server 8000
```

Siin:

- `-d` paneb konteineri taustale
- `--name pyweb` annab konteinerile nime
- `-p 8000:8000` seob hosti pordi `8000` konteineri pordiga `8000`

Kontroll:

```
docker ps
docker logs pyweb
docker stop pyweb
```

Brauseris peaks teenus olema aadressil:

`http://localhost:8000`

Kui port on juba kasutusel, vali vasakul pool teine hosti port, näiteks `-p 8080:8000`.

`logs` ja `exec`

Kui konteiner on juba käimas, alusta veaotsingut logidest.

```
docker logs pyweb
docker logs -f pyweb
```

`-f` jälgib logi jooksvalt.

Käsk konteineri sees:

```
docker exec pyweb pwd
docker exec -it pyweb sh
```

`docker exec` ei loo uut konteinerit. See käivitab käsu olemasolevas töötavas konteineris.

Praktiline märkus: paljudes väikestes image'ites ei ole `bash` paigaldatud. `sh` töötab sagedamini.

Bind mount: hosti kaust konteinerisse

Builditud image sobib valmis rakenduse proovimiseks. Arenduses tahad sageli, et konteiner näeks sinu praegust projektikausta kohe.

```
docker run --rm -it --mount type=bind,src="$PWD",dst=/app -w /app python:3.13-slim sh
```

Sama lühem, vanem ja väga levinud kuju on:

```
docker run --rm -it -v "$PWD":/app -w /app python:3.13-slim sh
```

Siin:

- `src="$PWD"` või `"$PWD"` on sinu praegune kaust hostmasinas
- `dst=/app` või `/app` on sama kaust konteineri sees
- `-w /app` teeb sellest töökausta

Konteineri sees saad näiteks käivitada:

```
python app.py
python -m pip install -r requirements.txt
pytest
```

Bind mount'i puhul muutub hosti fail päriselt. See on arenduses mugav, aga tähendab ka, et konteineris tehtud muudatus võib muuta sinu töökausta.

Named volume

Bind mount seob konteineri konkreetse hosti kaustaga. Named volume on Dockeri hallatav püsiv andmeala.

Kõige tavalisem kasutus on andmebaasi andmed:

```
volumes:
  pgdata:
```

Compose'i teenuses:

```
services:
  db:
    image: postgres:16
    volumes:
      - pgdata:/var/lib/postgresql/data
```

Pea meeles:

- `docker compose down` peatab ja eemaldab teenuste konteinerid ning võrgu
- `docker compose down -v` eemaldab ka named volume'id
- andmebaasi volume'i kustutamine tähendab sageli andmete kustutamist

Arenduse tööjaotus

Hea algaja tööjaotus on:

Kus	Mida tee
hostmasinas	muuda faile, tee Git commit'e, hoia dokumentatsiooni
konteineris	käivita rakendus, installi sõltuvusi, testi, silu keskkonda
volume'is	hoia andmeid, mis peavad konteineri vahetamisel alles jääma

Väldi olulise koodi kirjutamist ainult konteineri sisemisse failisüsteemi. Kui konteiner kustub, võib see töö kaduda.

Kui rakendus vajab pakette

Pythoni projektis on sõltuvused sageli failis `requirements.txt`.

`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 järjekord:

- sõltuvuste fail kopeeritakse enne rakenduse koodi
- `RUN pip install ...` jääb võimalusel build cache'i
- `COPY app.py .` tuleb hiljem, sest rakenduse kood muutub sagedamini

`docker compose`: mitu teenust korraga

Kui projekt vajab rakendust ja andmebaasi, muutub üks pikk `docker run` käsk ebamugavaks. Siis kirjelda teenused failis `compose.yaml`.

```
services:
  app:
    image: python:3.13-slim
    working_dir: /app
    volumes:
      - .:/app
    command: python -m http.server 8000
    ports:
      - "8000:8000"

  db:
    image: postgres:16
    environment:
      POSTGRES_DB: opik
      POSTGRES_USER: opik
      POSTGRES_PASSWORD: opikpass
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Tüüpilised käsud:

```
docker compose up
docker compose up -d
docker compose logs -f app
docker compose exec app sh
docker compose down
```

Tähendused:

- `up` käivitab teenused ja näitab logisid
- `up -d` käivitab teenused taustal
- `logs -f app` jälgib ühe teenuse logi
- `exec app sh` avab käsu töötavas teenusekonteineris
- `down` peatab ja eemaldab selle compose-projekti konteinerid ning võrgu

Uues Dockeris kasutatakse kuju `docker compose`. Vanemates juhendites võid näha kuju `docker-compose`.

`depends_on` ei ole valmisolekukontroll

Compose'is näed sageli:

```
depends_on:
  - db
```

See aitab käivitusjärjekorraga, kuid ei tähenda automaatselt, et andmebaas on juba ühendusteks valmis.

Päris projektis võib vaja minna:

- rakenduse enda retry-loogikat
- andmebaasi healthcheck'i
- käivitusskripti, mis ootab teenuse valmidust

Algaja jaoks piisab esialgu teadmisest: kui `app` ütleb käivitumisel, et andmebaas pole valmis, vaata esmalt `docker compose logs db` ja proovi pärast hetke uuesti.

Millal valida `run`, `build` või `compose`

Vajadus	Sobiv tööriist
proovi üht käsku puhtas image'is	`docker run --rm image käsk`
kontrolli valmis rakenduse image'it	`docker build` ja `docker run`
arenda koodi konteineri sees, kuid failid jäävad hosti	bind mount
käivita rakendus koos andmebaasi või muu teenusega	`docker compose`
IDE avab kogu tööruumi konteineris	arenduskonteiner ehk devcontainer

See valik vähendab segadust: ära tee compose'i ainult selleks, et käivitada üht lühikest käsku, ja ära kirjuta üht hiiglaslikku `docker run` käsku, kui projektis on mitu teenust.

Ohutus ja privaatsus

Konteiner ei ole maagiline turvasein.

Ole ettevaatlik, kui:

- image on tundmatust allikast
- annad konteinerile hosti kaustu bind mount'iga
- paned keskkonnamuutujatesse paroole või võtmeid
- näed juhendit, mis mount'ib Docker socket'i, näiteks `/var/run/docker.sock`
- käsk kasutab `--privileged`

Algaja hea reegel: ära käivita võõrast image'it koos oma kodukataloogi või salajaste failide mount'iga.

Kettaruumi kontroll ja koristamine

Docker võib aja jooksul koguda image'eid, peatunud konteinereid ja build cache'i.

Ohutu vaatamine:

```
docker system df
docker ps -a
docker images
```

Koristuskäske kasuta alles siis, kui saad aru, mida need kustutavad:

```
docker container prune
docker image prune
```

Need ei ole esimesed õppimiskäsud. Need on hilisemaks, kui Docker on juba igapäevases kasutuses.

Seos arenduskonteineritega

Arenduskonteiner ehk `devcontainer` on Dockeri peale ehitatud IDE-töövoog.

Tavaliselt toimub nii:

- Docker käivitab konteineri
- IDE ühendub konteineri sisse
- projektikaust mount'itakse tööruumina
- terminal, testid ja keele tööriistad jooksevad konteineris

Arenduskonteiner ei asenda Dockeri põhimõtteid. Ta teeb sama loogika mugavamaks, eriti siis, kui projekt vajab keerulisemat keskkonda.

Praktiline õppimisjärjekord

Õpi Dockerit selles järjekorras:

1. `docker run --rm alpine echo tere`
2. image ja konteineri vahe
3. väike `Dockerfile`

4. `docker build -t nimi .`
5. port, logid ja `exec`
6. bind mount arenduseks
7. `docker compose` mitme teenuse jaoks
8. volume'id ja andmete püsivus

Nii on igal sammul selge, kas sa käivitad valmispilti, ehitad uut image'it, jagad faile või juhid teenuste komplekti.

Minitest

1. Mis vahe on image'il ja konteineril?
2. Miks on `--rm` kasulik lühikeste katsete puhul?
3. Mida teeb `-p 8000:8000`?
4. Mis vahe on bind mount'il ja named volume'il?
5. Millal valid `docker compose` tavalise `docker run` asemel?
6. Miks ei tasu `docker compose down -v` käsu tähendust arvamata käivitada?

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: image on käivitusmall; konteiner on sellest tehtud töötav eksemplar; andmed püsivad ainult hostis, volume'is või Git-repos.

Põhikäsud

- `docker --version` — kontrolli olemasolu
- `docker run --rm alpine echo tere` — ohutu esimene käivitus
- `docker build -t rakendus .` — ehita image
- `docker run --rm rakendus` — käivita image
- `docker ps` — töötavad konteinerid
- `docker logs -f nimi` — jälgi logi
- `docker exec -it nimi sh` — sisene töötavasse konteinerisse
- `docker compose up -d` — teenused taustale
- `docker compose down` — peata komplekt

Olulisemad lipud, märgid ja kiirnupud

- `image` — käivitusmall
- `konteiner` — töötav eksemplar
- `Dockerfile` — image'i retsept

- `compose.yaml` — mitu teenust
- `volume` — püsiv andmesisu
- `registry` — image'ite hoidla
- `--rm` — kustuta lühikatse konteiner pärast lõppu
- `-p 8000:8000` — seo hosti port konteineri pordiga
- `--mount type=bind,src="$PWD",dst=/app` — jaga praegune kaust konteinerisse
- `-w /app` — määra konteineri töökaust
- `down -v` — peatab ja kustutab ka volume'id

Pane tähele: Kui konteiner ei tööta, vaata enne docker logs; ära käivita tundmatut image'it oma kodukataloogi või saladuste bind mount'iga.

Edasi: Järgmine loomulik samm: IDE-d ja arenduskeskkonnad.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

IDE-d ja arenduskeskkonnad

Loogika

Arenduskeskkond on tervik, mitte ainult tekstiredaktor. Terminal, Git, keelekeskkond ja vajadusel Docker või kaugühendus peavad töötama koos.

Kiirülevaade

Eesmärk on panna terminal, Git, interpreter ja vajadusel konteinerid sama projekti koos nägema.

Kontroll või valik	Milleks	Mida tavaliselt näed
`--version` käsud	kontrolli tööriista olemasolu	üks lühike versioonirida
IDE interpreter	vali projekti Python, Node vms	valitud keskkond IDE seadetes
sisseehitatud terminal	jooksuta käske projekti kontekstis	sama töökaust ja projektifailid
devcontainer või Docker	anna projektile eraldi käivituskeskkond	konteineri terminal ja tööriistad
puuduv tööriist	vajab paigaldust või PATH parandust	"command not found" laadne viga

Tüüpilised algaja vead

- arvatakse, et IDE "teab ise", millist Pythoni või Node'i kasutada
- terminal ja IDE kasutavad tegelikult eri keskkondi

- proovitakse korraga liiga palju tööriistu, kuigi alguses piisab lihtsast tervikust

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, on järgmine samm sageli 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`.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: hea arenduskeskkond tähendab, et terminal, Git, interpreter ja vajadusel konteiner osutavad samale projektile, mitte eri maailmadele.

Kontrollid ja valikud

- `python3 --version` — kontrolli Pythonit
- `python3 -m pip --version` — kontrolli pip-i
- `git --version` — kontrolli Git-i

- `docker --version` — kontrolli Dockerit
- `node --version` — kontrolli Node'i
- `npm --version` — kontrolli npm-i

Olulised mõisted

- `interpreter` — tegelik käivitaja
- `sisseehitatud terminal` — sama projektivaade
- `Remote SSH` — tööta kaugmasinas
- `devcontainer` — IDE konteineris

Pane tähele: Kui IDE-s "miski ei tööta", kontrolli kõigepealt, kas IDE kasutab sama Pythonit, Node'i või konteinerit mis sinu terminal.

Edasi: Järgmine loomulik samm: Andmeteaduse eelteadmised käsurea vaates.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

Andmeteaduse eelteadmised käsurea vaates

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

Kiirülevaade

Eesmärk on siduda käsurida andmeteaduse eelteadmistega: mitte õpetada kogu statistikat või Pythonit, vaid näidata, kuidas andmete kuju ja töövoog nähtavaks teha.

Teema või tööriist	Milleks	Mida praktikas näed
programmeerimine	korda andmetöötlussamme	skriptid ja korduvad käivitused

Teema või tööriist	Milleks	Mida praktikas näed
SQL ja andmebaasid	struktureeri ja küsi andmeid	tabelid, võtmed ja päringud
failivormingud	määra, kuidas andmeid lugeda	CSV read, JSON puud, XML märgendid
matemaatika ja statistika	anna tulemustele sisu	analüüs, mida käsurida üksi ei asenda
`head`, `column`, `jq`, `wc`, `sqlite3`	tee esimene kontroll	faili kuju, read, veerud ja mahud
Python, SQL, notebook	tee tõsisem analüüs	suurem töövoog pärast esmast kontrolli

Tüüpilised algaja vead

- arvatakse, et andmeteadus algab kohe suure mudeli või teegiga
- minnakse otse keerukasse analüüsi enne, kui andmefaili kuju on kontrollitud
- alahinnatakse failivormingute ja töövoo korrastatuse tähtsust

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.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: andmeteaduse stardis ei piisa ühest tööriistast; vaja on korraga failivormingute tunnetust, SQL-i mõtteviisi, programmeerimist ja statistilist mõtlemist.

Põhikujud

- `fail -> head/less -> column/jq -> sqlite3 -> python3` — alusta väikese vaatusega
- `Pythoni venv` — projekti töölaud
- `CSV, JSON ja XML` — andmete kuju
- `SQLite ja Python` — esimene andmebaas
- `Teksti teisendamine` — väiksed filtrid
- `Vood ja tabelid` — koondamine

Põhiteljed

- `programmeerimine` — korduv töö loogikaks
- `SQL` — küsi ja seo andmeid
- `vormingud` — CSV, JSON, XML

- `statistika` — anna tähendus

Pane tähele: Ära mine otse mudeli või notebook'i juurde enne, kui oled vähemalt korra kontrollinud, mis kujul andmed sul tegelikult on.

Edasi: Järgmine loomulik samm: CSV, JSON ja XML käsureal.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

CSV, JSON ja XML käsureal

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

Kiirülevaade

Eesmärk on kiiresti ära tunda andmete kuju ja valida selle järgi järgmine tööriist.

Vorming või käsk	Milleks	Mida tavaliselt näed
CSV	lihtne tabel	read ja väljad
JSON	objektide ja massiivide puu	taandega struktuur või valitud võtmed
XML	märgenditega puu	elemendid ja pesastatud struktuur
`head`	vaata faili algust	esimesed read
`column`	tee lihtne tabel loetavamaks	joondatud veerud
`python3 -m json.tool`, `jq`	vaata JSON-i	vormindatud või filtreeritud JSON
`xmllint`	vorminda XML	loetavam märgendipuu

Tüüpilised algaja vead

- aetakse segi tabeli, puu ja märgendipõhise struktuuri loogika
- proovitakse keerukat CSV-d töödelda liiga lihtsa eraldajaloogikaga
- eeldatakse, et kõik JSON-failid on ühetaolised lihtsad võtme-väärtuse paarid

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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: vormingu tüüp ütleb, milline tööriist mõistab andmeid kõige loomulikumalt: tabelit, puud või märgendistruktuuri.

Põhikujud

- `head fail.csv` — vaata CSV algust
- `column -s, -t < fail.csv` — joonda lihtne CSV
- `cut -d, -f1 fail.csv` — võta üks veerg
- `python3 -m json.tool fail.json` — vorminda JSON
- `jq '.students[].name' fail.json` — vali JSON välju
- `xmllint --format fail.xml` — vorminda XML

Olulised mõisted

- `CSV` — lihtne tabel
- `JSON` — objektide puu
- `XML` — märgendipuu
- `jq` — JSON filtriks

Pane tähele: Kui CSV-s on jutumärgid, komad välja sees või reavahetused, ei pruugi cut -d, enam piisata; siis mine pigem Pythonisse või spetsiaalsema parseri juurde.

Edasi: Järgmine loomulik samm: Andmebaasi algus: sqlite ja Python.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

Andmebaasi algus: SQLite ja Python

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.

Kiirülevaade

Eesmärk on liikuda tekstifailidest andmebaasi mõtlemisse ilma eraldi serverit haldamata.

Käsk või mõiste	Milleks	Mida tavaliselt näed
`sqlite3 andmed.db`	ava või loo andmebaasifail	interaktiivne SQLite'i käsurida
`.tables`	näita tabeleid	tabelite loend
SQL `SELECT`	loe ridu	tulemuste tabel või loend
SQL `INSERT`	lisa ridu	edukal juhul sageli vaikne
Pythoni `sqlite3`	kasuta sama andmebaasi programmist	päringu tulemus Pythoni objektidena

Tüüpilised algaja vead

- aetakse segi andmebaasifail ja selles olevad tabelid
- proovitakse `JOIN`-i enne, kui tabelite seos on iseenda jaoks selge
- unustatakse, et SQL-i tulemused on päringud andmete peal, mitte "faili uus sisu"

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.

```
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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: SQLite on sild tekstifailide ja päris andmebaasimõtte vahel: failina lihtne, SQL-i mõttes siiski relatsiooniline andmebaas.

Põhikujud

- `sqlite3 andmed.db` — ava või loo fail
- `sqlite3 andmed.db '.tables'` — vaata tabeleid
- `sqlite3 andmed.db '.schema'` — vaata struktuuri
- `sqlite3 andmed.db 'select * from students limit 5;'` — piilu ridu
- `sqlite3 andmed.db 'select city, count(*) from students group by city;'` — koonda read
- `sqlite3 andmed.db 'select s.name, r.score from results r join students s on s.id = r.student_id;'` — ühenda tabelid
- `python3 naide.py` — kasuta Pythonist

Olulised mõisted

- `primary key` — rea unikaalne id
- `foreign key` — viide teise tabelisse
- `JOIN` — seo tabelid
- `GROUP BY` — koonda read

Pane tähele: Ära tee JOIN-i ainult käsu pärast; enne joonista enda jaoks välja, milline väli viitab millisele tabelile.

Edasi: Järgmine loomulik samm: Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java

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

Kiirülevaade

Eesmärk on mõista, miks eri keelte programmide käivitamine erineb: mõni loetakse jooksvalt, mõni ehitatakse enne artefaktiks.

Kuju või tööriist	Milleks	Mida tavaliselt näed
tõlgendiga skript	käivita lähtekood otse	programmi väljund kohe
kompilaator	tee lähtekoodist uus artefakt	binaar või buildikataloog
`go build`, `cargo`	juhi buildisamme tööriistaga	buildilogi ja väljundfailid
Java	kompileeri ja käivita eri sammudena	`.class` failid ja programmi väljund
süntaksi- või kompileerimisviga	käivitus peatub enne tulemust	veateade enne programmi tööd

Tüüpilised algaja vead

- arvatakse, et kõik lähtefailid on otse käivitatavad
- aetakse segi buildikäsk ja run-käsk
- unustatakse vaadata, mis failid pärast buildi tegelikult tekkisid

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/`

Keel	Lähtefail	Tüüpiline käivitus	Mis tekib
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.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: mõni fail käivitatakse tõlgendiga otse; mõni fail kompileeritakse kõigepealt teise vormi; mõni tulemus on päris binaar

Põhikäsud

- `go` — Go tööriist
- `cargo` — Rusti tööriist
- `python3` — käivita Python
- `cc` — kompileeri C
- `javac` — kompileeri Java
- `java` — käivita JVM-is

Tüüpilised kujud

- `chmod +x hello.sh` — õigused
- `python3 hello.py` — käivita Python
- `cc hello.c -o hello-c` — kompileeri C
- `go run hello.go` — Go tööriist
- `go build -o hello-go hello.go` — Go tööriist
- `mkdir -p hello-rust` — loo puuduv rada

Pane tähele: Kui käivitatav fail "ei tööta", kontrolli kõigepealt, kas ta üldse ehitati valmis, kus ta asub ja kas pead kasutama kuju ./fail.

Edasi: Järgmine loomulik samm: LaTeX käsurealt.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

LaTeX käsurealt

Loogika

LaTeX on selles raamatus pigem järgmine samm pärast Markdowni, kui tahad jõuda väga hea PDF-väljundini.

See teema on seotud buildi ja dokumentide vormindamisega, mitte Linuxi kõige esimese baasringiga.

Kiirülevaade

Eesmärk on teha tekstifailist kvaliteetne PDF, kus vormindus, valemid ja viited on korratavalt ehitatavad.

Käsk või fail	Milleks	Mida tavaliselt näed
`pdflatex fail.tex`	kompileeri PDF-iks	pikk logi ja `.pdf` fail
`xelatex fail.tex`	kasuta moodsamat fondituge	pikk logi ja `.pdf` fail
`latexmk`	korda vajalikke samme automaatselt	mitu kompileerimissammu
`.tex`	lähtefail	jääb tekstifailiks
`.aux`, `.log`	abifailid	tekivad PDF-i kõrval

Tüüpilised algaja vead

- eeldatakse, et üks kompileerimiskäik lahendab alati kõik ristviited
- ehmatatakse logi pikkust, kuigi suur osa sellest on normaalne buildiinfo
- muudetakse väljundfaile, kuigi tegelik allikas on `.tex`

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:

Vali ainult üks sinu süsteemile sobiv paigaldustee:

Debianis või Ubuntus:

```
sudo apt install texlive-latex-base
```

macOS-is Homebrew kaudu:

```
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.

Lisalugemine

Selle teema usaldusväärsemad viited leiad lisast Lisa E: usaldusväärsed viited ja lisalugemine.

Peatüki täisspikker

Tase: **Edasijõudnu**

Eesmärk: LaTeX-i mõte on lihtne: lähtetekst jääb tekstifailiks, aga buildist sünnib väga täpselt juhitav PDF.

Põhikujud

- `pdflatex tere.tex` — kompileeri PDF
- `xelatex tere.tex` — parem Unicode tugi
- `latexmk -pdf tere.tex` — korda vajalikud käigud
- `latexmk -xelatex tere.tex` — xelatex automaatselt
- `ls tere.*` — vaata abifaile
- `open tere.pdf` — ava tulemus

Olulised failid

- `.tex` — lähtefail
- `.pdf` — tulemus
- `.aux` — ristviidete abi
- `.log` — kompileerimislogi

Pane tähele: Kui ristviited või sisukord ei ilmu kohe õigesti, ei tähenda see tingimata viga; LaTeX vajab sageli rohkem kui üht kompileerimiskäiku.

Edasi: Järgmine loomulik samm: Lisa A: kopeeritavad minitestid.

Osa PDF: ./spikrid/osa-v-arendus-ja-toovood-spikker.pdf

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
```

Peatüki täisspikker

Tase: **Referents**

Eesmärk: Siia koondame lühikesed peatükkide kaupa jaotatud copy-paste harjutused.

Põhikujud

- `mkdir -p ~/tmp/test1` — loo puuduv rada
- `cd ~/tmp/test1` — vaheta kaust
- `touch a.txt b.txt` — loo või aja tempel
- `mkdir kaust` — loo kaust
- `mv b.txt kaust/` — liiguta/nimeta
- `ls -la` — pikk + peidetud
- `printf` — vorminda tekst

Olulisemad lipud, märgid ja kiirnupud

- `|` — toru edasi
- `>` — kirjuta üle
- `>>` — lisa faili lõppu juurde

Pane tähele: Kui kahtled, kontrolli enne pwd ja ls, siis tee alles järgmine samm.

Edasi: Järgmine loomulik samm: Lisa B: spikrite register.

Osa PDF: ./spikrid/lisad-spikker.pdf

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 juurde; ära kirjuta üle
- `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
- `Ctrl-c` katkesta programmi töö
- `Ctrl-z` peata programm ajutiselt
- `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 hiljem, kui ajaloo-otseteed on juba tuttavad
- `!n` korda ajaloo kirjet numbri järgi
- `!sona` korda viimast sobivat käsku

Peatüki täisspikker

Tase: **Referents**

Eesmärk: kõige lühemad meelespead teemade kaupa: failid, tekst, võrk, arendus, loogika ja protsessid

Mida siit leiad

- `Failid` — liigu ja vaata
- `Tekst` — loe ja filtreeri
- `Võrk` — ühendu ja kopeeri
- `Arendus` — Git, Docker, build
- `Loogika` — torud ja vead
- `Protsessid` — tööd ja signaalid

Pane tähele: See lisa on kõige lühem meelespea; kui mõni käsk tundub hägune, mine tagasi vastava peatüki juurde.

Edasi: Järgmine loomulik samm: Lisa C: sõnastik ja terminoloogia.

Osa PDF: ./spikrid/lisad-spikker.pdf

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
- `CLI` ehk `command-line interface`: käsurealiides; viis arvutiga suhelda käske kirjutades
- `GUI` ehk `graphical user interface`: graafiline kasutajaliides; viis arvutiga suhelda akende, nuppude, menüüde ja ikoonide kaudu
- `viip` ehk `prompt`: terminalirea alguses olev tekstiosa, mis näitab, et shell ootab sisestust; jooksvas tekstis eelista sõna `viip`
- `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 `~`
- `bin`: käivitatavate programmide kataloginimi; ajalooliselt sõnast `binary`, näiteks `/bin`, `/usr/bin` või `~/bin`
- `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

Peatüki täisspikker

Tase: **Referents**

Eesmärk: ühtlusta sõnad nagu CLI, GUI, viip, shell, lipp, stdin, stdout ja repo kogu õpiku ulatuses

Põhimõisted

- `CLI` — käsurida
- `GUI` — graafiline liides
- `viip ehk prompt` — ootab sisestust
- `shell` — tõlgendab käske
- `lipp` — käsu valik
- `repo` — Git-i hoidla

Pane tähele: Kui valid kahe sõna vahel, eelista seda lisa kui kogu õpiku terminoloogilist alust.

Edasi: Järgmine loomulik samm: Lisa D: edasised võimalused ja lugejate soovid.

Osa PDF: ./spikrid/lisad-spikker.pdf

Lisa D: edasised võimalused ja lugejate soovid

See lisa ei ole vigade nimekiri ega kriitika olemasolevale materjalile. Põhiraamat katab juba käsurea baasi, süsteemi pildi, failid, võrgu, tekstivood, arendustööriistad ja mitu kõrvalrada. Siin on pigem avatud nimekiri teemadest, mida võiks järgmistes versioonides süvendada.

Õpik on kasvanud kiiresti ja mitmed varasemad puudujäägid on nüüdseks kaetud. Lisandunud on näiteks:

- failisüsteemi kaart ja kettaruumi peatükk
- lihtne veaotsing, võrgu põhitööriistad, logid ja teenused
- `tmux`, `screen`, `cron`, `find` ja `xargs`
- esimene shelliskript ning nüüd ka shelli seadistusfailide lisa `bash`-i ja `zsh`-i jaoks
- Git-i, Pythoni virtuaalkeskkondade, Dockeri ja IDE-de peatükid
- andmeteaduse stardirada, `CSV`, `JSON`, `XML`, SQLite ja LaTeX
- miniõpikud eri sihtrühmadele
- spikrite register ja osa-spikrite HTML/PDF-vaated
- usaldusväärsete viidete ja lisalugemise lisa
- kuvatõmmiste töövoog ja valitud peatükkidesse seotud pildid

Seetõttu ei ole järgmine küsimus enam “mis kõige tähtsam puudub?”, vaid “milline süvendus oleks lugejale kõige kasulikum?”.

Kuidas lugeja saab aidata

Kõige väärtuslikum tagasiside on konkreetne. Kui saadad oma eelistusi või parandusi, siis on eriti kasulik mainida:

- milline peatükk või teema sind aitas
- kus jäi seletus liiga lühikeseks või liiga järsuks
- millist teemat tahaksid näha järgmises versioonis
- kas vajad rohkem algaja näiteid, spikreid, arendaja töövooge või süsteemihalduse selgitusi
- millises keskkonnas sa õpikut kasutad: Linux, macOS, WSL, kooli server, Raspberry Pi või midagi muud

Tagasisidet saab saata lehe `Õpikust` kaudu. Seal on valmis kirjapõhi soovituste ja murekohtade saatmiseks.

Tugevad järgmised kandidaadid

1. GitHubi koostööharjutused suurema projekti jaoks

Git-i peatükk katab nüüd ka harude haldamise, sissetulevad muudatused, `merge`/`rebase` vahe, konfliktid ning GitHubi issue ja pull request'i põhitöövoo. Järgmine sügavam ring võiks olla eraldi harjutusrada, kus sama projekti kallal tegutseb mitu inimest:

- issue'dest tööjärje tegemine
- pull request'i review eri rollides
- fork ja upstream avatud lähtekoodi töövoos
- mitme sõltuva haru korrastamine
- release'i ettevalmistamine GitHubis
- konfliktide harjutamine meelega tehtud näidisrepos

See oleks kasulik lugejale, kes oskab juba Git-i põhitöövoogu, kuid tahab harjutada päris meeskonnatöö rütmi.

2. Shelliskriptide teine aste

Esimese skripti peatükk annab alguse. Järgmine tase võiks lisada:

- funktsioonid
- `case`
- `set -euo pipefail`
- ajutised failid
- argumentide kontroll
- veakindlam sisenditöötlus
- skripti jagamine väiksemateks loetavateks osadeks

See sobiks hästi neile, kes tahavad muuta korduvad käsurea töövood päris tööriistadeks.

3. Backup ja taastamine

Failide kopeerimine, sünkroonimine, arhiivid ja räsid on olemas, aga eraldi ülesandepõhine peatükk võiks siduda need üheks tervikuks:

- varukoopia põhimõtted
- `rsync` varundamiseks
- arhiiv ja snapshot
- checksum'id ja tervikluse kontroll
- miks taastamist peab ka päriselt proovima
- mida tähendab "varukoopia on olemas" ainult siis, kui taastamine on läbi proovitud

See on praktiline teema nii õppijale, arendajale kui väikese serveri haldajale.

4. `systemd` timerid

`cron` on olemas, kuid Linuxi süsteemides kohtab üha sagedamini ka `systemd` timereid. Edasine peatükk võiks selgitada:

- mis vahe on `cron`-il ja `systemd` timeril
- kuidas käivad kokku `.service` ja `.timer`
- kuidas vaadata timeri logisid
- millal timer on parem valik kui `cron`

See teema sobib pigem Linuxi-haldamise järgmisse kihti.

5. R ja notebook'ide sild

Andmeteaduse stardirada on olemas, kuid analüüsi poolel võiks hiljem lisada silla:

- `R`
- Jupyter või muu notebook'i tööloogika
- millal kasutada shelli
- millal minna SQL-i, Pythonisse või R-i
- kuidas käsurida aitab andmeid enne notebook'i avamist kontrollida

See ei peaks muutma õpikut andmeteaduse kursuseks, vaid aitama lugejal valida õige tööriista õiges etapis.

6. Regulaaravaldiste eraldi harjutusrada

`grep`, `sed`, `awk` ja `perl` peatükid puudutavad regulaaravaldisi juba mitmes kohas. Eraldi väike harjutusrada võiks anda:

- lihtsad mustrid

- rea algus ja lõpp
- märgiklassid
- kordused
- grupid ja tagasiviited
- millal regex ei ole enam parim tööriist

See oleks kasulik siis, kui lugeja tahab mustriotsingu teadlikumalt selgeks saada.

Teemad, mis sobivad hilisemaks

Mõned teemad on kasulikud, kuid ei pea tingimata tulema enne ülalolevaid:

- väga sügav `awk` või `sed`
- keerulisem Docker Compose mitme teenusega projektides
- terminali prompti väga peen välimuse häälestamine
- kerneli või süsteemikutsete süvateooria
- paketihalduse sisemine ehitus
- SSH tunnelite ja hüppemasinate keerulisemad skeemid

Need võivad sobida eraldi edasijõudnute peatükkideks või miniõpikuteks.

Praegune seis

Praegune õpik on juba kasutatav nii õppimiseks kui referentsiks. Edasine töö võiks liikuda vähem "veel üks baaspeatükk" ja rohkem "vali üks päris probleem ning näita seda otsast lõpuni" suunas.

Kui mitu lugejat soovib sama teemat, on see hea märk, et just sellest võiks saada järgmine peatükk või miniõpik.

Peatüki täisspikker

Tase: **Referents**

Eesmärk: järgmine voor peaks pigem süvendama olemasolevaid tugevaid teemasid kui lisama juhuslikke uusi baaskäske

Tugevad järgmised teemad

- `R ja notebook'id` — analüüsi järgmine aste
- `Git järgmine aste` — harud ja konfliktid
- `Shelliskriptid II` — veakindlam loogika
- `Statistika sild` — andmetöö mõtteviis
- `systemd timerid` — ajastus Linuxis
- `Varukoopiad` — taastamine ja kontroll

Pane tähele: Siin nimekirjas ei ole "mida iga hinna eest kohe lisada", vaid teemad, mis annaksid järgmises ringis kõige rohkem väärtust.

Edasi: Järgmine loomulik samm: Lisa E: usaldusväärsed viited ja lisalugemine.

Osa PDF: ./spikrid/lisad-spikker.pdf

Lisa E: usaldusväärsed viited ja lisalugemine

See lisa koondab teemade kaupa ametlikud ja usaldusväärsed lisalugemise allikad.

Järjekord on üldiselt selline:

1. ametlik getting started või ülevaade
2. ametlik reference või manuaal
3. standard või spetsifikatsioon
4. üks pikem ja hea õppematerjal, kui see on laialt tunnustatud

Kui tahad edasi õppida, siis alusta iga teema puhul esimesest viitest. Kui vajad täpset detaili, mine reference-manuaali juurde.

Shellid ja põhikäsud

1. GNU Bash Reference Manual
2. POSIX Shell Command Language
3. Zsh Documentation
4. Linux man-pages project
5. GNU Coreutils Manual

Linux, macOS, Windows ja WSL

1. Install WSL
2. Windows Subsystem for Linux Documentation
3. GNU Bash Reference Manual
4. Zsh Documentation
5. Linux man-pages project

SSH ja võrgutööriistad

1. OpenSSH Manual Pages
2. curl documentation
3. GNU Wget Manual
4. Linux man-pages project

Tekstitöötlus ja filtrid

1. GNU Grep Manual
2. GNU sed Manual
3. GNU gawk Manual
4. jq Manual
5. GNU Coreutils Manual

Git ja GitHub

1. Git documentation
2. gittutorial
3. giteveryday
4. Pro Git

Python ja virtuaalkeskkonnad

1. Python `venv` documentation
2. Install packages in a virtual environment using pip and venv
3. Python `json` documentation
4. Python `csv` documentation

Docker ja arenduskonteinerid

1. Docker Docs
2. Docker Get Started
3. Docker Get Started: Introduction
4. Lab: Getting Started with Docker
5. Developing inside a Container
6. Create a Dev Container

JSON, CSV, XML ja SQLite

1. Python `json` documentation
2. Python `csv` documentation
3. jq Manual
4. SQLite Documentation
5. SQLite `SELECT` reference
6. Python `sqlite3` documentation

LaTeX

1. LaTeX Documentation
2. LaTeX Project
3. LaTeX links and contributed resources
4. CTAN

Peatüki täisspikker

Tase: **Referents**

Eesmärk: ametlikud käsiraamatud ja getting started allikad shelli, SSH, Git-i, Dockeri, JSON-i, SQLite'i ja LaTeX-i jaoks

Edasiõppimise teemad

- `Shellid ja põhikäsud` — bash, zsh, coreutils
- `SSH ja võrk` — OpenSSH, curl, wget
- `Git ja GitHub` — docs ja Pro Git
- `Docker` — get started ja compose
- `JSON, CSV, XML, SQLite` — ametlikud viited
- `LaTeX` — projekt ja CTAN

Pane tähele: Alusta iga teema puhul esimesest viitest; reference-manuaal on tavaliselt järgmine samm alles siis, kui põhikujutlus on olemas.

Edasi: Järgmine loomulik samm: Lisa F: shelli seadistusfailid bash ja zsh jaoks.

Osa PDF: ./spikrid/lisad-spikker.pdf

Lisa F: shelli seadistusfailid bash ja zsh jaoks

See lisa annab kopeeritavad algusfailid kahe levinud shelli jaoks:

- `bash`
- `zsh`

Shell on käsutõlk: programm, mis loeb terminali sisestatud käsuridu, otsib vastava käsu üles ja käivitab selle. Kui avad terminaliakna, käivitub tavaliselt sinu vaikimisi shell. Seda saad vaadata käsuga:

```
echo "$SHELL"
```

Tüüpiline tulemus on näiteks:

```
/bin/zsh
```

või:

```
/bin/bash
```

Kuhu need failid käivad

Shelli seadistusfailid on tavaliselt sinu kodukataloogis:

Shell	Peamine fail	Kus see asub
`zsh`	`.zshrc`	`~/.zshrc`
`bash` Linuxis	`.bashrc`	`~/.bashrc`
`bash` macOS-is terminali login-shellina	`.bash_profile` ja `.bashrc`	`~/.bash_profile`, `~/.bashrc`

Punktiga algavad failid on peidetud failid. Kodukataloogis näed neid näiteks nii:

```
ls -la "$HOME"
```

Enne muutmist tee oma shelli failist varukoopia. `zsh` puhul:

```
cp ~/.zshrc ~/.zshrc.backup
```

`bash` puhul:

```
cp ~/.bashrc ~/.bashrc.backup
```

Kui faili veel ei ole, võib `cp` anda veateate. See on sel juhul ootuspärane: järelikult ei olnud varasemat seadistusfaili, mida varundada.

Pärast muutmist lae seadistus uuesti sisse:

```
source ~/.zshrc
```

või:

```
source ~/.bashrc
```

Kui kasutad macOS-is `bash`-i ja terminal avab login-shelli, pane faili `~/.bash_profile` vähemalt see:

```
# Lae bash'i tavaline interaktiivne seadistus.
if [ -f "$HOME/.bashrc" ]; then
  . "$HOME/.bashrc"
fi
```

Siis võid põhiseadistuse hoida failis `~/.bashrc`.

Kuidas näiteid kasutada

Vali allpool üks tase ja kopeeri vastav plokk oma shelli faili:

- `zsh` puhul faili `~/.zshrc`
- `bash` puhul faili `~/.bashrc`

Ära pane kõiki tasemeid korraga samasse faili. Alusta ühest tasemest ja lisa hiljem juurde.

Tase 1: kõige lihtsam

See tase muudab ainult ajaloo, lihtsa prompti ja mõne kõige tavalisema aliase. See sobib esimeseks katseks.

`~/.bashrc`

```
# ~/.bashrc
# Tase 1: kõige lihtsam bash'i seadistus.

# Ajalugu.
HISTSIZE=1000
```

```
HISTFILESIZE=2000

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Lihtne prompt.
PS1='\u@\h:\w\$ '
```

~/.zshrc

```
# ~/.zshrc
# Tase 1: kõige lihtsam zsh seadistus.

# Ajalugu.
HISTFILE="$HOME/.zsh_history"
HISTSIZE=1000
SAVEHIST=2000

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Lihtne prompt.
PROMPT='%n@%m:%~%# '
```

Tase 2: lühike igapäevane seadistus

See tase lisab mugava `PATH`-i, ajaloo korduste vähendamise, otsingukäsud ja mõned navigeerimise alias'ed.

~/.bashrc

```
# ~/.bashrc
# Tase 2: lühike igapäevane bash'i seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  PATH="$HOME/bin:$PATH"
fi
if [ -d "$HOME/.local/bin" ]; then
  PATH="$HOME/.local/bin:$PATH"
fi
export PATH

# Ajalugu.
HISTSIZE=5000
HISTFILESIZE=10000
HISTCONTROL=ignoredups:erasedups
shopt -s histappend

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'

# Otsing ja failimaht.
alias h='history | tail -n 20'
alias dus='du -sh * 2>/dev/null | sort -h'
alias biggest='du -a . 2>/dev/null | sort -nr | head'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Prompt: kasutaja, host ja praegune kataloog.
PS1='\u@\h:\w\$ '
```

~/.zshrc

```
# ~/.zshrc
# Tase 2: lühike igapäevane zsh seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  path=("$HOME/bin" $path)
fi
if [ -d "$HOME/.local/bin" ]; then
  path=("$HOME/.local/bin" $path)
fi
export PATH

# Ajalugu.
HISTFILE="$HOME/.zsh_history"
HISTSIZE=5000
SAVEHIST=10000
setopt APPEND_HISTORY
setopt SHARE_HISTORY
setopt HIST_IGNORE_DUPS
setopt HIST_REDUCE_BLANKS

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'

# Otsing ja failimaht.
alias h='history | tail -n 20'
alias dus='du -sh * 2>/dev/null | sort -h'
alias biggest='du -a . 2>/dev/null | sort -nr | head'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Prompt: kasutaja, host ja praegune kataloog.
PROMPT='%n@%m:%~%# '
```

Tase 3: mugavam prompt ja rohkem alias'eid

See tase lisab värvilise prompti, parema `less` käitumise, Git-i lühikäsud ja paar väikest abifunktsiooni.

~/.bashrc

```
# ~/.bashrc
# Tase 3: mugavam bash'i seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  PATH="$HOME/bin:$PATH"
fi
if [ -d "$HOME/.local/bin" ]; then
  PATH="$HOME/.local/bin:$PATH"
fi
export PATH

# Ajalugu.
HISTSIZE=10000
HISTFILESIZE=20000
HISTCONTROL=ignoreboth:erasedups
shopt -s histappend
shopt -s checkwinsize

# Less ja värvid.
export LESS='-R'
export CLICOLOR=1

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'
alias ltr='ls -ltr | tail'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'
alias home='cd "$HOME"'

# Failid ja kettaruum.
alias dus='du -sh * 2>/dev/null | sort -h'
alias dush='du -sh .[!.]* * 2>/dev/null | sort -h'
```

```
alias biggest='du -a . 2>/dev/null | sort -nr | head'

# Ajalugu ja käsu leidmine.
alias h='history | tail -n 30'
alias path='printf "%s\n" ${PATH//:/ }'
alias whichall='type -a'

# Git.
alias gs='git status --short'
alias gd='git diff'
alias gdc='git diff --cached'
alias gl='git log --oneline --decorate -10'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Abifunktsioon: tee kataloog ja mine sinna sisse.
mkcd() {
  mkdir -p "$1" && cd "$1"
}

# Värviline prompt: kasutaja@host:kataloog.
PS1='\[\033[32m\]\u@\h\[\033[0m\]:\[\033[34m\]\w\[\033[0m\]\$ '
```

~/.zshrc

```
# ~/.zshrc
# Tase 3: mugavam zsh seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  path=("$HOME/bin" $path)
fi
if [ -d "$HOME/.local/bin" ]; then
  path=("$HOME/.local/bin" $path)
fi
export PATH

# Ajalugu.
HISTFILE="$HOME/.zsh_history"
HISTSIZE=10000
SAVEHIST=20000
setopt APPEND_HISTORY
setopt SHARE_HISTORY
```

```
setopt HIST_IGNORE_DUPS
setopt HIST_REDUCE_BLANKS

# Less ja värvid.
export LESS='-R'
export CLICOLOR=1
autoload -Uz colors && colors

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'
alias ltr='ls -ltr | tail'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'
alias home='cd "$HOME"'

# Failid ja kettaruum.
alias dus='du -sh * 2>/dev/null | sort -h'
alias dush='du -sh .[!.]* * 2>/dev/null | sort -h'
alias biggest='du -a . 2>/dev/null | sort -nr | head'

# Ajalugu ja käsu leidmine.
alias h='history | tail -n 30'
alias path='printf "%s\n" ${(ps.:.)PATH}'
alias whichall='type -a'

# Git.
alias gs='git status --short'
alias gd='git diff'
alias gdc='git diff --cached'
alias gl='git log --oneline --decorate -10'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Abifunktsioon: tee kataloog ja mine sinna sisse.
mkcd() {
  mkdir -p "$1" && cd "$1"
}
```

```
# Värviline prompt: kasutaja@host:kataloog.
PROMPT='%F{green}%n@%m%f:%F{blue}%~%f%# '
```

Tase 4: arendajale

See tase lisab projektitöö, Git-i, Pythoni, Dockeri ja tekstivormingute abikäsud. Kopeeri see alles siis, kui eelnevad tasemed tunduvad arusaadavad.

~/.bashrc

```
# ~/.bashrc
# Tase 4: arendajale suunatud bash'i seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  PATH="$HOME/bin:$PATH"
fi
if [ -d "$HOME/.local/bin" ]; then
  PATH="$HOME/.local/bin:$PATH"
fi
export PATH

# Ajalugu.
HISTSIZE=20000
HISTFILESIZE=50000
HISTCONTROL=ignoreboth:erasedups
shopt -s histappend
shopt -s checkwinsize

# Less, editor ja värvid.
export LESS='-R'
export EDITOR="${EDITOR:-nano}"
export CLICOLOR=1

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'
alias ltr='ls -ltr | tail'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'
```

```
alias home='cd "$HOME"'
alias proj='cd "$HOME/projects"'

# Failid, otsing ja kettaruum.
alias dus='du -sh * 2>/dev/null | sort -h'
alias dush='du -sh .[!.]* * 2>/dev/null | sort -h'
alias biggest='du -a . 2>/dev/null | sort -nr | head'
alias ff='find . -type f -name'
alias serve='python3 -m http.server 8000'

# Ajalugu ja PATH.
alias h='history | tail -n 40'
alias path='printf "%s\n" ${PATH//:/ }'
alias whichall='type -a'

# Git.
alias gs='git status --short'
alias gd='git diff'
alias gdc='git diff --cached'
alias ga='git add'
alias gc='git commit'
alias gl='git log --oneline --decorate -15'
alias gb='git branch'
alias gco='git checkout'
alias gsw='git switch'

# Python.
alias py='python3'
alias venv='python3 -m venv .venv'
alias va='source .venv/bin/activate'
alias pipup='python3 -m pip install --upgrade pip'

# Docker.
alias dps='docker ps'
alias di='docker images'
alias dc='docker compose'
alias dcu='docker compose up --build'
alias dcd='docker compose down'
alias dcl='docker compose logs -f'

# Tekstivormingud.
alias json='python3 -m json.tool'
alias csvlook='column -s, -t'

# Ohutumad vaikekujud.
alias cp='cp -i'
```

```
alias mv='mv -i'
alias rm='rm -i'

# Abifunktsioon: tee kataloog ja mine sinna sisse.
mkcd() {
  mkdir -p "$1" && cd "$1"
}

# Abifunktsioon: loo Python venv, kui seda veel ei ole.
venvup() {
  if [ ! -d .venv ]; then
    python3 -m venv .venv
  fi
  . .venv/bin/activate
}

# Git haru prompti jaoks.
git_branch() {
  git branch --show-current 2>/dev/null
}

# Prompt: kasutaja@host:kataloog (git-haru).
PS1='\[\033[32m\]\u@\h\[\033[0m\]:\[\033[34m\]\w\[\033[0m\] \[\033[33m\]$(git_branch)\[\033
```

~/.zshrc

```
# ~/.zshrc
# Tase 4: arendajale suunatud zsh seadistus.

# PATH: kasutaja enda käsud.
if [ -d "$HOME/bin" ]; then
  path=("$HOME/bin" $path)
fi
if [ -d "$HOME/.local/bin" ]; then
  path=("$HOME/.local/bin" $path)
fi
export PATH

# Ajalugu.
HISTFILE="$HOME/.zsh_history"
HISTSIZE=20000
SAVEHIST=50000
setopt APPEND_HISTORY
setopt SHARE_HISTORY
setopt HIST_IGNORE_DUPS
setopt HIST_REDUCE_BLANKS
```

```
# Less, editor ja värvid.
export LESS='-R'
export EDITOR="${EDITOR:-nano}"
export CLICOLOR=1
autoload -Uz colors && colors

# Elementaarsed käsud.
alias ll='ls -lh'
alias la='ls -la'
alias l='ls -CF'
alias lt='ls -lt | head'
alias ltr='ls -ltr | tail'

# Navigeerimine.
alias ..='cd ..'
alias ...='cd ../..'
alias c='clear'
alias home='cd "$HOME"'
alias proj='cd "$HOME/projects"'

# Failid, otsing ja kettaruum.
alias dus='du -sh * 2>/dev/null | sort -h'
alias dush='du -sh .[!.]* * 2>/dev/null | sort -h'
alias biggest='du -a . 2>/dev/null | sort -nr | head'
alias ff='find . -type f -name'
alias serve='python3 -m http.server 8000'

# Ajalugu ja PATH.
alias h='history | tail -n 40'
alias path='printf "%s\n" ${(ps.:.)PATH}'
alias whichall='type -a'

# Git.
alias gs='git status --short'
alias gd='git diff'
alias gdc='git diff --cached'
alias ga='git add'
alias gc='git commit'
alias gl='git log --oneline --decorate -15'
alias gb='git branch'
alias gco='git checkout'
alias gsw='git switch'

# Python.
alias py='python3'
```

```
alias venv='python3 -m venv .venv'
alias va='source .venv/bin/activate'
alias pipup='python3 -m pip install --upgrade pip'

# Docker.
alias dps='docker ps'
alias di='docker images'
alias dc='docker compose'
alias dcu='docker compose up --build'
alias dcd='docker compose down'
alias dcl='docker compose logs -f'

# Tekstivormingud.
alias json='python3 -m json.tool'
alias csvlook='column -s, -t'

# Ohutumad vaikekujud.
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'

# Abifunktsioon: tee kataloog ja mine sinna sisse.
mkcd() {
  mkdir -p "$1" && cd "$1"
}

# Abifunktsioon: loo Python venv, kui seda veel ei ole.
venvup() {
  if [ ! -d .venv ]; then
    python3 -m venv .venv
  fi
  . .venv/bin/activate
}

# Git haru prompti jaoks.
setopt PROMPT_SUBST
git_branch() {
  git branch --show-current 2>/dev/null
}

# Prompt: kasutaja@host:kataloog (git-haru).
PROMPT='%F{green}%n@%m%f:%F{blue}%~%f %F{yellow}$(git_branch)%f%# '
```

Kuidas viga tagasi võtta

Kui pärast seadistusfaili muutmist terminal käitub valesti, ava uus terminal või käivita shell ilma seadistusfailita:

```
bash --noprofile --norc
```

`zsh` puhul saad ajutiselt käivitada:

```
zsh -f
```

Seejärel taasta varukoopia:

```
cp ~/.zshrc.backup ~/.zshrc
cp ~/.bashrc.backup ~/.bashrc
```

Kui varukoopiat ei olnud, ava fail redaktoris ja kommenteeri viimati lisatud read välja:

```
nano ~/.zshrc
nano ~/.bashrc
```

Kommentaar algab märgiga `#`. Shell ignoreerib sellist rida.

Minitest

1. Vaata käsuga `echo "$SHELL"`, kas kasutad praegu `bash`-i või `zsh`-i.
2. Tee oma seadistusfailist varukoopia.
3. Lisa kõige lihtsam tase ja lae fail `source` käsuga uuesti sisse.
4. Proovi käske `ll`, `la` ja `h`.
5. Selgita oma sõnadega, miks ei tasu korraga kopeerida mitut taset samasse faili.

Peatüki täisspikker

Tase: **Referents**

Eesmärk: See lisa annab kopeeritavad algusfailid kahe levinud shelli jaoks:

Põhikujud

- `echo "$SHELL"` — prindi tekst
- `ls -la "$HOME"` — pikk + peidetud
- `cp ~/.zshrc ~/.zshrc.backup` — kopeeri
- `cp ~/.bashrc ~/.bashrc.backup` — kopeeri
- `alias ll='ls -lh'` — lühinimi
- `alias la='ls -la'` — lühinimi
- `mv` — liiguta/nimeta
- `rm` — kustuta

Olulisemad lipud, märgid ja kiirnupud

- `;` — järjest
- `|` — toru edasi
- `2>` — vead faili
- `-h` — lühike abi
- `&&` — ainult õnnestumisel

Pane tähele: Ära pane kõiki tasemeid korraga samasse faili. Alusta ühest tasemest ja lisa hiljem juurde.

Edasi: Selle osa viimase peatüki järel tasub vaadata järgmise osa algust või osa-spikrit.

Osa PDF: ./spikrid/lisad-spikker.pdf