

PRAKTIINE TEATMETEOS

Linux ja macOS käsura õpik

Autor: ChatGPT

Teda abistas: Jaak Vilo

*Beetaversioon: sisu on enamasti terviklik ja avalikuks katsetamiseks valmis,
kuid vajab veel tehnilist, keelelist ja kasutatavuse kontrolli.*

Kolofon

Linux ja macOS käsura õpik

Autor: ChatGPT

Teda abistas: Jaak Vilo

Beetaversioon: sisu on enamasti terviklik ja avalikuks katsetamiseks valmis, kuid vajab veel tehnilist, keelelist ja kasutatavuse kontrolli.

Versioon: v0.3.0-beta.1

Kood: v0.3.0-beta.1-21-g1a17054

Versioonikirje (commit): 1a17054

Märgend (tag): v0.3.0-beta.1

Kuupäev: 2026-10-02 19:25:09 CEST

Sisukord

Osa I: Esimesed sammud

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

Osa II: Süsteemi pilt ja haldus

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

Osa III: Failid, võrk ja süsteemitöö

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

Osa IV: Tekst, otsing ja automatiseerimine

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

- Esimene shelliskript
- `cron` ja ajastatud tööd

Osa V: Arendus ja töövood

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

Lisad

- Lisa A: kopeeritavad minitestid
- Lisa B: spikrite register
- Lisa C: sõnastik ja terminoloogia
- Lisa D: edasised võimalused ja lugejate soovid
- Lisa E: usaldusväärsed viited ja lisalugemine
- Lisa F: shelli seadistusfailid `bash` ja `zsh` jaoks

Kuidas seda õpikut kasutada

Õpi käsureal tegema nähtava tulemusega tööd: leia failid, töötle teksti, kasuta kaugmasinat ja automatiseeri korduvaid samme. Õpik sobib nii järjest õppimiseks kui ka hilisemaks käsiraamatuks.

Vali oma algus

- **Täiesti algaja:** alusta terminali esimestest sammudest. Esimesel korral piisab 20–30 minutist: vaata infot, liigu kataloogide vahel ja loo oma harjutuskoht. Giti, Dockerit ega süsteemihaldust ei pea veel tundma.
- **Põhikäskudega tuttav:** vali õpiteede hulgast oma eesmärk, näiteks failihaldus, tekstitöötlus või arendus.
- **Otsid kiiret vastust:** kasuta peatükkide lõpu spikreid ja spikrite registrit. Enne muutvat käsku loe ka selle hoiatus.

Harjutamiseks sobib oma Linuxi või macOS-i terminal, Windowsis WSL-i Linuxi shell ning brauseris `terminal.cs.ut.ee`. Brauserikeskkonnas on ka valmis ülesanded.

CLI ja GUI

CLI (*command-line interface*) on **käsurealiides**: annad juhiseid tekstina. **GUI** (*graphical user interface*) on **graafiline kasutajaliides**: kasutad aknaid, nuppe ja menüüsid. Need on eri viisid sama töö tegemiseks, mitte paremate ja halvemate kasutajate tunnused.

Käsureal suhtled tavaliselt terminali kaudu **shell**i ehk **käsitõlgiga**. Nende rollid ja shelli kuvatav viip saavad selgeks esimeste käskude juures.

Kuidas näidet kasutada

1. Loe, mida käsk teeb ning kas ta muudab faile.
2. Katseta oma harjutuskataloogis. Kopeeri käsk, mitte viip või näidisväljund.
3. Võrdle tulemust seletusega. Failiteed, kuupäevad ja kasutajanimed võivad sinu arvutis erineda.
4. Tee minitest: muuda sisendit või ennusta väljundit enne käivitamist.

Kui ühes näites on Linuxi ja macOS-i alternatiivid, **vali ainult oma süsteemi oma**. Nimed nagu **kasutaja@server** ja **fail.txt** tuleb sobitada oma ühenduse või olemasoleva failiga. Ära käivita tundmatut koodi pelgalt sellepärast, et see on kopeeritav.

Täpsed mõisted leiad sõnastikust, põhjalikuma dokumentatsiooni lisalugemise viidetest.

Spikker

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 ajutiste failide kataloogi; sobib harjutamiseks

Põhimõisted

- CLI — käsurida

- GUI — graafile line liides
- `terminal` — aken käsurreaks
- `shell` — tõlgendab kāske

Pane tähele: Sa ei pea esimeses ringis veel teadma Giti, Dockerit ega keerulismaid töövooge.

Õpatee ja õppetunnid

Vali rada oma seniste oskuste ja eesmärgi järgi. Algajana läbi kolm esimest etappi järjest, seejärel loo süsteemipilt. Ülejäänud teemadest vali vajalik; kõiki radu ei pea läbima.

Loe peatükki, katseta selle näiteid ja tee raja lõpus harjutus. Kasuta uut harjutuskataloogi, et mitte oma faile üle kirjutada. Kui jääd harjutusega hätta, korda vastavat peatükki.

Õpatee 1: täiesti algaja

Alusta siit, kui käsuri on uus. Loe igas etapis peatükke antud järjekorras ja katseta näiteid oma harjutuskataloogis. Kõike ei pea ühe korraga läbima.

1. Tutvu ja liigu

Eeldused: käsurrea kogemust pole vaja; saad avada terminali või kasutada brauseriterminali.

1. Terminali esimesed sammud
2. Abi leidmine
3. Kataloogid ja failid

Harjuta: tee failide ja kataloogide minitest esimesed seitse sammu: loo harjutuskoht, kopeeri ja nimeta fail ümber ning võrdle peidetud failide kuvamist. Kustutamist pole selles harjutuses vaja.

Nüüd oskad: leida oma asukoha, liikuda kataloogides, luua harjutuskoha ja avada käsu juhendi.

2. Tööta failidega

Eeldused: oskad liikuda kataloogides ja luua faili.

1. Teksti vaatamine ja liikumine
2. Failide muutmine
3. Käskude kuju ja argumentide loogika

Harjuta: tee tekstiredaktori minitest: muuda näidisfaili, salvesta ning proovi ka salvestamata väljumist.

Nüüd oskad: faili lugeda ja muuta ning eristada käsu nime, valikuid ja argumente.

3. Ühenda oskused

Eeldused: oskad faili lugeda ja muuta ning tunned argumente ja jutumärke.

1. Sisend, väljund, torud ja suunamine
2. Esimene tervikharjutus: 30 minutit

Harjuta: lahenda tervikharjutuse iseseisev ülesanne: lisa koopiasse rida, kuva faili lõpp ja kontrolli, et originaal ei muutunud.

Nüüd oskad: salvestada väljundi faili, ühendada käske ja kontrollida oma töö tulemust.

Kui mõni kontrollpunkt jäi ebakindlaks, korda vastava etapi näidet. Muidu jätkka allpool süsteemipildi loomisega; Giti, Dockerit ja süsteemihaldust ei pea enne põhioskusi õppima.

Õpitee 2: süsteemipildi loomine

Eeldused: algajaraja kolm etappi on läbitud. See on järgmine ühine alus enne spetsialiseerumist.

1. Linux, Unix, GNU, macOS, Windows ja shellid
2. Failisüsteemi kaart
3. Kettaruum ja süsteemi maht
4. Õigused, omanikud ja täitmisbitid
5. Kasutajad, grupid ja sudo
6. Muutujad, keskkond, PATH ja aliases
7. Paketihaldus: apt, dnf, pacman, brew
8. Lihtne veaotsing käsureal
9. Võrgu põhitooriistad

Harjuta: tee veaotsingu minitest: kontrolli käsu leitavust, tekita olematu failinimega ohutu viga ja koosta täpne veakirjeldus.

Nüüd oskad: selgitada failiteid, õigusi ja käskude leitavust ning valida oma süsteemile sobiva kontrollkäsu enne parandamist.

Õpitee 3: igapäevane Linuxi ja serveri kasutaja

Eeldused: tunned failiteid, õigusi ja võrgu põhikontrolle. SSH katsetamiseks vajad lubatud ligipääsu serverile; kopeerimist ja arhiive saad harjutada ka oma arvutis.

1. Kauglogimine ja SSH
2. Failide kopeerimine ja sünkroonimine
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

Harjuta: läbi koopia taastamise näide: paki harjutusfailid, taasta eraldi kataloogi ja võrdle sisu. Selleks pole serverit vaja.

Nüüd oskad: kopeerida ja taastada faile, kasutada SSH-võtmeid ning uurida protsesse ja logisid.

Õpitee 4: tekst, filtrid ja automatiseerimine

Eeldused: tunned torusid, suunamist, jutumärke, muutujaid ja faili käivitamisõigust.

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

Harjuta: alusta teksti filtreerimise ja loendamise minitestist, seejärel tee skripti minitest eri argumentidega. Ajasta alles käsitsi kontrollitud töö.

Nüüd oskad: koostada tekstifiltri, kirjutada argumentidega skripti ning eristada edukat lõppu veast.

Õpitee 5: arendaja suund

Eeldused: oskad faile muuta, programme käivitada ning lugeda õigusi ja keskkonnamuutujaid. Arendustööriistad paigalda oma süsteemi juhendi järgi.

1. Git, GitHub ja töövoog
2. Pythoni venv ja eraldatud keskkonnad
3. Dockeri alused
4. IDE-d ja arenduskeskkonnad
5. Kompileerimine ja käivitamine: shell, Python, C, C++, Go, Rust, Java

Harjuta: tee Giti lõppharjutus: loo haru, muuda faili, vaata erinevusi ja salvesta versioonikirje. Seejärel proovi Pythoni virtuaalkeskkonda.

Nüüd oskad: hoida muudatusi Gitis, eraldada projekti sõltuvusi ja valida sobiva käivitusviisi.

Dokumentide koostamiseks jätkka peatükiga LaTeX käsuraalt. Andmevormingute ja SQL-i jaoks vali järgmine rada.

Õpitee 6: andmeteaduse stardirada

Eeldused: oskad faile lugeda ja muuta ning tunned torusid ja suunamist. Pythoni näited eeldavad ka selle keele algteadmisi.

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. Pythoni venv ja eraldatud keskkonnad
8. Andmebaasi algus: sqlite ja Python

Harjuta: tee andmevormingute minitest ja SQLite'i minitest: vali andmetest vajalikud väljad ja read ning muuda päringu tingimust.

Nüüd oskad: valida CSV, JSON-i või XML-i jaoks sobiva tööriista, teha SQL-päringu ja kontrollida tulemust.

Kiirtee: varasemate oskuste värskendamiseks

Eeldused: oled käsurida varem kasutanud ning tunned argumente ja jutumärke. See rada tuletab oskused meelde, mitte ei asenda algaja põhiteed.

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

Harjuta: lahenda Giti minitest ja torude minitest näidiskäsked vaatamata. Loe uuesti ainult neid osi, mille vastuses kahtled.

Nüüd oskad: taastada oma töövõtted ning märgata, millised failihalduse, serveritöö või Giti oskused vajavad kordamist.

Millal kasutada raamatut teatmeteosena

Kui töövõte on juba tuttav, piisab spikrist. Kui vajad seletust, ava vastav peatükk. Lisakatseteks sobivad kopeeritavad minitestid; süvenemiseks lisalugemine teemade kaupa.

Spikker

Eesmärk: Vali eeldustele vastav rada, loe peatükke järjest ja kontrolli oskusi harjutusega.

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, koostamine
- Andmeteaduse stardirada — failid, vormingud, SQL

Kuidas õppida

- Eeldused — mida peaksid enne oskama
- Lugemisjärjekord — liigu loendis ülalt alla
- Harjutus — proovi uut töövõtet eraldi katsekataloogis
- Nüüd oskad — kontrolli, kas saad ülesandega iseseisvalt hakkama

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

Terminali esimesed sammud

Alustame käskudest, mis ainult näitavad infot. Seejärel loome oma harjutuskataloogi ja õpime nimesid vähem trükkima.

Terminal, shell ja viip

Terminal on aken või rakendus, kus sisestad teksti ja näed väljundit. **Shell** ehk käsutõlk on selles töötav programm, näiteks Bash või zsh: ta tõlgendab käsurida ning käivitab programmi või oma sisekäsu. Sama terminalirakendusega saab kasutada erinevaid shelle.

Viip ehk *prompt* näitab, et shell ootab järgmist käsku. Näites on viip kasutaja@arvuti:~\$ ja sisestatud käsk pwd:

```
kasutaja@arvuti:~$ pwd
/home/kasutaja
```

Viip võib näidata kasutajat, arvuti nime ja kataloogi; ~ tähistab kodukataloogi. Selle välimus on muudetav ega tõenda iseenesest sinu asukohta või õigusi.

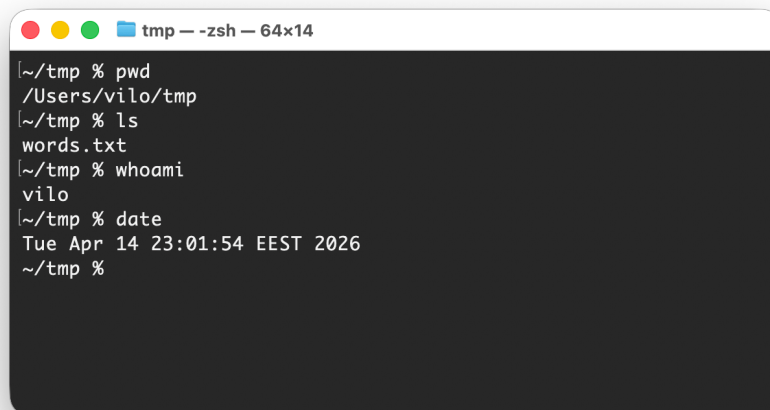
Ära kopeeri viipa kaasa. Sisesta ainult käsk, siin `pwd`, ja vajuta `Enter`. Väljundiridu terminali tagasi ei sisestata.

Esimesed ohutud käsud

Proovi tabeli kāske ükshaaval. Need ei muuda faile.

Kāsk	Mida nāed
<code>pwd</code>	praeguse kataloogi täielik tee
<code>ls</code>	selle kataloogi failide ja alamkataloogide nimed
<code>whoami</code>	sinu kasutajanimi
<code>date</code>	praegune kuupäev ja kellaaeg

Kui `ls` ei väljasta midagi, võib kataloog olla tühi või sisaldada ainult peidetud kirjeid.



```
tmp — -zsh — 64x14
[~/tmp % pwd]
/Users/vilo/tmp
[~/tmp % ls]
words.txt
[~/tmp % whoami]
vilo
[~/tmp % date]
Tue Apr 14 23:01:54 EEST 2026
~/tmp %
```

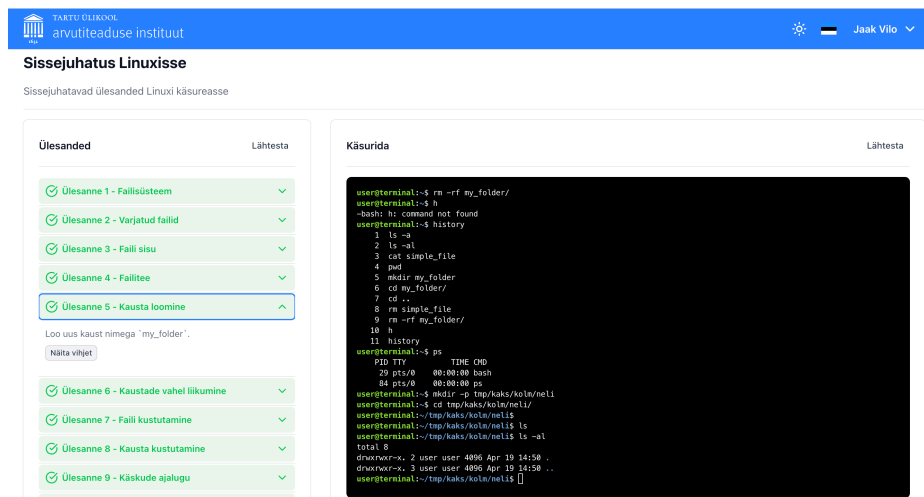
Joonis 1: Terminalis näitavad `pwd`, `ls`, `whoami` ja `date` asukohta, kataloogi sisu, kasutajat ning aega.

1. `pwd` väljastab töökoha tee.
2. `ls` loetleb seal olevad nimed, mitte failide sisu.
3. `whoami` ja `date` vastavad kumbki ühe reaga; uus viip näitab, et käsk on lõpetanud.

Valmis harjutuskeskkond

terminal.cs.ut.ee pakub terminali ja sissejuhatavaid ülesandeid brauseris. Seal saad alustada ka oma arvutisse Linuxit paigaldamata.

Vali vasakult ülesanne, sisesta parempoolsesse terminali käsk ning võrdle tulemust ülesande nõudega. Veebikeskkonna failid ei ole sinu enda arvuti failid.



Joonis 2: Tartu Ülikooli harjutuskeskkonna ülesanded ja terminal.

Käsu nimi ja argument

Käsuresas `ls /etc` on **ls** käsu nimi ja **/etc** argument: kataloog, mille sisu soovid näha. `ls` üksi vaatab praegust kataloogi. Lisa `-l` annab pikema loendi, kus on ka õigused, suurus ja muutmisaeg:

```
ls /etc
ls -l /etc
```

`-l` on **valik ehk lipp**, mitte failinimi. Enamasti kirjutatakse kõigepealt käsk, siis valikud ja seejärel argumentid. Üksikasjad tulevad peatükis Käskude kuju ja argumentide loogika.

Liikumine ja esimene oma kataloog

`cd` muudab shelli töökataloogi, mitte kataloogi asukohta kettal.

Kuju	Tulemus
<code>cd ~</code>	mine kodukataloogi
<code>cd ..</code>	mine ülemkataloogi
<code>cd /</code>	mine juurkataloogi
<code>cd proov</code>	mine praeguse kataloogi alamkataloogi proov

mkdir loob kataloogi. Tee kodukataloogi oma harjutuskoht:

```
cd ~
mkdir proov
```

```
cd proov
pwd
ls
```

`pwd` väljund lõpeb nüüd `/proov`. `mkdir` ja `cd` on õnnestudes tavaliselt vaiksed; tühi `ls` väljund kinnitab, et uues kataloogis pole veel nähtavaid faile. Kui nimi `proov` on juba kasutusel, vali uus nimi ja kasuta seda ka `cd` käsus.

Tab aitab pikki nimesid lõpetada

Loo harjutuskataloogi kaks alamkataloogi:

```
mkdir pildid-loodus pildid-linn
```

Kirjuta käsureale `cd pil`, kuid ära vajuta veel **Enter**. Vajuta **Tab**. Shell täiendab nime ühise alguseni või näitab võimalikke vasteid. Kui real on nüüd `cd pildid-l`, lisa `ood` ja vajuta uuesti **Tab**: alles jääb üks vaste, `pildid-loodus`. Kui automaatset täiendust ei tulnud, kirjuta ise `cd pildid-lood` ja proovi uuesti. Alles täieliku nime järel vajuta **Enter** ja kontrolli asukohta käsuga `pwd`.

Tab on klahv: sõna **Tab** ega märgendeid `<Tab>` käsureale ei kirjutata. Kui midagi ei juhtu, võib vasteid olla mitu või mitte ühtegi. Kontrolli kirjaipilti, lisa mõni täht või vajuta **Tab** uuesti. Täpne käitumine sõltub shelli seadistusest.

Ajalugu, abi ja katkestamine

history kuvab käsuajaloo. Ülesnool toob varasema käsu muutmiseks ette; al-lanool liigub uuema poole. **Ctrl-r** alustab ajaloootsingut: kirjuta osa käsust. **Kontrolli leitud käsk enne Enter-it üle**. Ajaloo pikkus sõltub varasemast kasutusest ja shelli seadistusest.

man ls avab käsu `ls` käsiraamatu. Välju klahviga **q**. Kui käsiraamatut ei leita, vaata järgmise peatüki abijuhist.

Klahv	Toiming tavapärasel Bashi või zsh käsureadaktoris
Ctrl-c	katkesta programmi töö või pooleli olev sisestus
Ctrl-a / Ctrl-e	liigu käsurea algusse / lõppu
Ctrl-k	löika tekst kursorist rea lõpuni

Need käsurea liikumisklahvid võivad eriseadistuse korral erineda. **Ctrl-c** ei võta juba tehtud failimuudatusi tagasi.

Ära käivita veebist või vestlusest kopeeritud käsku, mille mõtet sa ei tea. Eriti ohtlik on alla laaditud koodi kohe käivitada: programm saab kasutada sinu õigusi ja lugeda sinu faile.

Minitest

1. Mille poolest erinevad **pwd** ja viibas olev katalooginimi?
2. Kas **ls /etc** muudab sinu töökohta? Kontrolli enne ja pärast käsuga **pwd**.
3. Liigu loodud pildikataloogist ülemkataloogi ja vali **Tab**-iga teine pildikataloog.
4. Too ajaloost ette **ls /etc**, lisa valik **-l** ning võrdle tulemust.

Spikker

Eesmärk: Vaata kõigepealt rahulikult, kus sa oled, kes sa oled ja mis selles kataloogis 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 ajutiste failide kataloogi; sobib harjutamiseks
- **mkdir proov** — loo uus kataloog
- **cd proov** — sisene loodud kataloogi

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.

Abi leidmine: man, --help ja info

Käsu kohalik dokumentatsioon kirjeldab tavaliselt just sinu arvutisse paigaldatud varianti. Alusta sellest, mitte juhusliku veebinäite kopeerimisest.

Vali sobiv abi

Kuju	Mida saada
man ls	käsu ls lehitsetav käsiraamat
ls --help	GNU ls kasutusjuhised terminalis; macOS-i ls seda valikut ei toeta

Kuju	Mida saad
whatis ls	käsiraamatu lühikirjeldus
apropos copy	märksõnale copy vastavad käsiraamatukirjed
info coreutils	GNU põhikäskude pikem dokumentatsioon, kui see on paigaldatud

--help on levinud, **mitte universaalne valik**. Ka -h ei tähenda alati abi: näiteks **ls -h** muudab suuruste esitusviisi. Suur- ja väiketäht on eri valikud.

Esimene käsiraamat

Ava **man ls**. Alguses näed tavaliselt pealkirja **LS(1)**: käsu nimi ning käsiraamatu jaotis, siin kasutajakäsud. Jaotis **NAME** ütleb lühidalt, mida käsk teeb; **SYNOPSIS** näitab süntaksit ja **DESCRIPTION** selgitab valikuid.

Tavapärases lehitsejas saad liikuda nii:

Klahv	Toiming
tühik / b	lehekülge edasi / tagasi
/-a ja Enter	otsi valikut -a
n	järgmine otsinguvaste
q	välju tagasi käsureale

Otsi **-a** seletus üles, välju ja võrdle **ls** ning **ls -a** väljundit. Nii kasutad dokumentatsiooni konkreetse küsimuse lahendamiseks.

Kui abi ei leita

Kui **man** puudub või ütleb **No manual entry**, võib puududa käsiraamatupakett, mitte käsk ise. **whatis** ja **apropos** kasutavad samuti kohalikku dokumentatsiooniandmebaasi; need ei asenda puuduvat juhendit.

Proovi käsu dokumenteeritud abivalikut või ava ametlik juhend. Vajadusel paigalda hiljem dokumentatsioon paketi halduriga. Ära proovi tundmatu käsu valikuid juhuslikult: mõni neist võib midagi muuta.

Shelli **sisekäsu** abi võib olla shelli enda juhendis. Näiteks Bashis selgitab **help cd** käsku **cd**; zsh-s vaata **man zshbuiltins**. **help** ei ole kõigi shellide ühine käsk.

Kui käsu nime ei mäleta

`apropos archive` otsib käsiraamatute kirjeldustest sõna **archive**. Tulemus on kandidaatide loend, mitte soovitus kõiki käske käivitada. Vali nime ja lühikirjelduse järgi sobiv kirje ning ava selle **man**-leht.

GNU tööriistade põhjalikum juhend võib olla **info**-vormingus. Käsk **info coreutils** avab selle, kui dokumentatsioon on olemas; **h** näitab kasutusabi ja **q** väljub.

Minitest

1. Leia **man ls** abil, mida teeb valik **-l**, ning proovi seda.
2. Mida tähendab pealkirjas **LS(1)** olev number?
3. Leia **apropos**-iga kopeerimisega seotud käsk. Kas selle nimi üksi ütleb, mida see kopeerib?
4. Miks pole **-h** turvaline universaalne viis abi küsimiseks?

Spikker

Eesmärk: Leia tundmatu käsu tähendus ja oma süsteemis toetatud valikud.

Käsud ja näited

- **man ls** — lehitsetav käsiraamat; alusta jaotistest **NAME** ja **SYNOPSIS**
- **ls --help** — GNU **ls**-i kiire abi; macOS-i **ls** kasutab **man**-lehte
- **whatis ls** — käsu ühe rea kirjeldus
- **apropos archive** — otsi käsiraamatute kirjeldustest
- **info coreutils** — GNU põhikäsud, kui dokumentatsioon on paigaldatud

Märgid ja valikud

- **q** — välju **man**-i lehitsejast
- **/tekst** — otsi **man**-lehelt; kinnita Enteriga
- **n / N** — järgmine / eelmine otsinguvastus

Pane tähele: **-help** ega **-h** pole universaalsed; kontrolli konkreetse programmi juhendit.

Kataloogid ja failid

Unix-laadses süsteemis taandub suur osa tööst failide ja kataloogidega tegelemisele. Selles peatükis teed esimesed tegelikud muudatused, kuid kontrollitult.

Kiire orientiir

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

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

Tüüpilised algaja vead

- `pwd` jääb vaatamata ja fail luuakse valesse kohta
- tühikuga 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 kustutamise- ja liigutamiskäskke päris projektikataloogis. Tee eraldi harjutuskoht.

```
pwd
mkdir -p ~/tmp/faali-naited
cd ~/tmp/faali-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
<code>.</code>	praegune kataloog
<code>..</code>	ülemine kataloog
<code>~</code>	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.

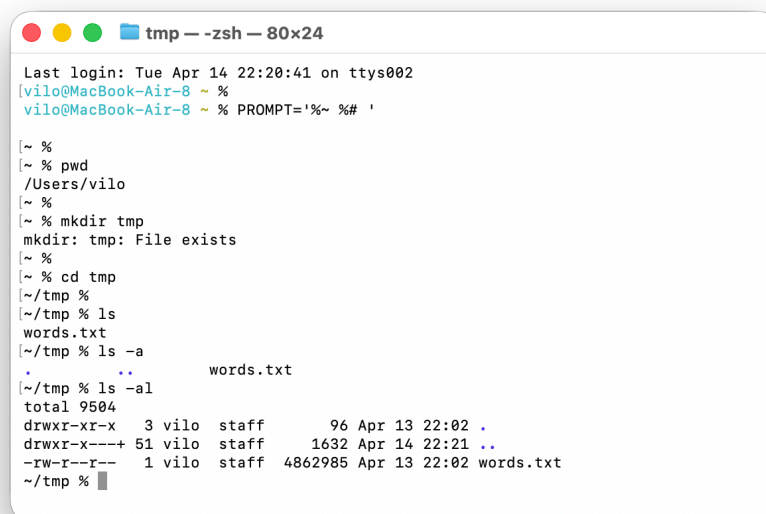
Kataloogi sisu vaatamine

`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



```
tmp — -zsh — 80x24
Last login: Tue Apr 14 22:20:41 on ttys002
[vilo@MacBook-Air-8 ~ %]
[vilo@MacBook-Air-8 ~ % PROMPT='%~ %# ' ]
[~ %]
[~ % pwd]
/Users/vilo
[~ %]
[~ % mkdir tmp]
mkdir: tmp: File exists
[~ %]
[~ % cd tmp]
~/tmp %
~/tmp % ls
words.txt
~/tmp % ls -a
.  ..  words.txt
~/tmp % ls -al
total 9504
drwxr-xr-x  3 vilo  staff    96 Apr 13 22:02 .
drwxr-xr-x+ 51 vilo  staff  1632 Apr 14 22:21 ..
-rw-r--r--  1 vilo  staff 4862985 Apr 13 22:02 words.txt
~/tmp %
```

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

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

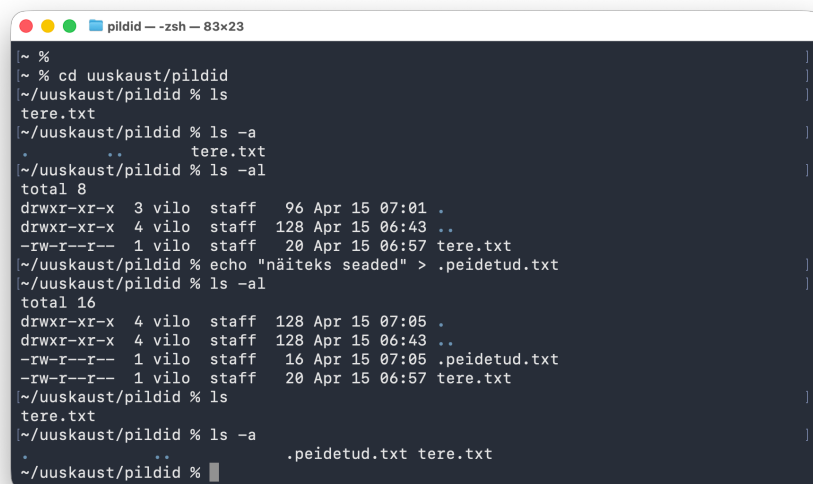
Punktiga algavad nimed

Punktiga algavad nimed on tavaliselt peidetud:

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

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

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



```
pildid -- zsh -- 83x23
~ %
~ % cd uuskaust/pildid
~/uuskaust/pildid % ls
tere.txt
~/uuskaust/pildid % ls -a
.  ..  tere.txt
~/uuskaust/pildid % ls -al
total 8
drwxr-xr-x  3 vilo  staff   96 Apr 15 07:01 .
drwxr-xr-x  4 vilo  staff  128 Apr 15 06:43 ..
-rw-r--r--  1 vilo  staff   20 Apr 15 06:57 tere.txt
~/uuskaust/pildid % echo "näiteks seaded" > .peidetud.txt
~/uuskaust/pildid % ls -al
total 16
drwxr-xr-x  4 vilo  staff  128 Apr 15 07:05 .
drwxr-xr-x  4 vilo  staff  128 Apr 15 06:43 ..
-rw-r--r--  1 vilo  staff   16 Apr 15 07:05 .peidetud.txt
-rw-r--r--  1 vilo  staff   20 Apr 15 06:57 tere.txt
~/uuskaust/pildid % ls
tere.txt
~/uuskaust/pildid % ls -a
.  ..  .peidetud.txt  tere.txt
~/uuskaust/pildid %
```

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

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
```

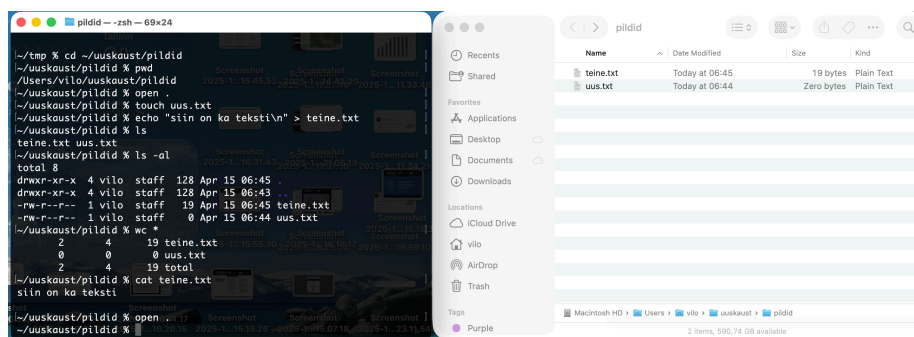
ls -lt sorteerib detailvaate aja järgi.

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 5: Finder ja terminal näitavad sama kataloogi: terminalis liigutakse kataloogi, avatakse sama koht käsuga `open .`, luuakse tühi fail `uus.txt`, kirjutatakse faili `teine.txt` üks rida ning võrreldakse tulemust Finderis ja 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 vaikumisi 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

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. Loo tühikuga failinimi ja nimeta fail ümber. Kuidas hoiad nime ühe argumendina?

Spikker

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 puuduv fail või uuenda ajatempleid
- `cp algne.txt koopia.txt` — tee koopia
- `mv vana.txt uus.txt` — nimeta ümber
- `mv fail.txt kaust/` — liiguta kataloogi
- `rm fail.txt` — kustuta fail
- `rmdir tyhi-kataloog` — eemalda tühi kataloog

Olulisemad lipud, märgid ja kiirnupud

- `.` — praegune kataloog
- `..` — ülemkataloog
- `~` — 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.

Teksti vaatamine ja liikumine

Faili **vaatamine ei muuda selle sisu**. Vali tööriist selle järgi, kui palju teksti korraga vajad:

Käsk	Toiming ja nähtav tulemus
<code>cat fail.txt</code>	kuva kogu fail terminali; sobib lühikese teksti jaoks
<code>less fail.txt</code>	ava lehitsetav ja otsitav vaade; välju q-ga
<code>head -n 7 fail.txt</code>	kuva esimesed seitse rida
<code>tail -n 7 fail.txt</code>	kuva viimased seitse rida

`-n` määrab ridade arvu. Ilma selle valikuta näitavad **head** ja **tail** kümmet rida. Lühikesest failist ei saa rohkem ridu kui seal on.

Võrdle algust ja lõppu

Linuxis ja macOS-is on `/etc/hosts` tavaliselt olemasolev tekstifail. Selle muutmise asemel vaatame seda:

```
cat /etc/hosts
head -n 3 /etc/hosts
tail -n 3 /etc/hosts
```

Võrdle väljavõtteid tervikuga. Faili sisu on masinati erinev; praegu ei pea sa võrguaadressidest aru saama. Kui fail puudub, vali mõni olemasolev tekstifail oma kataloogist. Piltide ja muude binaarfailide jaoks need vaatamiskäsud ei sobi.

Pikka faili sirvi less-iga

cat saadab pika faili korraga terminali kerimisajalukku. **less** laseb valida, millist kohta parajasti näed. Näiteks Linuxi ja macOS-i teenuseniimede loend on tavaliselt piisavalt pikk:

```
less /etc/services
```

Kui seda faili pole, ava mõni teine pikk tekstifail. Järgmised klahvid sisestatakse **less-i** sees, mitte shelli viiba järel:

Klahv	Toiming
tühik / b	lehekülg edasi / tagasi
/ssh ja Enter	otsi tekstist ssh
n / N	järgmine / eelmine otsinguvaste
g / G	faili algus / lõpp

Klahv	Toiming
78g	mine reale 78
25%	liigu umbes veerandi kaugusele failis
q	välju tagasi shelli

Mitut faili saab avada kujul `less esimene.txt teine.txt`. Siis viivad `:n` ja `:p` järgmise ning eelmise faili juurde; tavaline `n` liigub endiselt otsinguvastete vahel. `v` avab praeguse faili redaktoris, sageli `vi` või `vim`: sellest alates võid faili ka muuta. Redaktorist väljumist õpetab järgmine peatükk.

Faili lõpp ja kasvav logi

`tail -n 20 app.log` näitab lõpu kakskümmend rida ja lõpetab. `tail -f app.log` näitab lõppu ning jääb juurde tulevaid ridu ootama. See ei ole kinnijäämine; katkesta jälgimine `Ctrl-c`-ga. Oma kahe terminaliga katse tuleb logide peatükis.

Minitest

1. Vali tekstifail ja näita ainult selle esimest rida. Millist valikut vajad?
2. Otsi `less`-is mõnda korduvat sõna ning liigu järgmise ja eelmise vaste juurde.
3. Mille poolest erinevad `n` ja `:n`? Millal kumbki kasulik on?
4. Miks `tail -f` tavaliselt ise ei lõpeta, aga `tail -n 20` lõpetab?

Spikker

Eesmärk: Vali faili pikkuse ja küsimuse järgi: kogu sisu, lehitsemine, algus või lõpp.

Käsud ja näited

- `cat /etc/hosts` — kuva lühikese faili sisu
- `less /etc/services` — lehitse pikemat faili
- `head -n 7 fail.txt` — näita esimesed seitse rida
- `tail -n 7 fail.txt` — näita viimased seitse rida
- `tail -f app.log` — oota lisanduvaid ridu; lõpeta `Ctrl-c`-ga
- `less a.txt b.txt` — ava mitu nimetatud faili

Märgid ja valikud

- `q` — `less`: välju
- `/tekst` — `less`: otsi edasi
- `n` / `N` — `less`: järgmine / eelmine vaste
- `78g` / `25%` — `less`: rida 78 / veerandi peale

- `g / G` — less: algusse / lõppu
- `v` — less: ava määratud redaktor; pärast väljumist naased
- `:n / :p` — less: järgmine / eelmine fail

Pane tähele: `cat` ei peata pikka väljundit ekraanitäite kaupa.

Failide muutmine: nano ja esimene kokkupuude vim-iga

`less`-iga vaatasid faili; **tekstiredaktoriga muudad seda**. Alguseks sobib `nano`, mille klahvijuht on akna allservas. `vi` ja `vim` on klassikalised tekstiredaktorid, kus sisestamine ja käskude andmine toimuvad eri režiimides.

Loo ja muuda fail nano-ga

Tööta oma harjutuskataloogis. `nano naide.txt` avab olemasoleva faili või uue tühja dokumendi. **Kettal muutub fail alles salvestamisel.**

```
nano naide.txt
```

Kirjuta redaktorisse kolm rida:

```
esimene
teine
kolmas
```

Salvesta **Ctrl-o**, kinnita nimi **Enter**-iga ja välju **Ctrl-x**-iga. Allserva märke `~O` tähendabki **Ctrl-o**, mitte kahte sisestavat märki.

Kontrolli tulemust shellis:

```
cat naide.txt
```

Ava fail uuesti, muuda teisel real üks sõna ja salvesta. **Ctrl-w** alustab `nano`-s tekstiotsingut. Kui väljud salvestamata muudatustega, küsib redaktor, kas soovid need salvestada; loe küsimust, mitte ära vajuta kinnitust harjumusest.

Kui avaneb vi või vim

Need redaktorid võivad avaneda ka mõne teise tööriista, näiteks Giti kaudu. Väljumiseks ei pea kogu redaktorit ära õppima:

Klahv või käsk	Toiming
Esc	naase tavarežiimi
i	alusta teksti sisestamist kursori kohale
:w ja Enter	salvesta fail
:q ja Enter	välju, kui salvestamata muudatusi pole
:q! ja Enter	hülga salvestamata muudatused ja välju

Klahv või käsk	Toiming
:wq ja Enter	salvesta ning välju

Kooloniga käsu eel vajuta **Esc. :q!** ei taasta varasema salvestamise eelset faili, vaid loobub ainult veel salvestamata muudatustest.

Kui nano puudub

`command -v nano` näitab, kust shell käsu leiaks; tühi väljund tähendab, et seda nime ei leitud. Kasuta olemasolevat tekstiredaktorit või paigalda **nano** oma süsteemi paketi halduriga. Ära ava tundmatut süsteemifaili lihtsalt katsetamiseks.

Minitest

1. Lisa failile **naide.txt** neljas rida ja kontrolli salvestatud tulemust **cat**-iga.
2. Ava sama fail uuesti, muuda teksti ja välju seekord salvestamata. Kas fail kettal muutus?
3. Kui **vi** või **vim** on olemas, ava sellega **naide.txt** ja välju muudatusi tegemata.

Spikker

Eesmärk: less-iga vaatasid faili; tekstiredaktoriga muudad seda. Alguseks sobib nano, mille klahvijuht on akna allservas. vi ja vim on klassikalised tekstiredaktorid, kus sisestamine ja käskude...

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.

Käskude kuju ja argumentide loogika

Uut käsku on lihtsam lugeda, kui eristad **toimingut**, selle **valikuid** ja **sisendit**.

Alusta ühest näitest

```
ls -la /etc
```

Osa	Täendus
ls	käsk: loetle kataloogi sisu
-l	valik: näita detailset loendit
-a	valik: kaasa ka peidetud kirjed
/etc	argument: millist kataloogi vaadata

-la ühendab siin valikud -l ja -a. Üldine märkimisviis on käsk [valikud] [argumentid]: käsiraamatu nurksulud tähistavad valikulist osa, **mitte käsureale kirjutatavaid sulge**. Tehnilises dokumentatsioonis võidakse argumentideks nimetada ka valikuid.

Lühikesed ja pikad valikud

Lühike valik algab ühe kriipsuga, pikk valik enamasti kahega. Näiteks GNU `ls -a` ja `ls --all` on sama tähendusega, kuid macOS-i `ls` pikka kuju `--all` ei toeta.

Mõni valik vajab väärtust. Näites `head -n 5 fail.txt` kuulub 5 valiku `-n` juurde, mitte ei ole teine failinimi. Ära ühenda väärtusega valikuid juhuslikult üheks lipureaks.

Lipul ei ole kõigi käskude jaoks sama tähendust: `ls -h` näitab suurusi ühikutega, teises programmis võib `-h` tähendada abi. Kontrolli `man`-ist oma käsu valikuid.

Argumentide roll ja järjekord

`cp vana.txt uus.txt` kopeerib **allika** `vana.txt` **sihtfailiks** `uus.txt`. Nende vahetamine muudab kopeerimise suunda. Sihtfaili olemasolev sisu võib üle kirjutatud saada.

Kõige ülekantavam harjumus on panna valikud failinimede ette, näiteks `ls -l /etc`. Kõik programmid ei luba valikuid suvalisse kohta paigutada.

Tühikud, jutumärgid ja paomärk

Shell eraldab sõnu tühikute järgi. Tühikuga nimi tuleb koos hoida:

```
touch "minu fail.txt"
ls -l "minu fail.txt"
```

Sama nime saab kirjutada ka `minu\ fail.txt`: **kurakriips** \ võtab järgnevalt tühikult eraldaja tähenduse.

Kuju	Shelli käitumine
"minu fail.txt"	üks argument; tühik jääb nime sisse
"\$HOME"	üks argument; muutuja asendatakse väärtusega
'\$HOME'	täht-tähelt tekst <code>\$HOME</code> , mitte muutuja väärtus

`HOME` on kodukataloogi sisaldav muutuja. Võrdle:

```
echo "$HOME"
echo '$HOME'
```

Esimene näitab koduteed, teine sõna `$HOME`. Üksikjutumärgid kaitsevad teksti **shelli** tõlgenduse eest; käivitata programm võib oma argumenti ikka eraldi tõlgendada, näiteks `printf` reavahetusmärgi `\n`.

Kriipsuga algav failinimi

Paljudes käskudes lõpetab eraldi `--` valikute lugemise. Seejärel käsitletakse ka kriipsuga algavat nime argumentina:

```
touch -- -markmed.txt
ls -l -- -markmed.txt
```

Siin `--` ise ei ole pikk valik; pikk valik oleks näiteks `--help`. Alternatiiv on nimi `./-markmed.txt`, mis algab punktiga, mitte kriipsuga.

Failinimede mustrid

Mustrilaiendus (*globbing*) on **shelli käitumine, mitte uus käsk**. Shell asendab mustri sobivate nimedega enne programmi käivitamist.

Muster	Sobivad nimed
<code>*.txt</code>	nimed, mis lõpevad <code>.txt</code> -ga
<code>words.0?.txt</code>	<code>words.0</code> järel täpselt üks märk, siis <code>.txt</code>
<code>words.0[0-9].txt</code>	keskel üks number 0–9: 00 kuni 09

Katseta oma harjutuskataloogis:

```
touch words.00.txt words.01.txt words.0a.txt
ls words.0?.txt
ls words.0[0-9].txt
```

Esimene muster leiab kõik kolm, teine ainult kaks numbrilist nime. ? ei tähenda „üks number“, vaid „üks märk“.

Muster võib sobitada ka katalooginime. `ls` võib siis loetleda selle kataloogi sisu; ainult sobitatud kirjete endi vaatamiseks kasuta `ls -d *.txt`. `-d` käsitleb kataloogi kirjena.

Jutumärgid, näiteks `*.txt`, takistavad shelli mustrilaiendust. Peidetud nimed ei sobi tavaliselt algava `*` alla. Kui vaste puudub, annab zsh vaikimisi vea `no matches found`, Bash jätab mustri tavaliselt sõnasõnaliselt käsule; seadistus võib seda muuta.

Minitest

1. Milline osa käsus `head -n 5 "minu fail.txt"` on käsu nimi, valik, valiku väärtus ja failinimi?
2. Mis vahe on `"$HOME"` ja `'$HOME'`?
3. Miks leiab `words.0?.txt` ka nime `words.0a.txt`?
4. Kuidas vaadata kriipsuga algava faili õigusi nii, et nimi ei muutuks valikuks?

Spikker

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

- `head -n 3 fail.txt` — valik `-n` vajab arväärtust
- `cp 'fail nimi.txt' koopia.txt` — nimi tühikuga
- `touch -- -algab-kriipsuga.txt` — ära tõlgenda lipuna
- `ls *.txt` — kõik `.txt` lõpuga failid
- `ls words.0?.txt` — 0 järel üks märk, mitte tingimata number

Olulised märgid

- `-` — lühike lipp
- `--` — üksinda lõpetab valikute loendi
- `"..."` — hoia koos; muutujad toimivad
- `'...'` — täht-tähele nagu kirjas

- * — failimuster
- ? — üks märk

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

Sisend, väljund, torud ja suunamine

Käsu tulemus ei pea jääma ainult ekraanile. Seda saab salvestada faili või anda järgmisele käsule. Õpime kõigepealt **andmete liikumist**, seejärel **käskude käivitamise tingimusi**.

Väljund faili: > ja >>

Tavalist väljundit nimetatakse **standardväljundiks** (*stdout*). **echo** väljastab teksti koos reavahetusega. Märk > suunab selle faili: kui fail puudub, luuakse see; olemasolev sisu kirjutatakse üle.

Tööta oma harjutuskataloogis ja vali failinimed, mida sa veel ei kasuta:

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

Ekraanil näed teksti alles **cat**-i järel, sest **echo** väljund läks faili. Märk >> lisab standardväljundi faili lõppu, säilitades senise sisu:

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

Nüüd on failis kaks rida: **esimene** ja **teine**. Suunamismärgid ise reavahetusi ei lisa; need annab siin **echo**.

Shell avab ja tühjendab > sihtfaili **enne käsu käivitamist**. Ära kasuta sama faili korraga sisendina ja ülekirjutatava väljundina, näiteks `cat read.txt > read.txt`.

Sisend failist ja torust

Standardsisend (*stdin*) on voog, millest programm saab andmeid lugeda. < annab standardsisendiks faili sisu. Käsk **wc -l** loendab reavahetusi; tavapärasel tekstifailis on see ridade arv.

```
wc -l read.txt
wc -l < read.txt
```

Mõlemad loendavad kaks rida. Esimesel juhul näitab **wc** ka talle argumendina antud failinime, teisel juhul loeb anonüümset sisendvoogu.

Toru | ühendab ühe käsu standardväljundi järgmise käsu standardsisendiga. **printf** vormindab teksti; selle näite `\n` tähistab reavahetust:

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

Tulemus on 3; vahepealset faili ei teki. Toru osad võivad töötada samal ajal: teine käsk loeb esimese toodetud andmeid, mitte ei oota tingimata kogu esimese käsu lõppu.

Näita ja salvesta: tee

tee kopeerib standardsisendi nii ekraanile kui ka faili. Vaikimisi kirjutab ta sihtfaili üle; **-a** lisab lõppu.

```
printf 'Tallinn\nTartu\n' | tee linnad.txt
printf 'Narva\n' | tee -a linnad.txt
wc -l linnad.txt
```

Näed linnanimedid kohe ning failis on lõpuks kolm rida.

Veateated eraldi: 2>

Veaväljund (*stderr*) on standardväljundist eraldi voog. **2>** suunab selle faili. Eeldame, et **puuduv-fail** ei ole olemas:

```
ls read.txt puuduv-fail > valjund.txt 2> vead.txt
cat valjund.txt
cat vead.txt
```

Esimeses failis on leitud nimi **read.txt**, teises teade puuduvast failist. Tavaline toru ei suuna veateateid järgmisele käsule.

2> /dev/null viskab veaväljundi ära. **/dev/null on erifail**, mis andmeid ei talleta: nali „**/dev/null** sai täis“ ei kirjelda päris kettaruumi probleemi. Veat-singul ära veateateid peida, sest tulemus võib muidu näida täielik ka siis, kui osa tööst ebaõnnestus.

Mõlemad vood samasse faili

```
ls read.txt puuduv-fail > koik.txt 2>&1
```

> suunab standardväljundi faili ja **2>&1** suunab veaväljundi standardväljundi praegusesse sihtkohta. Järjekord loeb: kujul **2>&1 > koik.txt** jääksid vead terminali.

Lõpetuskood ja tingimuslik käivitamine

Väljundtekstist eraldi annab käsk **lõpetuskoodi** (*exit status*). **0 tähendab edu**, nullist erinev väärtus muud tulemust, sageli viga. **echo \$?** näitab vahetult eelmise käsu lõpetuskoodi:

```
ls read.txt
echo $?
```

```
ls puuduv-fail
echo $?
```

Esimene kood on 0, teine nullist erinev. Küsi koodi kohe: ka järgmine `echo` asendab selle enda lõpetuskoodiga.

Märk	Käivitamisreegel
<code>;</code>	oota esimese käsu lõppu, käivita siis teine ka vea korral
<code>&&</code>	käivita teine ainult siis, kui esimene lõpetas koodiga 0
<code>\\ </code>	käivita teine ainult siis, kui esimene lõpetas nullist erineva koodiga

Katseks sobib **false**: see ei muuda midagi, ei prindi teksti ja lõpetab alati ebaedukalt.

```
false ; echo 'käivitun ikkagi'
false && echo 'seda rida ei näe'
false || echo 'käivitun ebaedu järel'
```

Praktiline kasutus on `mkdir uus-katse && cd uus-katse`: kui kataloogi loomine ebaõnnestub, ei liiguta shell sind mujale. `;` ei edasta andmeid, `toru |` ei kontrolli eelmise käsu edu.

Minitest

1. Loo kolmerealine fail ning lisa neljas rida nii, et senine sisu säilib.
2. Salvesta sama väljund faili ja näita seda samal ajal terminalis.
3. Võrdle `printf 'tere\\n' | wc -l` ja `printf 'tere\\n'; wc -l read.txt` tulemust. Milliseid andmeid kumbki `wc` loeb?
4. Miks veateadete peitmine ei muuda ebaõnnestunud käsku edukaks?
5. Ennusta, kuhu lähevad vood kujul `> koik.txt 2>&1` ja ümberpööratud järjekorras.

Spikker

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 lõpu 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 loendus
- `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 lõpu 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`.

Esimene tervikharjutus: 30 minutit

Oled õppinud terminalis liikuma, faile vaatama ja muutma ning väljundit suunama. Nüüd tee neist oskustest üks töövoog: **loo väike sõnaloend, kopeeri see ja muuda koopiat originaali puutumata.**

1. Vali töökoht

`mkdir -p` loob kogu puuduva tee, ka kataloogi `tmp`, kui seda veel pole. Kui oled sama katset juba teinud, vali **tervikharjutus** asemel uus nimi, et mitte vanu faile üle kirjutada.

```
mkdir -p ~/tmp/tervikharjutus
cd ~/tmp/tervikharjutus
pwd
ls
```

Kontroll: tee lõpeb `/tmp/tervikharjutus` ja uue kataloogi loend on tühi.

2. Kirjuta ja kontrolli

Esimene rida loob faili; järgmised lisatakse lõppu. Ennusta enne viimast käsku ridade arvu.

```
echo tere > sonad.txt
echo maailm >> sonad.txt
echo linux >> sonad.txt
cat sonad.txt
wc -l sonad.txt
```

Näed kolme sõna eraldi ridadel ning seejärel ridade arvu 3. Kui tulemus erineb, peatu: korduv >> lisab ridu juurde, aga > kirjutab vana sisu üle.

3. Muuda ainult koopiat

```
cp sonad.txt koopia.txt
echo terminal >> koopia.txt
wc -l sonad.txt koopia.txt
```

Originaalis peab olema kolm ja koopias neli rida. Viimane kokkuvõtteid näitab kahe faili peale seitset rida. Vaata mõlemat `cat`-iga, kui arvud ei klapi.

Iseseisev ülesanne

1. Lisa koopiale `nano` abil viies rida ja salvesta.
2. Näita koopiast ainult viimased kaks rida.
3. Nimeta koopia ümber nimeks minu `sonad.txt`. Miks on käsus jutumärke vaja?
4. Kontrolli, et algne `sonad.txt` sisaldab endiselt kolme rida.
5. Selgita, mis muutuks, kui kasutaksid lisamisel >> asemel >.

Kui mõne käsu tähendus on ebaselge, vaata abi leidmist või suunamist. Kui saad harjutuse tehtud, jätka operatsioonisüsteemide ja shellide ülevaatest, seejärel liigu failisüsteemi kaardi juurde.

Spikker

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

Lühike töövoog

- `pwd && ls` — kontrolli alguskohta
- `mkdir proov && cd proov` — loo töökataloog
- `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.

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.

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 <code>uname</code> väljundis
GNU	paljud tuttavad käsureatööriistad	käsud ja nende GNU-spetsiifilised lipud
shell	loeb ja tõlgendab käsuri	viip, muutujad ja shelli sisekäsud
macOS, Windows	sarnane pind, eri tööriistad	sama käsk võib käituda erinevalt
<code>uname -a</code>	kontrolli süsteemi ja kernelit	üks pikk süsteemirida
<code>echo "\$SHELL"</code>	näita vaikimisi shelli	tee nagu <code>/bin/zsh</code>
<code>ps -p \$\$</code>	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

Võrdle oma keskkonda

Unix-laadses shellis võrdle `uname -a` süsteemiinfot, `echo "$SHELL"` seadistatud sisselogimisshell ja `ps -p $` tegelikult töötavat shelliprotsessi. `$SHELL` ei pruugi uue shelli käivitamisel muutuda.

Vajaduse järgi kasuta üht lisakontrolli:

Keskkond	Käsk	Tulemus
paigaldatud Bash või zsh macOS	bash --version või zsh --version sw_vers	valitud shelli versioon
Windowsi PowerShell	wsl -l -v	operatsioonisüsteemi versioon paigaldatud WSL-jaotused ja WSL-i põlvkond

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 käsu keskkond ja skriptikeel

Linux, Unix ja GNU erinevused

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

- Linux annab kerneli
- GNU annab suure hulga käsuraatööriistu
- 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 käsuraal 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-töövoogudest on sama

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

macOS: mis on teisiti

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

- paketi haldur 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 kasutada mõne Linuxi tööriistaga samanimelist käsku, tee seda vajaduse korral, mitte „igaks juhuks“. Näiteks võrgukäskude puhul võib hiljem abiks olla **brew install iproute2mac**, kuid alguses ei ole seda vaja.

Windows: milline tee valida

Windowsis on kolm peamist teed:

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

Kui soovid, et õpiku näited töötaksid Windowsis Linuxiga võimalikult sarnaselt, 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 kāske, 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 hostsüsteem
- 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 PowerShell 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 PowerShellis, 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 Linuxilaadse 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.

Spikker

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

Kontrollkäsud

- `uname -a` — süsteemi ja kerneli rida
- `echo "$SHELL"` — seadistatud sisselogimisshell
- `ps -p $$` — praegu töötav 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.

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, millist tüüpi faile eri kataloogides tavaliselt leidub, on ka veaotsing ja navigeerimine palju lihtsam.

Kiirülevaade

Eesmärk on luua süsteemist ülevaade, et teaksid, kus on ohutu töötada ja millised kataloogid kuuluvad süsteemile.

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 <code>ls</code> , <code>cat</code> , <code>grep</code>
/etc	seadistused	palju süsteemifaile
/var	muutuv sisu, näiteks logid	logid, vaheandmed ja teenuste failid
/tmp	ajutised failid	ajutine tööala
<code>pwd</code>	näita praegust asukohta	üks täistee
<code>echo "\$HOME"</code>	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

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 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 nii teele endale kui ka selle rollile.

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.

Spikker

Eesmärk: õpi eristama süsteemi juurt, oma kodukataloogi ning ajutisi ja süsteemseid katalooge; 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 kataloogi

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.

Kettaruum ja süsteemi maht

df näitab failisüsteemi vaba ruumi; **du** mõõdab failide ja kataloogide kasutatud ruumi. Nendega saad eristada küsimusi „kas ketas on täis?“ ja „mis siin ruumi võtab?“.

Alusta üldpildist

Käsk	Mida näed
df -h .	praegust kataloogi sisaldava failisüsteemi kogu-, kasutatud ja vaba maht
df -h	mitme failisüsteemi tabel koos haakepunktidega
du -sh .	praeguse kataloogi ja selle sisu kogumaht

-h kuvab suurused ühikutega, näiteks K, M ja G. **du -s** annab ühe kokkuvõttereala, mitte kogu alamkataloogide loendi. Suure puu mõõtmine võib võtta aega.

Leia suurimad kirjed

du -a loetleb lisaks kataloogidele failid. **-k** seab kõigi suuruste ühikuks 1024 baiti, et Linuxi ja macOS-i tulemusi saaks samal alusel lugeda. **sort -n** sordib arvuliselt, **-r** pöörab järjestuse kahanevaks.

```
du -ak . | sort -nr | less
```

Üleval on suurimad kirjed; **less**-is saad otsida ja **q**-ga väljuda. Kataloogi rida sisaldab juba alamkataloogide mahtu: **ära liida kogu loendi ridu kokku**, muidu loendad sama sisu mitu korda.

Näiteks kui ülemiste ridade seas on mahukas **./projekt/.git**, mõõda järgmiseks projekti, mitte kogu ketast. Peidetud kataloogid kaasatakse automaatselt, sest lähtekohaks on **.**, mitte shelli muster *****.

Lühem loend ja ühikutega sort

Linuxi ja macOS-i **du -d 1** piirab kuvatava kataloogisügavuse ühe tasemega; mõõtmiseks loetakse siiski kogu sisu:

```
du -k -d 1 . | sort -nr
```

See annab alamkataloogide mahud ja kogusumma, mitte eraldi iga tavalise faili rida. Kui sinu **du** ei toeta **-d**, kasuta eelmist **du -ak** varianti.

Inimloetava pika loendi jaoks sobib:

```
du -ah . | sort -hr | less
```

sort -h arvestab suurusliiteid; ilma **-r**-ita tuleb väiksem liide enne suuremat. See sobib ühe **du -h** käivituse väljundile. Suvalisi ühikutega arve ei teisenda see täpselt baidisummaks: näiteks 12345K järjestub enne 1M, sest liidet võrreldakse enne arvu. Täpsemaks ühtseks võrdluseks eelista **du -k | sort -n**. Kõik vanemad **sort**-i variandid ei toeta **-h**.

Kui mõõtmine ei klapi

- **Permission denied:** osa puust jäi lugemata. Veateate peitmine kujul `2>/dev/null` ei tee mõõtmist täielikuks.
- **df ja du erinevad:** esimene mõõdab tervet failisüsteemi, teine valitud puus nähtavat sisu. Erinevust võivad tekitada ka hetktõmmised, jagatud kettaplokid või kustutatud fail, mida protsess veel avatuna hoiab.
- **ls -l ja du erinevad:** faili loogiline pikkus ei pea võrduma talle eraldatud kettaruumiga, näiteks hõredas failis.
- **Mõõtmine on aeglane:** alusta konkreetsest projektist või allalaadimiste kataloogist. Kogu / läbimine pole tavaliselt esimene vajalik samm.

Suur fail ei ole automaatselt tarbetu. Tuvasta sisu ja selle kasutaja enne kustutamist; ruumimõõtmine ise midagi ei eemalda.

Minitest

1. Kui palju on vaba ruumi failisüsteemis, kus praegu töötad?
2. Leia ühe projekti suurim alamkataloog. Kas see on nähtav ka tavalise **ls**-iga?
3. Miks on **du** loendi kogusumma liitmine eksitav?
4. Mille poolest erinevad **sort -n** ja **sort -h**? Proovi arve 9 ja 100, seejärel 900K ja 1M.

Spikker

Eesmärk: **df** näitab failisüsteemi vaba ruumi, **du** loendab valitud failide ja kataloogide kettakasutust.

Käsud ja näited

- **df -h .** — selle failisüsteemi kasutatud ja vaba maht
- **du -sh .** — praeguse kataloogipuu kogumaht
- **du -h -d 1 .** — esimese taseme mahud, ka peidetud kataloogid
- **du -h -d 1 . | sort -h** — sordi ühikusufiksiga mahud kasvavalt
- **du -ak . | sort -nr | less** — kõik kirjed 1024-baidistes ühikutes, suurimad ees

Märgid ja valikud

- `du -s` — ainult kogusumma
- `du -a` — ka üksikfailide mahud
- `du -k` — ühtlane 1024-baidine ühik
- `sort -h` — arvestab ühikusufiksi järku, siis arvvaartust
- `sort -nr` — arvuliselt kahanev järjestus

Pane tähele: Veateated võivad tähendada puudulikku kogusummat. Ära liida du alamkataloogide ja nende vanemate mahtusid uuesti kokku.

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 eeldavad õiguste õiget mõistmist.

Kiirülevaade

Eesmärk on lugeda õiguste rida, teha fail vajadusel käivitatavaks ja mõista veateadet `Permission denied`.

Käsk või mõiste	Milleks	Mida tavaliselt näed
<code>ls -l</code>	näita õigusi, omanikku ja gruppi	rida nagu <code>-rw-r--r--</code>
<code>chmod</code>	muuda õigusi	edukal juhul sageli vaikne
<code>chown</code>	muuda omanikku või gruppi	edukal juhul sageli vaikne
täitmisbitt <code>x</code>	luba faili käivitada	faili saab programmina käivitada
kataloogi <code>x</code>	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

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**-õigusega võib kataloog olemas olla, aga sa ei saa selles normaalselt liikuda

Kataloogi **r** lubab nimede loendit lugeda, **w** kirjeid luua või eemaldada ning **x** teadaolevaid nimesid kasutada. Faili kustutamine sõltub tavaliselt vanemkataloogi **w**- ja **x**-õigusest, mitte faili enda kirjutamisõigusest; lisareeglid, näiteks sticky-bit ja ACL, võivad seda piirata.

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 annab omanikule lugemise ja kirjutamise, grupile ning teistele ainult lugemise
- 755 annab omanikule kõik kolm õigust, grupile ning teistele lugemise ja täitmise

Numbrid on summad: $r = 4$, $w = 2$, $x = 1$. Näiteks $6 = 4 + 2$. Need määravad õigused, **mitte faili tüübi**.

Omaniku muutmine

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

Käivitataavaks tegemine

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

- shebang nagu `#!/bin/sh` või `#!/usr/bin/env python3` ütleb, millise tõlgendajaga 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
```

Minitest

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

Spikker

Eesmärk: loe õiguste rida, mõista `rwX` tähendust ja erista faili käivitataavust 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` — omanik `rw`, teised `r`
- `755` — omanik `rw`, teised `rx`

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

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
<code>whoami</code>	näita praegust kasutajat	üks kasutajanimi
<code>id</code>	näita kasutaja tunnuseid ja gruppe	pikem rida <code>UID/GID</code> ja gruppidega
<code>groups</code>	näita gruppe lühemalt	grupid nimede loend
<code>sudo käsk</code>	käivita üks käsk kõrgemate õigustega	parooliviip või veateade
<code>sudo -l</code>	kontrolli lubatud <code>sudo</code> käske	lubade loend või keeld

Tüüpilised algaja vead

- arvatakse, et `sudo` tähendab „paranda see käsk võluvael ära“

- jäädakse pikaks ajaks **root**-i või kõrgemate õigustega shelli
- käivitatakse ohtlik käsk **sudo**-ga enne, kui kontrollitakse, kus parajasti ollakse

Esimene kontroll

```
whoami
id
groups
```

Need kolm käsku ei muuda midagi. Võrdle nende üksikasjalikkust: **whoami** vastab ühe nimega, **groups** annab grupinimed lühidalt ja **id** näitab sama teavet tehnilisemalt UID/GID numbritena.

```
sudo -l
```

sudo -l kontrollib, milliseid kõrgemate õigustega kāske sinu kasutaja käivitada tohib. Kui see küsib parooli, on see normaalne.

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 viibas ära tunda

Sageli on **root**-kasutaja viiba lõpus märk **#**, tavakasutajal aga **\$** 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-shell'i või vahetavad kasutajat. See tähendab, et järgmised käsud töötavad juba kõrgemate õigustega seni, kuni sellest shellist väljud.

Algajale on see riskantsem, sest:

- viip võib muutuda
- iga järgmine käsk võib teha suurema mõjuga muudatuse
- on lihtne unustada, et sa ei ole enam tavakasutaja

Seepärast on hea algreegel:

- eelista **sudo käsk**
- väldi püsivat root-shell'i, kui selleks ei ole selget põhjust

Mida root-ina mitte teha

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

Välgi eriti:

- `rm -rf` käske `root`-kasutajana
- 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

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`.

Spikker

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.

Muutujad, keskkond, PATH ja aliased

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

Kiire orientiir

Kuju	Milleks	Mida tavaliselt näed
<code>nimi='Mari'</code> <code>echo "\$nimi"</code> <code>export</code> <code>NIMI='väärtus'</code> <code>env</code>	loo shellimuutuja näita muutuja sisu anna väärtus alamprotsessidele vaata keskkonnamuutujaid	edukal juhul vaikne väärtus terminalis edukal juhul vaikne ridade loend
<code>echo "\$PATH"</code>	vaata käsuotsingu teid	koolonitega eraldatud teed
<code>command -v python3</code> <code>type cd</code>	vaata, mis käsk leitakse vaata käsu liiki	failitee või tühi väljund sisekäsk, alias, failitee või muu
<code>type -a grep</code>	näita kõik sama nimega vasted	mitu võimalikku vastet
<code>alias ll='ls -lah'</code> <code>source ~/.zshrc</code>	loo lühinimi käivita faili käsud praeguses zsh-s	edukal juhul vaikne tulemus sõltub faili sisust

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 käsku `which` ega märgata aliast või shelli sisseehitatud käsku
- lisatakse alias valesse seadistusfaili ja imestatakse, miks uus terminaliaken seda ei tunne
- kopeeritakse viiba 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 käivitata 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	<code>ll</code>	shell asendab lühinime käsuga
shelli funktsioon	<code>mycmd</code>	kasutaja või seadistusfaili määratud funktsioon
sisekäsk	<code>cd</code>	shelli sisseehitatud käsk
reserveeritud sõna	<code>if, for</code>	shellikeele osa
väline programm	<code>grep, python3</code>	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 käivitata 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 käske. 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 käsura 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
- kataloogi lisamine muutujasse `PATH` lubab skripti käivitada 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 sisselogimisshelliga või interaktiivse shelliga.

Praktiline algaja jaotus:

Shell	Kuhu panna aliased ja mugavused	Kuhu pannakse sageli PATH-i muudatused
zsh	~/ <code>.zshrc</code>	~/ <code>.zprofile</code> või ~/ <code>.zshrc</code>

Shell	Kuhu panna aliases ja mugavused	Kuhu pannakse sageli PATH-i muudatused
bash	~/.bashrc	~/.bash_profile või ~/.bashrc

Kui sa ei taha veel käivitusfailide üksikasjadesse süveneda, alusta 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.

Aliases

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 aliaseid:

```
alias
type ll
```

Eemalda alias praeguses shellis:

```
unalias ll
```

Välldi aliaseid, mis teevad ohtliku käsu liiga „mugavaks“. Näiteks `rm` ümberkirjutamine võib aidata ühes shellis, kuid 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 aliaseid ja sisekäskke ning arvestab shelli vaatenurka
- `command -v` on lühike ning sobib hästi kontrolliks

Kui „käsk käitub imelikult“, alusta kontrollist:

```
type käsk
type -a käsk
```

Viip ja ajalugu lühidalt

Viip (`prompt`) on tekst enne käsu sisestamist. See ei ole käsu osa.

Näide:

```
~/projekt %
```

Kui viip näitab üht asja ja oled ebakindel, kontrolli asukohta käsuga:

```
pwd
```

Ajutine lihtne viip `zsh`-is:

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

Ajutine lihtne viip `bash`-is:

```
PS1='$ '
```

Viiba mall on shellimuutuja, mitte terminaliakna seadistus. Vaata seda muutmata kujul `printf '%s\n' "$PS1"`; `zsh`-s sobib ka `print -r -- "$PROMPT"`. Näed malli koodi, mitte tingimata ekraanile joonistatud viipa.

terminal.cs.ut.ee laadse ajutise viiba jaoks vali **üks**, oma shelli kuju:

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

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

Need nimed on viiba näidistekst, mitte kasutajakonto ega arvuti nime muutmine. Proovi ainult praeguses shellis. Püsiseadistusi selgitab shelli seadistusfailide lisa.



```
pildid -- zsh -- 83x23
% pwd
/Users/vilo/uuskaust/pildid
% PROMPT='$ '
$
$ pwd
/Users/vilo/uuskaust/pildid
$
$ PROMPT='%~ %# '
~/uuskaust/pildid %
~/uuskaust/pildid % pwd
/Users/vilo/uuskaust/pildid
~/uuskaust/pildid %
~/uuskaust/pildid %
```

Joonis 6: Viiba lihtsustamine ei muuda praegust töökataloogi.

1. `pwd` näitab kataloogi täielikku teed.
2. Viiba mall asendatakse lühikese dollarimärgiga.
3. Järgmine `pwd` kinnitab, et kataloog jäi samaks.
4. `zsh` mall `%~ %#` toob viipa tagasi lühendatud tee ja lõpumärgi.

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	kontole seadistatud sisselogimisshell; ei pruugi olla praegu töötav shell
EDITOR	vaikimisi tekstiredaktor paljude tööriistade jaoks

Muutuja	Tähendus
LANG	keele- ja lokaaliseadistus

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.

Spikker

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

Põhikujud

- `nimi='Mari'; echo "$nimi"` — shellimuutuja kehtib selles shellis
- `export DEMO=1` — anna muutuja edasi käivitavatele programmidele
- `env | grep DEMO` — kontrolli, kas muutuja on keskkonnas
- `echo "$PATH"` — vaata käskude otsinguteed
- `command -v python3` — näita käsu lahendust, ka alias või sisekäsk
- `type cd` — vaata käsu liiki
- `type -a grep` — näita kõik vasted
- `export PATH="$HOME/bin:$PATH"` — pane oma käskude kataloog otsingutee 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, sisekäsk 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.

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.

Kiire orientiir

Tööriist	Kus tavaliselt	Milleks
<code>apt</code>	Debian, Ubuntu	süsteemi paketid
<code>dnf</code>	Fedora	süsteemi paketid
<code>pacman</code>	Arch Linux	süsteemi paketid
<code>brew</code>	macOS, vahel Linux	kasutajataseme tööriistad
<code>python3 -m pip</code>	Python	Pythoni paketid
<code>npm</code>	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 juhised

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 vaikeset 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 paketi haldur 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.

Paketi info ja eemaldamine

Vali ainult oma süsteemi rida. Info küsimine on vaatamine; eemaldamine muudab paigaldatud tarkvara.

Süsteem	Loe kirjeldust	Eemalda teadlikult paigaldatud näidispakett
Debian / Ubuntu	<code>apt show ripgrep</code>	<code>sudo apt remove ripgrep</code>
Fedora	<code>dnf info ripgrep</code>	<code>sudo dnf remove ripgrep</code>
Arch Linux	<code>pacman -Si ripgrep</code>	<code>sudo pacman -R ripgrep</code>
Homebrew	<code>brew info ripgrep</code>	<code>brew uninstall ripgrep</code>

Ära eemalda süsteemiteeke ega pakette, mille rolli sa ei tunne. Loe paketihalduri kavandatud muudatused enne kinnitamist läbi.

Süsteemi paketihaldur ja keele paketihaldur

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

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

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.

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'ga sageli valiku `--cask` abil:

```
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`.

Kui paigaldus ebaõnnestub

Kõigepealt loe veateate viimaseid ridu. Seal on tavaliselt põhjus.

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

Minitest

1. Ütle, milline süsteemi paketiadministraator 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`.

Spikker

Eesmärk: vali õige tase: süsteemi paketi haldur tööriistadele, keele paketi haldur teekidele ja projekti sõltuvused võimalusel projekti sisse

Põhikujud

- `apt search ripgrep` — otsi Debianis või Ubuntu
- `sudo apt install ripgrep` — paigalda Debianis või Ubuntu
- `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` — Linux'i 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 paketi haldurit ja `pip/npm`-i; need lahendavad eri taseme probleeme.

Lihtne veaotsing käsureal

Alusta **täpsest käsust ja veateatest**, mitte juhuslikust parandusest. „Ei tööta“ võib tähendada vale failiteed, puuduvat programmi, õiguste probleemi või vale shelli.

Kontrolli ilma parandusi tegemata

Asenda tabelis näidisnimi oma käsu või failiga.

Küsimus	Kontroll ja tähendus
Kus töötan?	<code>pwd</code> näitab praegust täisteed
Kas fail on siin?	<code>ls -lah</code> näitab ka peidetud nimesid, suurusi ja õigusi
Kas programm leitakse?	<code>command -v python3</code> annab leitud nime või tee; tühi väljund tähendab, et käsku ei leitud
Kas nimi on alias?	<code>type -a python3</code> näitab Bashi ja zsh leitud variante
Kas faili saan käivitada?	<code>ls -l skript.sh</code> näitab õigusi; <code>head -n 1 skript.sh</code> näitab võimalikku shebang-rida
Milline shell töötab?	<code>ps -p \$\$</code> näitab praegust shelliprotsessi
Kas Bashi süntaks on korrektne?	<code>bash -n skript.sh</code> loeb skripti seda käivitamata

bash -n ei tõenda skripti ohutust ega õiget tulemust. See leiab süntaksivigu; failiteed ja käskude tagajärjed tuleb ikka üle vaadata. Tavaline `bash skript.sh` seevastu käivitab faili päriselt.

Vali järgmine samm veateate järgi

Veateade	Kontrolli enne muutmist
<code>command not found</code>	kirjapilt, paigaldus ja <code>PATH</code> ; vaata <code>command -v nimi</code>
<code>No such file or directory</code>	asukoht, suhteline tee, suur- ja väiketähed ning jutumärgid
<code>Permission denied</code>	tühikutega nime ümber faili ja vanemkataloogi õigused; näiteks <code>ls -ld kataloog</code> vaatab kataloogi ennast
<code>syntax error</code>	kas käsu kuju ja kasutatav tõlgendaja sobivad kokku
<code>package not found</code>	paketi nimi, valitud hoidlad ja paketinimekirja värskus
<code>database is locked</code>	milline programm vea andis; põhjus võib olla teine paketi haldur või andmebaasi kirjutav protsess

Lukufaili kustutamine ei lõpeta lukku hoidvat protsessi. Ära eemalda lukke ega lisa käsule `sudo` enne põhjuse leidmist.

Vale failitee

`ls "Minu fail.txt"` otsib üht tühikuga nime. Ilma jutumärkideta otsiks käsk kahte nime. Suhteline tee `../andmed/sisend.txt` sõltub praegusest asukohast; täistee mitte.

Vale shell

`echo "$SHELL"` näitab tavaliselt kasutajale seadistatud sisselogimisshelli, **mitte tingimata praegu töötavat shelli**. Ka praegune shell ei määra üksi skripti tõlgendajat: `bash skript.sh` valib Bashi, `sh skript.sh` valib `sh`, otsekäivitus `./skript.sh` kasutab shebang-rida.

Kontrolli skripti sisu enne käivitamist. Kui usaldusväärsel oma skriptil puudub täitmisõigus, selgitab selle lisamist õiguste peatükk.

Programm on paigaldatud, kuid ei leita

Vaata `echo "$PATH"` abil otsinguteid. Pythoni puhul võib vaja olla õige virtuaal-keskkonna aktiveerimist; mõne paigalduse järel uue shelli avamist. Kontrolli ka, kas paketinimi ja käivitatava programmi nimi üldse kattuvad.

Hea veakirjeldus

Abi küsides lisa eesmärk, täpne käsk, täielik veateade ning operatsioonisüsteem ja töökeskkond. Asukoha kirjeldamisel eemalda paroolid, võtmed, isikuandmed ja tundlikud teed; ka logi ei tasu enne ülevaatamist avalikult jagada.

Minitest

1. Kontrolli, kust sinu shell leiab `python3`. Kas paigaldatud variante on mitu?
2. Kasuta meelega olematut failinime ja selgita saadud veateadet.
3. Miks `bash -n skript.sh` ja `bash skript.sh` ei ole samaväärsed kontrollid?
4. Koosta lühike veakirjeldus nii, et teine inimene saaks sama olukorra turvaliselt taastada.

Spikker

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-rida
- `bash -n skript.sh` — kontrolli süntaksit ilma käivitamata
- `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` — andmebaasi hoiab teine toiming; põhjus sõltub programmist

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

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, kuid neid kontrollitakse eri käskudega.

Kiire orientiir

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

Kui kasutad macOS-i ja `ip a` või `ss` annab `command not found`, siis see on normaalne. Need on Linux-i 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 Linuxis käsk **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 arvuti tegelikku võrguaadressi
- kontrollitakse kaugest 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 Linuxis 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	Täendus
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 aadressil `127.0.0.1` 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, kuid näitab, kas TCP-ühenduse loomine selle pordiga 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 võrgukontrolleks 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.

Spikker

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

- `curl -I` — küsi HTTP-päised HEAD-päringuga
- `curl -L` — järgi HTTP ümbersuunamisi
- `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.

Failide kopeerimine ja sünkroonimine

Vali tööriist allika ja eesmärgi järgi:

Vajadus	Tööriist ja oluline erinevus
koopida samas masinas	<code>cp</code> ; õnnestub tavaliselt vaikides
fail üle SSH	<code>scp</code> ; näitab enamasti ülekande edenemist
korduv kataloogide sünkroonimine	<code>rsync</code> ; saab enne näidata plaani
fail veebiaadressilt	<code>curl</code> või <code>wget</code> ; kumbki ei vaja SSH-kontot

Koopia võib sihtfaili üle kirjutada. Enne suurt ülekannet kontrolli suunda ja seda, kas sihtkohas on säilitamist vajavaid faile.

Kohalik katse

Loo uus katsekoht. `cp -R` kopeerib rekursiivselt, s.t ka alamkataloogid koos sisuga.

```
mkdir -p ~/tmp/kopeeri-naide/allikas/alam
cd ~/tmp/kopeeri-naide
printf 'tere\n' > allikas/alam/fail.txt
cp allikas/alam/fail.txt koopia.txt
cp -R allikas allikas-koopia
cat koopia.txt
ls allikas-koopia/alam
```

Näed teksti `tere` ja koopiapuus nime `fail.txt`. Kui katsekataloog oli juba olemas, vali enne alustamist uus nimi: eriti `cp -R` tulemus sõltub sihtkataloogi olemasolust.

rsync: enne plaan, siis ülekanne

`rsync -a` kopeerib puu koos tavapäraste õiguste ja ajatemplitega. `-v` näitab tehtavat, `-n` teeb kuivkäivituse ilma sihti muutmata.

```
mkdir siht
rsync -avn allikas/ siht/
rsync -av allikas/ siht/
```

Esimene käsk näitab plaani, teine teeb koopia. Vaata `cat siht/alam/fail.txt` abil sisu. Korda sama päris ülekannet: muutumatu faili sisu ei pea uuesti saatma.

Vaikimisi otsustab `rsync` ülekanne vajaduse peamiselt **faili suuruse ja muutmisaja** järgi, mitte kõiki sisusid läbi lugedes. Valik `-c` võrdleb kontrollsummasid, kuid nõuab failide lugemist ja võib olla aeglasem. `-a` ei säilita automaatselt kõiki võimalikke metaandmeid, näiteks ACL-e, laiendatud atribuute ega kõvalinkide seoseid.

Allika lõpu kaldkriips muudab tulemust

Võrdle olemasoleva sihtkataloogi puhul:

Kuju	Kuhu näitefail jõuab
<code>rsync -av allikas/ siht/</code>	<code>siht/alam/fail.txt</code>
<code>rsync -av allikas siht/</code>	<code>siht/allikas/alam/fail.txt</code>

Esimeses kantakse üle **sisu**, teises **kataloog koos nimega**. Võrdle neid enne `-n`-iga. Ära eelda, et `cp` ja `scp` kasutavad lõpu kaldkriipsu kõigis süsteemides täpselt sama reegli järgi.

Vaikimisi ei eemalda `rsync` sihtkoha lisafaile. **--delete** eemaldab sihist ülekantava puu ulatuses failid, mida allikas pole; see ei ole tavalise varukoopia jaoks automaatselt vajalik. Kontrolli sellist plaani eriti hoolikalt.

Üle SSH teise masinasse

Asenda `kasutaja@server` oma kontoga. Järgmised on käsukujud, mitte toimiva näidiskonto aadressid:

```
scp koopia.txt kasutaja@server:~/  
rsync -avn allikas/ kasutaja@server:~/allikas/
```

`scp -r` kopeerib kataloogi; `scp -p` säilitab õigused ning ajad. `rsync` peab olema mõlemas masinas olemas. Pärast kuivkäivituse kontrolli eemalda `-n`, kui soovid päris ülekannet. Sisselogimise ja võtmete juhis tuleb SSH peatükis.

Tavakasutaja ei saa üldjuhul määrata failile suvalist omanikku. Metaandmete säilimine sõltub ka sihtfailisüsteemist ja kasutaja õigustest.

Veebist faili toomine

`curl -f` märgib HTTP veavastuse ebaeduks, `-L` järgib ümbersuunamisi ning `-O` võtab kohaliku nime **URL-i viimasest teeosast**, mitte serveri suvalisest pakutud nimest.

Vali üks allalaadija:

```
curl -fL -O https://artificial.eu/linux/data/generated-words.txt
```

Või, kui `wget` on olemas:

```
wget https://artificial.eu/linux/data/generated-words.txt
```

Kontrolli head `generated-words.txt` abil, et said teksti, mitte sisselogimis- või vealehe. Rohkem valikuid leiad veebist sisu toomise peatükist.

Minitest

1. Lisa allikafaili üks rida. Mida näitab uus `rsync -avn`?
2. Kuidas muudab allika lõpu kaldkriips tulemuse kataloogipuud?
3. Miks `rsync -av` pole sama mis kahe kataloogi täielik peegeldamine?
4. Mida pead üle kontrollima enne faili teise masinasse kopeerimist?

Spikker

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 kataloog
- `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://artifical.eu/linux/data/generated-words.txt` — laadi fail
- `curl -fLO https://artifical.eu/linux/data/generated-words.txt` — järgi suunamisi, HTTP-vea korral ebaõnnestu

Olulisemad lipud, märgid ja kiirnupud

- `cp -R` — kopeeri kataloog
- `scp -r` — saada kataloog
- `rsync -a` — rekursiivselt, säilitades tavalised Unixi metaandmed
- `rsync -n` — proovi enne
- `curl -L` — järgi suunamist
- `curl -O` — salvesta URL-i lõpuosa nimega

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

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. määra korduvale ühendusele failis `~/.ssh/config` alias
6. macOS-is lase Apple'i võtmehoidjal (Keychain) võtmefraas meelde jätta

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	<code>ssh</code>	kaugmasina viip või võtme
millised võtmefailid on olemas	<code>ls -la ~/.ssh</code>	<code>id_ed25519</code> , <code>id_ed25519.pub</code> jms
mis võtmed agendis on	<code>ssh-add -l</code>	võtmete sõrmejäljed või teade, et agent on tühi
loo uus võti	<code>ssh-keygen -t ed25519</code>	privaat- ja avaliku võtme failid
lisa võti macOS-i võtmehoidjasse	<code>/usr/bin/ssh-add --apple-use-keychain ~/.ssh/id_ed25519</code>	võtmefraas salvestatakse võtmehoidjasse
kasuta aliasi	<code>ssh opik-server</code>	ühendus ~/.ssh/config kirje järgi

Esmakordsel ühendusel küsib SSH sageli serveri sõrmejälje kinnitamist. See on serveri tuvastus, mitte sinu kasutajavõti. Kontrolli uut sõrmejälge serveri haldurilt või teisest usaldusväärsest kanalist enne kinnitamist; pelgalt õige aadress ei tõenda serveri ehtsust.

Tüüpilised algaja vead

- aetakse segi privaatvõti ja avalik võti
- kirjutatakse olemasolev võti kogemata üle
- arvatakse, et Apple'i võtmehoidja asendab ~/.ssh võtmefailid
- lisatakse serverisse privaatvõti, kuigi lisada tuleb ainult .pub fail
- seadistatakse keeruline pordi- või agendiedastus enne, kui tavaline sisselogimine töötab
- unustatakse, et faili ~/.ssh/config aliaseid saavad kasutada ka scp, rsync ja Git

Esimene ühendus

Kõigepealt proovi kõige tavalisemat kuju:

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

Pärast sisselogimist käivad käsud kaugmasina kohta. Tagasi kohalikku shelli saad käsuga `exit`.

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 aliases 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 hoi 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: määra privaativõtmele võtmefraas. See kaitseb võtit ka siis, 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, kasuta käsus vastavat failinime.

macOS: võtmehoidja ja ssh-agent

macOS-is on kaks eri asja:

- **ssh-agent** hoiab võtmeid jooksva kasutusseansi ajal mälus
- Apple'i võtmehoidja (Keychain) saab hoida privaativõtme võtmefraasi, et sa ei peaks seda iga kord uuesti sisestama

Võtmehoidja ei asenda võtmefaile. Privaativõti on endiselt näiteks ~/.ssh/id_ed25519; võtmehoidja aitab selle võtmefraasi hallata.

Kasuta Apple'i tööriista /usr/bin/ssh-add: näiteks Homebrew kaudu paigaldatud samanimeline programm ei pruugi Apple'i valikuid toetada. Lisa võti agendile ja salvesta võtmefraas võtmehoidjasse:

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

Kui võtmefraas on juba võtmehoidjas ja tahad võtmed agendile laadida:

```
/usr/bin/ssh-add --apple-load-keychain
```

Kui tahad võtme ainult jooksvast agendist eemaldada:

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

Kui tahad eemaldada ka võtmehoidjasse salvestatud võtmefraasi, kasuta eemaldamisel Apple'i võtmehoidja valikut:

```
/usr/bin/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 agendi ei saa ühendada, ava uus terminaliaken ja proovi uuesti. macOS-is käivitab graafiline kasutajaseanss agendi tavaliselt ise.

~/.ssh/config: aliased ja püsiseaded

Kui ühendud sama serveriga korduvalt, määra seadistusfailis 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 rida `UseKeychain`, sest Linux OpenSSH ei pruugi Apple'i valikut tunda. Nii töötab sama seadistusfail paremini 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 aliasega

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 eelistatakse 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 seadistusfaili 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:

- pordiedastus (`port forwarding`): `ssh -L ...`
- agendiedastus (`agent forwarding`): `ssh -A ...`
- hüppemasinad: `ProxyJump`
- püsivad ühendused: `ControlMaster`

Need on kasulikud, aga tulevad pärast seda, kui tavaline sisselogimine, võtmed ja ~/.ssh/config aliased 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 /usr/bin/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.

Spikker

Eesmärk: tee SSH korduvkasutatavaks: kontrolli olemasolevaid võtmeid, lisa avalik võti serverisse, kasuta failis ~/.ssh/config määratud aliaist ja macOS-is Apple'i võtmehoidjat

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
- `/usr/bin/ssh-add --apple-use-keychain ~/.ssh/id_ed25519` — macOS: salvesta võtmefraas võtmehoidjasse
- `ssh-copy-id -i ~/.ssh/id_ed25519.pub kasutaja@server.example.org` — lisa avalik võti serverisse
- `ssh opik-server` — kasuta seadistusfaili aliaist
- `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` — võtmefraas macOS-i võtmehoidjas
- `IdentitiesOnly yes` — kasuta nimetatud võtit

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

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 katkenud allalaadimist
3. vaata HTML-i loetava tekstina või kogu sealt linke

Just selle järgi tasub tööriist valida:

- **curl** ühe päringu, päiste, API või toor-HTML jaoks
- **wget** allalaadimise, jätkamise ja veebis roomamise jaoks
- **lynx** HTML-i tekstivaate ja linkide loendi jaoks

Veebikraapimine (**scraping**) tähendab siin veebist saadud sisu edasist töötlemist käsureal. Kui ametlik API või andmefail on olemas, 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 alustada ühe URL-iga, kontrollida tulemust ja alles siis mõelda ulatuslikumale roomamisele.

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

- alustatakse kohe ulatuslikku roomamist, 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

Kui tööriist puudub

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

Vali ainult oma süsteemile sobiv paigalduskäsk:

Debianis või Ubuntu:

```
sudo apt install wget lynx
```

Fedoras:

```
sudo dnf install wget lynx
```

macOS-is Homebrew kaudu:

```
brew install wget lynx
```

`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 režiim
- `-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 vaikinisi väljundi ekraanile
- seetõttu sobib ta hästi torude ja filtritega
- `curl` ise ei tee „veebipuud“, vaid ühe või mõne konkreetse päringu

`wget`: allalaadija ja veebis roomaja

`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 jätkka katkenud allalaadimist
- -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 aeglusta roomamist

Näited:

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

Piiratud ulatusega roomamine:

```
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 suure saidi juures 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 eraldi

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 HTML on juba failis, saad sama teha kohalikult:

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

Vahe on lihtne:

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

Väike veebikraapimise 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 allalaadimiseks ja veebis roomamiseks
- `lynx` ei ole roomaja, vaid tekstivaate ja linkide tööriist
- HTML-i kraapimine on habras, sest lehe struktuur võib muutuda
- JavaScriptiga loodud sait ei pruugi anda `curl`-ile või `wget`-ile sama sisu, 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

- ekspordifaili, näiteks 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`.

Spikker

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 -fsSL https://example.com/ | lynx -stdin -dump` — loe tekstina
- `lynx -dump -listonly https://example.com/` — kogu lingid

Olulisemad lipud, märgid ja kiirnupud

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

Pane tähele: Alusta alati ühest URL-ist; ulatuslik veebis roomamine ja automaatne veebikraapimine olgu järgmine, mitte esimene samm.

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

Tähtsamad valikud

- **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 Unixi 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 juurkataloog
- 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.

Spikker

Eesmärk: arhiiv koondab failid; tihendamine võib mahtu vähendada, kuid juba pakitud sisu puhul mitte tingimata

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` — paki kataloog ZIP-arhiivi
- `unzip -l` — vaata zip sisu

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

Tervete kataloogipuude haldus ja jagamine

Üksikfaili kopeerimise asemel tuleb projekti jagamisel otsustada, **mis kaasa läheb ja kuidas tulemust kontrollida**. Siinne harjutus ühendab sünkroonimise ja arhiivid; üksikkäskude lipud leiad neist peatükkidest.

Vali eesmärgi järgi

Vajadus	Sobiv viis	Piirang
Proovi muudatust kohalikus koopias	<code>cp -R</code>	sama nimega sihtfail võib üle kirjutuda
Uuenda olemasolevat koopiat	<code>rsync</code> , algul proovikäivitus	sünkroonimine võib kopeerida ka eksimuse
Saada üks kindel hetkeseis	<code>tar.gz</code> või ZIP	järgmist versiooni jagad eraldi
Tee koostööd ja jälgi muudatusi	Git	ei talleta kõiki faile ega failisüsteemi metaandmeid

Varukoopia pole lihtsalt teine kataloog samal kettal. Kettarikke vastu aitab eraldi andmekandja või masin; kogemata muudetud faili taastamiseks on vaja ka varasemaid versioone.

Koosta jagatav koopia

Harjutame enda loodud failidega. Punktiga algav `.env` tähistab siin seadistus-faili, milles päris projektis võivad olla saladused. Näites saladusi ei ole.

```
mkdir -p ~/tmp/jagamise-naide/projekt/cache
cd ~/tmp/jagamise-naide
printf '%s\n' 'Projekti juhend' > projekt/README.txt
printf '%s\n' 'AINULT_NAIDIS=1' > projekt/.env
printf '%s\n' 'taaslloodav sisu' > projekt/cache/ajutine.txt
```

`rsync --exclude` jätab mustriks vastavad kirjed ülekandest välja. Allika lõpu / valib kataloogi **sisu**. `-avn` säilitab tavalised Unixi metaandmed, loetleb kavatatud töö ja teeb ainult proovikäivituse:

```
rsync -avn --exclude='.env' --exclude='cache/' projekt/ jagamiseks/
```

Loendis peab olema `README.txt`, kuid mitte `.env` ega `cache/ajutine.txt`. Kui tulemus sobib, korda sama käsku ilma `n`-ita:

```
rsync -av --exclude='.env' --exclude='cache/' projekt/ jagamiseks/
ls -la jagamiseks
```

Väljajätmine ei kustuta varem sihtkohta sattunud saladusi. Kasuta jagamiseks uut puhast kataloogi. Ära lahenda seda pimesi valikuga `--delete`.

Kontrolli ka taastamist

Paki jagatav koopia arhiivi, vaata liikmete loendit ja ava arhiiv eraldi kataloogi. `-C` määrab olemasoleva lahtipakkimiskoha.

```
tar -czf jagamiseks.tgz jagamiseks/
tar -tf jagamiseks.tgz
```

```
mkdir taastatud
tar -xzf jagamiseks.tgz -C taastatud
diff -r jagamiseks taastatud/jagamiseks
```

`diff -r` võrdleb kataloogide faile rekursiivselt. Vaikne väljund tähendab siin, et failisisus erinevusi ei leitud; see ei kinnita kõigi õiguste, omanike ega ajatemplite võrdsust. Järgmine kord vali uus taastamiskataloog, mitte ära kirjuta vana testi üle.

`rsync -a` ei hõlma automaatselt kõiki ACL-e, laiendatud atribuute ega kõvalinke. Git omakorda ei säilita näiteks failide omanikke, ajatemplite ajalugu ega tühje katalooge. Täpseks süsteemivarunduseks vali selleks mõeldud lahendus ja testi taastamist.

Minitest

1. Lisa projekti teine tekstifail. Kas proovikäivitus leiab ainult vajaliku täienduse?
2. Muuda taastatud koopias üht rida. Milline `diff -r` väljund tekib?
3. Miks ei tõesta sama failide arv või kataloogimaht veel koopiade võrdsust?
4. Millist varundust vajaksid, et taastada eiline versioon pärast tänast eksimust?

Spikker

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

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/` — `rsync`-allikas: kataloogi sisu
- `projekt` — `rsync`-allikas: kataloog ise
- `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.

Protsessid, tööd ja signaalid

Protsess on töötav programmi eksemplar, millel on süsteemi protsessinumber ehk **PID**. Shelli **töö** võib koosneda ühest või mitmest protsessist, näiteks torust. **Signaal** on protsessile saadetak juhtsõnum.

Vaata, mis töötab

Käsk	Tulemus
<code>ps</code>	selle terminaliga seotud protsesside hetkeseis
<code>ps aux</code>	laiem loend koos omaniku, PID-i, CPU ja mälu kasutusega
<code>top</code>	uuenev protsessivaade; välju <code>q</code> -ga
<code>jobs -l</code>	selle shelli töönumbrid, seisud ja PID-id

`htop` on eraldi paigaldatav mugavam jälgimisvaade. CPU ja mälu näitude tähendus ning arvutusvahemik sõltuvad tööriistast; üks hetkepilt ei tõesta pikaajalist koormust.

`ps aux` tavaväljundis on kolmas veerg CPU protsent ja neljas mälu protsent. `sort -nrk` sordib valitud veeru arvuliselt kahanevaks; `head` jätab kümme esimest rida:

```
ps aux | sort -nrk 3 | head
ps aux | sort -nrk 4 | head
```

Need annavad kiire ülevaate, kuid päiserida ei püsi eraldi päisena. Pikema jälgimise jaoks kasuta `top`-i.

Järjest või korraga

`sleep N` ootab N sekundit. Tavaliselt ootab shell käsu lõppu enne järgmise alustamist; `;` eraldab järjestikuseid käske.

```
sleep 3; echo 'pikk töö valmis'
sleep 1; echo 'lühike töö valmis'
```

Esimene teade ilmub umbes kolme, teine nelja sekundi järel. Nüüd võrdle taustatöödega: `&` lubab shellil kohe edasi minna, `sh -c` käivitab jutumärkides oleva käsujada eraldi shellis ja `wait` ootab oma taustatööde lõppu.

```
sh -c 'sleep 3; echo "pikk töö valmis"' &
sh -c 'sleep 1; echo "lühike töö valmis"' &
jobs
wait
```

Tavaliselt ilmub hiljem alustatud **lühikese töö** teade esimesena ning mõlemad lõpevad umbes kolme sekundiga. Käivitamisjärjekord ei määra lõpetamisjärjekorda. Kui sisestad ridu aeglaselt, võib töö enne `jobs`-i juba valmis olla.

Peata ja jätkka sama tööd

Tee see katse **interaktiivses shellis**, mitte ühe kopeeritud skriptina.

1. Käivita `sleep 120`, vajuta `Ctrl-z`. Programm peatub ja viip tuleb tagasi.
2. Käivita veel kord `sleep 120` ning vajuta jälle `Ctrl-z`.
3. Vaata `jobs -l`. Otsi oma kahe `sleep`-töö numbrid.
4. Kui need on 1 ja 2, jätkka esimest käsuga `bg %1` ja too teine ette käsuga `fg %2`.
5. Katkesta esiplaani töö `Ctrl-c`-ga. Too esimene ette `fg %1` ja katkesta ka see.

Kasuta tegelikke töönumbreid, mitte tingimata 1 ja 2. %1 on praeguse shelli töönumber; paljas number käsule `kill` tähendab PID-i. `bg` ja `fg` jätkavad sama programmi, mitte ei käivita seda algusest.

Lõpetamine ja signaalid

Tegevus	Mõju
<code>Ctrl-c</code>	saadab esiplaani protsessigrupile tavaliselt <code>SIGINT</code> katkestussoovi
<code>Ctrl-z</code>	saadab tavaliselt <code>SIGTSTP</code> ajutise peatamise signaali
<code>kill PID</code>	vaikimisi <code>SIGTERM</code> : programm saab viisakalt lõpetada
<code>kill -KILL PID</code>	sunnitud lõpetamine; programm ei saa enam koristada ega salvestada

Ära proovi suvalist PID-i. Harjutuseks käivita oma ootaja ja salvesta `$!` kaudu viimase taustaprotsessi PID muutujasse:

```
sleep 120 &
ootaja=$!
ps -p "$ootaja"
kill "$ootaja"
```

Võid näha shelli teadet **Terminated**. Sunnitud lõpetamine on viimane abinõu, mitte veaotsingu algus.

Väike kohalik server

Python peab olema paigaldatud. Loo eraldi kataloog, milles pole isiklikke faile. `--bind 127.0.0.1` lubab ühendusi ainult sellest arvutist; muidu kuulab Python serveri vaikimisi ka võrguliideseid.

```
mkdir -p ~/tmp/serveri-naide
cd ~/tmp/serveri-naide
```

```
printf '%s\n' 'Tere kohalikust serverist!' > index.html
python3 -m http.server 8000 --bind 127.0.0.1 &
serveri_pid=$!
```

Ava brauseris localhost:8000. Kui port on hõivatud, vali teine port nii käsus kui aadressis. Lõpetamiseks kasuta **samas shellis** `kill "$serveri_pid"`. See lõpetab ainult sinu salvestatud PID-iga protsessi.

Kas töö jääb akna sulgemisel alles?

& üksi seda ei taga. Interaktiivse kaugtöö jaoks kasuta tmux-i või screeni.

`nohup` paneb käivitatava programmi eirama ühenduse katkemisega seotud SIGHUP-signaali. Suuna ka sisend ja väljund terminalist eemale:

```
nohup sh -c 'sleep 10; echo "valmis"' < /dev/null > nohup-naide.log 2>&1 &
```

Kümne sekundi pärast peaks logis olema **valmis**. `nohup` ei kaitse taaskäivituse, voolukatkestuse ega administraatori sessioonipuhastuse eest.

Bashi ja `zsh` `disown` eemaldab töö shelli tööde loendist; see ei vabasta programmi automaatselt terminalisisendist ega -väljundist. Kasuta seda vaid siis, kui tead programmi sõltuvusi.

Minitest

1. Muuda paralleelkatse ooteajad viieks ja kaheks sekundiks. Ennusta järjekutuse ja samaaegse variandi kogukestus.
2. Miks ei näe uus terminaliaken vana akna `jobs`-loendit?
3. Mis vahe on peatatud ja lõpetatud programmil?
4. Käivita kohalik server, vaata selle PID-i ning lõpeta ainult see protsess. Kontrolli brauseris, et teenus enam ei vasta.

Spikker

Eesmärk: erista protsessi, shelli tööd ja signaali: vaata, millised protsessid töötavad, peata vajadusel ning jätkata pikka tööd taustal

Põhikujud

- `ps aux | grep python` — otsi protsess
- `ps aux | sort -nrk 3 | head` — suur CPU kasutus selles hetkepildis
- `ps aux | sort -nrk 4 | head` — suur mälu kasutus
- `sleep 300 &` — käivita taustal
- `jobs` — vaata töid
- `fg %1` — too töö ette
- `top` — jälgi reaalajas
- `kill` — saada signaal
- `bg` — jätkata 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.

Logid ja teenused

Teenus on süsteemi hallatav taustatöö; selle olek ütleb, kas see töötab, **logi** aitab mõista juhtunut. Üks teenus võib käivitada mitu protsessi. Alusta olekust, seejärel vaata sama teenuse hiljutisi teateid.

systemd-ga Linux

Järgmised käsud vajavad **systemd-d**. Need ei ole macOS-i käsud ega tööta igas Linuxi konteineris või WSL-i seadistuses. **systemctl** uurib teenust, **journalctl** loeb süsteemipäevikut.

Käsk	Mida näed
systemctl status ssh	teenuse olek, käivituse andmed ja viimased teated
journalctl -u ssh -n 50	-u valib teenuse, -n viimaste kirjete arvu
journalctl -u ssh --since today	kirjed alates tänase päeva algusest
journalctl -u ssh -f	-f jääb lisanduvaid kirjeid ootama; lõpeta Ctrl-c -ga

SSH-teenus võib olla nimega **sshd**, mitte **ssh**, või üldse paigaldamata. Asenda näites nimi oma teenusega. **Unit ... could not be found** ei tähenda sama mis olemasoleva teenuse **failed**-olek. Lehitsetavast vaatest välju **q**-ga.

Olek **active** ja käivitamisel lubatud olek **enabled** ei ole sama: teenus võib praegu töötada, kuid mitte järgmisel alglaadimisel. **-- No entries** -- ei tõesta vigade puudumist; põhjuseks võivad olla vale nimi, piiratud lugemisõigus või teistsugune logimine.

macOS kasutab teenuste halduseks **launchd-d** ja süsteemilogide jaoks muu hulgas käsku **log show**. Tavalise tekstilogiga saad mõlemas süsteemis teha järgmise katse.

Vaata logi kasvamist kahes aknas

Loo esimeses **terminalis** oma näidislogi. **printf** kirjutab ühe rea, > kirjutab samanimelise faili üle. Ära kasuta selleks päris teenuse logi.

```
mkdir -p ~/tmp/logi-naide
cd ~/tmp/logi-naide
printf '%s\n' 'INFO teenus käivitus' > app.log
tail -f app.log
```

Näed algset rida ja käsk jääb ootama. **Teises terminalis** kasuta täisteed, sest uus aken ei pruugi olla samas kataloogis. >> lisab lõppu, mitte ei kirjuta algust üle:

```
printf '%s\n' 'WARN ühendus aeglane' 'ERROR ühendus katkes' >> ~/tmp/logi-naide/app.log
```

Esimesse aknasse ilmuvad uued read. Katkesta seal jälgimine **Ctrl-c**-ga; see ei kustuta logi ega lõpeta seda kirjutanud teenust. Käsk **tail -n 2 app.log** näitab nüüd ainult kaht viimast rida.

Leia oluline sündmus

Päris logis on tavaliselt ajatempel, tähtsusaste ja sõnum. Märksõnad nagu **ERROR**, **failed** ja **denied** aitavad alustada, kuid **loe ka ümbritsevaid ridu**: viimane viga võib olla varasema vea tagajärg.

Otsingu peatükis õpid eraldama veakirjeid käsuga **grep**. Suuremat näidislogi pakub õpiku andmefail.

Süsteemi tekstilogid võivad asuda näiteks **/var/log/syslog** või **/var/log/messages**, kuid need failid ei pea sinu süsteemis olemas olema. Ära loo puuduvat süsteemilogi käsitsi ega muuda päris logide õigusi lihtsalt vaatamise hõlbustamiseks.

Minitest

1. Lisa näidislogisse kolmas sündmus ja vaata, kuidas see esimesse aknasse ilmub.
2. Kuidas erinevad **tail -n 2** ja **tail -f**?
3. Kui kasutad **systemd**-ga Linuxit, leia üks olemasolev teenus ja võrdle selle olekut viimaste logiteadetega.
4. Miks ei tõesta tühi logivaade, et teenus töötab veatult?

Spikker

Eesmärk: Kontrolli teenuse olekut ja seejärel sama teenuse logi. **systemctl/journalctl** vajavad **systemd**-ga Linuxit.

Käsud ja näited

- `systemctl status ssh` — teenuse seis ja hiljutised teated; nimi võib olla `sshd`
- `journalctl -u ssh -n 50` — selle teenuse viimased 50 kirjet
- `journalctl -u ssh --since today` — tänased teenusekirjed
- `journalctl -u ssh -f` — jälgi juurde tulevaid kirjeid
- `tail -n 20 app.log` — tavalise logifaili lõpp, ka ilma `systemd`-ta
- `tail -f app.log` — jälgi faili; Ctrl-c lõpetab jälgimise

Pane tähele: Logi puudumine ei tõesta vigade puudumist: kontrolli teenuse nime, logimise seadistust ja lugemisõigust.

Püsivad terminalisessioonid: tmux ja screen

tmux hoiab shelli ja selles käivitatud töö serveris alles ka pärast SSH-ühenduse katkemist. Hiljem saad sama sessiooni juurde tagasi tulla. Käivita **tmux** sellel masinal, kus töö peab jätkuma, kaugkasutuses seega pärast serverisse logimist.

See ei kaitse serveri väljalülitamise, taaskäivituse ega protsessi krahhi eest. Taus-tamärk & ükski ei anna samuti püsivat terminalisessiooni.

Katse: lahku ja tule tagasi

`tmux new -s opik` loob nimega `opik` sessiooni ja avab selle sees uue shelli. Kui sessioon on juba olemas, vali teine nimi või ühendu olemasolevaga.

```
tmux new -s opik
```

Nüüd oled **sessiooni sees**. Käivita lühike ootamine:

```
sleep 60
```

Vajuta **Ctrl-b**, lase klahvid lahti ja vajuta **d**. See on eraldumine (*detach*), mitte väljumine: `sleep` jätkab tööd.

Nüüd oled **sessioonist väljas**. `tmux ls` loetleb sessioonid; `attach -t` valib sessiooni, mille juurde naasta:

```
tmux ls
tmux attach -t opik
```

Kui minut pole möödunud, ootab `sleep` endiselt. Muidu näed taas viipa. Vajadusel katkesta ootamine **Ctrl-c**-ga. `exit` lõpetab sessioonis oleva shelli; kui see oli viimane aken, lõpeb ka sessioon. Ära aja `exit`-it segi eraldumisega.

Sama tööviis screen-iga

Kui masinas on **tmux** asemel **screen**, kasuta neid vasteid:

Toiming	screen-i kuju
loo nimega sessioon	screen -S opik
eraldu sessiooni seest	Ctrl-a , seejärel d
loetle väljast sessioonid	screen -ls
naase väljast sessiooni	screen -r opik

Vali üks tööriist; mõlemat pole vaja sama katse sees käivitada.

Sessioon või üks taustatöö?

tmux ja **screen** säilitavad terminali, millega saab uuesti ühenduda. Protsesside peatüki **nohup** muudab ühe töö käitumist ühenduse katkemisel ning **disown** eemaldab töö shelli tööde loendist; kumbki ei anna sama terminalivaadet tagasi.

Minitest

1. Loo nimega sessioon, eraldu ja ühendu uuesti.
2. Millisel masinal peab sessioon töötama, et sülearvuti sulgemine serveritööd ei lõpetaks?
3. Mis vahe on eraldumisel, **exit**-il ja serveri taaskäivitamisel?

Spikker

Eesmärk: Säilita interaktiivne serverisessioon ka SSH-ühenduse katkemisel.

Käsud ja näited

- **tmux new -s katse** — loo nimega sessioon
- **tmux ls** — loetle sessioonid välisest shellist
- **tmux attach -t katse** — ühendu olemasoleva sessiooniga
- **screen -S katse** — alternatiiv: loo screeni sessioon
- **screen -r katse** — taasühendu eraldatud screeniga

Märgid ja valikud

- **Ctrl-b**, siis **d** — **tmux**: eraldu, töö jätkub
- **Ctrl-a**, siis **d** — **screen**: eraldu, töö jätkub
- **exit** — lõpeta praeguse akna shell

Pane tähele: Sessioon ei ela üle serveri taaskäivitust ega volukatkestust.

Graafilised rakendused kaugmasinast

Selles peatükis vaatame, millal kasutada X11-edastust, millal veebiliidest ja mis on praktilised piirangud.

Loogika

Kaugelt graafilise rakenduse kasutamiseks on mitu võimalust, kuid need ei ole võrdselt mugavad. Enamasti tasub eelistada veebiliidest või IDE Remote SSH lahendust ning jätta X11-edastus 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
<code>ssh -X</code>	saada X11-rakenduse aken sinu masinasse	terminal on vaikne, aken võib avaneda eraldi
<code>ssh -L</code>	too serveri port oma <code>localhost</code> aadressile	SSH-seanss jääb lahti, brauser kasutab kohalikku aadressi
Remote SSH IDE-s	redigeeri oma arvutis, käivita serveris	IDE-s serveri failipuu ja terminal
veebiliides	jäta graafika brauseri tööks	HTTP-aadress brauseris

Tüüpilised algaja vead

- proovitakse X11-edastust enne, kui tavaline `ssh` töötab
- unustatakse, et pordiedastuse käsus ja brauseris viitavad kaks `localhost`-i eri masinatele
- arvatakse, et GUI-edastus on alati parim tee, kuigi veebiliides või Remote SSH on sageli kiirem ja töökindlam
- jäetakse pordiedastus liiga laialt avatuks; alguses hoia seos ainult oma masinaga

Käivita need käsud

```
ssh -X kasutaja@server
```

Veebiliidese puhul peab serveris juba töötama teenus pordil 8888. Järgmine käsk käivitatakse sinu arvutis: `-L` teeb pordiedastuse, `127.0.0.1` piirab kohaliku kuulamise selle arvutiga ja `-N` jätab kaugshelli avamata.

```
ssh -N -L 127.0.0.1:8888:localhost:8888 kasutaja@server
```

Hoia ühendus lahti ning ava kohalikus brauseris `localhost:8888`. Tunneli loomine ei käivita teenust ega asenda selle autentimist. Lõpeta tunnel `Ctrl-C`-ga.

Käsu ja brauseri kaks `localhost`-i viitavad eri masinatele:

- käsus olev `localhost:8888` on serveri enda vaade teenusele

- brauseris avatav `http://localhost:8888` on sinu arvuti vaade edasisuunatud pordile

X11-edastus

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 masinas on X11-server ja kaugmasinas sobiv X11-rakendus ning lubatud X11-edastus.
2. Pane kirja üks juhtum, kus veebiliides on mõistlikum kui X11.
3. Selgita, miks Remote SSH võib olla arenduses mugavam kui vahetu X11-edastus.

Spikker

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

Põhikujud

- `ssh -N -L 127.0.0.1:8888:localhost:8888 kasutaja@server` — tunnel juba töötava serveriteenuseni
- `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.

Teksti otsimine: grep ja sugulased

grep valib sisendist mustriga sobivad read. Ta ei muuda faili ega kuva vaikimisi ainult leitud sõna: välja tuleb terve rida. Sobimatu tähesuurus või mustri erimärk võib tulemust oluliselt muuta.

Esimene võrdlus

Valmista enda harjutuskataloogis väike fail. > kirjutab sama nimega faili üle.

```
printf 'kass\nkoer\nKass\nkassipoeg\n' > loomad.txt
grep 'kass' loomad.txt
```

Näed kass ja kassipoeg, mitte Kass. Otsing võib leida vaste ka pikema sõna seest.

Valik	Tähendus	Katse
-i	eira tähesuurst	grep -i 'kass' loomad.txt leiab ka Kass
-n	lisa reanumber	grep -n 'koer' loomad.txt annab 2:koer
-x	muster katab terve rea	grep -x 'kass' loomad.txt jätab kassipoeg välja
-v	vali mittesobivad read	grep -iv 'kass' loomad.txt jätab ainult koer
-c	loenda vastega ridu	grep -c 'kass' loomad.txt annab 2

Lipud võivad koos töötada. -c ei loenda kõiki sõna esinemisi: kui ühel real on kolm vastet, on see ikkagi üks vastega rida.

Vasteta otsing annab tavaliselt tühja väljundi ja lõpetuskoodi 1; leitud vaste annab 0, viga tavaliselt 2. Seega ei tähenda iga nullist erinev kood siin riket.

Täpne tekst või regulaaravaldis

Vaikimisi on muster **regulaaravaldis**: näiteks punkt . tähendab üht suvalist märki. -F lülitab mustrikeele välja ja otsib täpset teksti. See ei nõua veel tervet rida; selleks lisa -x.

```
printf 'fail.json\nfailXjson\n' > nimed.txt
grep '.json' nimed.txt
grep -F '.json' nimed.txt
```

Esimene leiab mõlemad read, teine ainult `fail.json`. Shelli jutumärgid hoiavad mustri ühe argumendina, kuid ei lülita `grep`-i mustrikeelt välja.

Muster	Tähendus
<code>^#</code>	rea alguses on <code>#</code>
<code>json\$</code>	rida lõpeb tekstiga <code>json</code>
<code>.</code>	üks suvaline märk
<code>[0-9]</code>	üks number
<code>.*</code>	null või rohkem suvalisi märke
<code>[0-9]+</code> koos <code>-E</code> -ga	vähemalt üks number

`-E` valib laiendatud regulaaravaldised. Vanade käsunimede `egrep` ja `fgrep` asemel kasuta `grep -E` ja `grep -F`.

Otsi kataloogipuust

`-r` otsib rekursiivselt. Anna nii muster kui ka alguskataloog; suurtes puudes võib otsing aega võtta.

```
grep -rn 'TODO' .
```

Tulemuse ees on failinimi ja reanumber. GNU `grep`-i `-R` järgib lisaks otsingu käigus leitud sümbollinke; `-r` ja `-R` ei ole lihtsalt kaks kirjepilti samale lipule. Ära otsi alguseks kogu juurkataloogist ega peida ligipääsuveateid.

Sõnamõistus, mille tulemust saab kontrollida

`-x 'a.t.l'` nõuab viiemärgilist rida: esimene märk `a`, kolmas `t` ja viies `l`. Tee väike kontrollitav sõnaloend:

```
printf '%s\n' antal aptal artal artel astel attal axtel kass > words.txt
grep -x 'a.t.l' words.txt
```

Vasteks on esimesed seitse rida, mitte `kass`. Suuremaks otsinguks sobib õpiku genereeritud näidisloend; see ei ole kontrollitud loomuliku keele sõnastik ning vastete arv sõltub andmetest.

Jätk pärast sortimise peatükki

See katse kasutab ka sortimist ja koondamist. `grep -o .` väljastab iga leitud märgi eraldi; `sort` toob võrdsed märgid kokku, `uniq -c` loendab need ja `sort -nr` järjestab arvuliselt kahanevalt:

```
grep -x 'a.t.l' words.txt | grep -o . | sort | uniq -c | sort -nr
```

Tähe `a` loendus on 11, `t` loendus 8 ja `l` loendus 7. Katse ühendab otsingu ja lihtsa statistika. Tähesuuruse ühtlustamist õpetab teksti teisendamine.

Edasijõudnule: korduva tüki leidmine

GNU `grep -E` toetab tagasiviiteid, kuid nende tugi laiendatud regulaaravaldistes pole kõigi tööriistade vahel ühesugune. `(. .)` jätab kahe märgi paari meelde, `\1` viitab samale paarile ja `+` nõuab vähemalt üht kordust.

```
printf '%s\n' aaaaaa a.a.a. ratatat kass | grep -E '(. .)\1\1+'
```

Esimesed kolm rida sobivad: `aa`, `a.` või `at` kordub vähemalt kolm korda. `ratatat` sees sobib osa `atatat`; valikuga `-x` see terve rida enam ei sobiks. Tagasiviited võivad olla aeglased: kasuta neid põhjusega, mitte tavalise tekstihaku asemel.

Minitest

1. Lisa `loomad.txt` lõppu rida `kass kass`. Kuidas muutub `grep -c` tulemus?
2. Leia nii `kass` kui `Kass`, kuid mitte `kassipoeg`, ühe käsuga.
3. Miks erinevad `grep '.json'` ja `grep -F '.json'?`
4. Muuda sõnamõistatuse mustrit ning ennusta tulemus enne käivitamist.
5. Kuidas eristad „vastet ei leitud“ ja „faili ei saanud lugeda“?

Lisalugemine

Ametlik GNU `grep`-i juhend ning tekstitöötamise viited.

Spikker

Eesmärk: Vali tekstist sobivad read; erista täpset teksti regulaaravaldisest.

Otsingukuju ja tulemus

- `grep 'kass' loomad.txt` — ka pikema sõna sees; väljastab terve rea
- `grep -in 'kass' loomad.txt` — eira tähesuurust, lisa reanumbrid
- `grep -x 'kass' loomad.txt` — ainult terveniisti sobiv rida
- `grep -v '^#' seadistus.conf` — jätta välja `#` märgiga algavad read
- `grep -F '.json' nimed.txt` — punkt tähendab päris punkti, mitte suvalist märki
- `grep -rn 'TODO' .` — otsi failipuust koos failinimede ja reanumbritega
- `grep -c 'kass' loomad.txt` — loenda vastega ridu, mitte kõiki esinemisi

`grep`-i tulemuskoodid

- 0 — vähemalt üks vaste
- 1 — vasteid pole, mitte tingimata viga
- 2 — tavaliselt viga; loe veateadet

Pane tähele: `-F` otsib täpset teksti ka pikema rea seest. Terve rea sobitamiseks lisa `-x`.

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
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 käsku **cut** keeruka CSV-failiga, kuigi lihtne eraldaja ei pruugi selle töötlemiseks piisata
- oodatakse, et **column** muudab faili, kuigi ta ainult vormib väljundi ilusamaks

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
```

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 sõnavoo, sobib hästi:

```
tr ' ' '\n' < tekst.txt | head -n 20
```

Siin:

- `tr ' ' '\n'` muudab tühikud reavahetusteks
- üks mitmesõnaline rida jaguneb eraldi sõnaridadeks

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 '=' ' '` eraldab võtmed ja väärtused loetavamalt
- `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 süsteemist:

```
strings /bin/ls | head -n 20
```

Siin:

- `/bin/ls` on binaarfail
- `strings` üritab sealt leida loetavaid tekstijuppe

See tuletab meelde, et kõik failid ei ole „lihtsalt tekst“, isegi kui neist saab mõnikord loetavaid tekstijuppe eraldada.

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 CR-märgid (*carriage return*)
- tulemuseks saad puhta LF-iga faili

Kui mõni tööriist käitub “imelikult”, siis põhjus võib olla just reavahetustes, mitte käsus endas.

Täpitähtega teksti puhul kontrolli tulemust. GNU `tr` ei toeta mitmeбайдсeid märke täielikult; UTF-8 teksti õ, ä, ö ja ü tõstu muutmiseks ei saa sellele

üldiselt loota. Siinsed tõstunäited kasutavad ASCII tähti. Unicode'i jaoks sobib näiteks Pythoni sõnemeetod `upper()`.

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.

Spikker

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.

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

Loogika

Kui sul on palju ridu, aitab see peatükk neid kokku võtta. Tüüpiline töövoog on: sorteer, koonda, loenda.

Kiirülevaade

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

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

Tüüpilised algaja vead

- kasutatakse käsku `uniq` sortimata sisendiga ja oodatakse kõigi duplikaatide kadumist
- aetakse segi ridade loendamine ja sõnade loendamine
- unustatakse, et `join` eeldab tavaliselt sobivalt ette valmistatud sisendit

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 loendamine ja tähesuuruse muutmine

```
echo 'Tere tere maailm' | tr '[:upper:]' '[:lower:]' | tr ' ' '\n' | sort | uniq -c
```

See on klassikaline Unix-laadne voog: teisenda, jaga ridadeks, sorteeeri, loenda.

`join` eeldab tavaliselt, et mõlemad sisendfailid on ühise välja järgi sorditud. `pr` on kasulik siis, kui tahad väljundit kiirelt veergudesse või printimiseks vormida.

Näide: kõige sagedasemad 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 töövoog loogika on:

- `sort` toob samad sõnad järjestikku
- `uniq -c` loendab järjestikused kordused
- `sort -nr` paneb suurimad loendused ette

Just see on üks Unix-laadse tekstitötluse 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` võib alguses tunduda abstraktne, kuid sobib hästi 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 käsu `pr` leht“
- see ei tähenda automaatselt üht füüsilist A4-paberilehte

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
```

Lokaat ja sortimine

`sort` ei tööta alati kõigis keskkondades täpselt ühtemoodi, sest tulemus sõltub ka keele- ja piirkonnasätetest (`locale`).

See on eriti nähtav täpitähtede puhul.

Näide:

```
printf 'Õun\nÄmber\nÕö\nUdu\n' > tahed.txt
sort tahed.txt
LC_ALL=C sort tahed.txt
```

Siin võib juhtuda, et:

- tavaline `sort` kasutab sinu keskkonna keele- ja piirkonnasätteid
- `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.

`wc -l` loendab täpselt reavahetusi, mitte visuaalseid ridu. Kui faili viimane tekstirida ei lõpe reavahetusega, jääb see selle loenduse alt välja. `uniq` võrdleb alati kõrvutisi ridu; kogu faili korduste koondamiseks tuleb samad read eelnevalt kõrvuti tuua, tavaliselt `sort`-iga.

Minitest

1. Loenda faili read.
2. Sorteerri 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.

Spikker

Eesmärk: tüüpiline töövoog on: sorteerri 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` ühendab ainult kõrvutised võrdsed read. Kui algjärjekord pole oluline, koonda võrdsed read esmalt käsuga `sort`.

sed, awk ja perl praktiliselt

`sed` sobib asendamiseks, `awk` väljade ja arvutuste jaoks ning `perl` keerukamaks mustritöötluks.

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
<code>sed</code>	muuda ridu mustri järgi	ümber kujundatud tekstivoog
<code>awk</code>	töötle välju ja veerge	valitud või arvutatud väljad
<code>perl</code>	tee keerukam mustri- ja tekstiloogika	skripti või üherealise käsu väljund
faili muutmine	nõuab eraldi kuju	sisendfail ei muutu vaikumisi
süntaksiviga	käsk ei käivitu ootuspäraselt	mustri- või süntaksiveateade

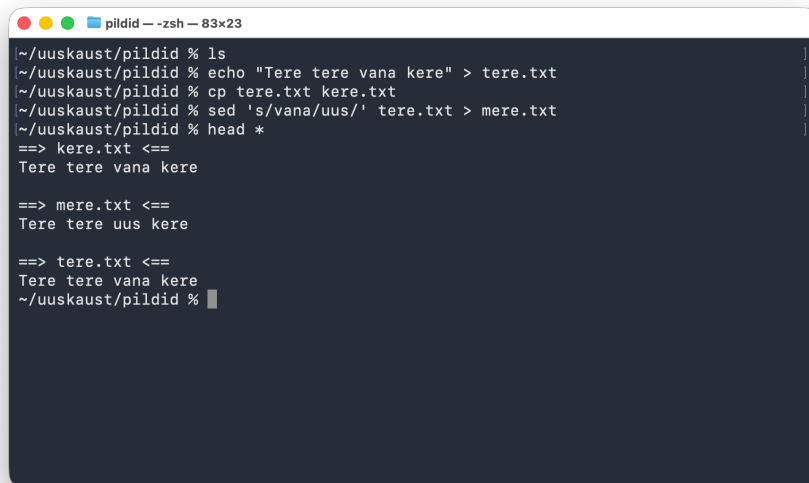
Tüüpilised algaja vead

- oodatakse, et `sed` või `awk` muudab faili kohe kohapeal
- aetakse segi „esimene vaste real“ ja „kõik vasted real“
- kirjutatakse üherealine käsk, mille loogika on endalegi liiga segane

Üks asendus ja kõik asendused

`sed 's/vana/uus/'` asendab iga rea esimese vaste; lõppu lisatud `g` asendab kõik vasted sellel real. Sisendfail ei muutu: tulemus läheb standardväljundisse.

```
echo 'kass koer kass' | sed 's/kass/rebane/'  
echo 'kass koer kass' | sed 's/kass/rebane/g'
```



```
pildid --zsh -- 83x23  
~/uuskaust/pildid % ls  
~/uuskaust/pildid % echo "Tere tere vana kere" > tere.txt  
~/uuskaust/pildid % cp tere.txt kere.txt  
~/uuskaust/pildid % sed 's/vana/uus/' tere.txt > mere.txt  
~/uuskaust/pildid % head *  
==> kere.txt <==  
Tere tere vana kere  
  
==> mere.txt <==  
Tere tere uus kere  
  
==> tere.txt <==  
Tere tere vana kere  
~/uuskaust/pildid %
```

Joonis 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.

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 üksteise järel

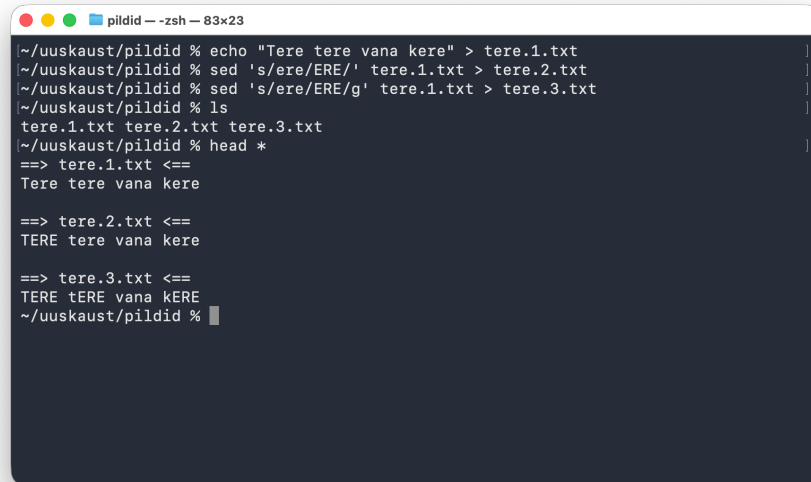
Ära suuna tulemust sisendfaili endasse: `sed 's/vana/uus/' fail > fail` tühjendab faili enne lugemist. Vali uus väljundfail. Kohapealse muutmise `sed -i` kuju erineb GNU ja macOS-i versioonis; kontrolli kohalikku käsiraamatut.

Üks asendus või kõik asendused

`sed`-i üks kõige tähtsamaid erinevusi on see, kas tehakse üks asendus või kõik asendused real.

Selle pildi sees on näha kaks väga erinevat tulemust:

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

A terminal window titled 'pildid -- zsh -- 83x23' showing a series of commands and their outputs. The commands create three files: tere.1.txt, tere.2.txt, and tere.3.txt. tere.1.txt contains 'Tere tere vana kere'. tere.2.txt is created by replacing 'ere' with 'ERE' in tere.1.txt, resulting in 'TERE tere vana kere'. tere.3.txt is created by replacing 'ere' with 'ERE' globally in tere.1.txt, resulting in 'TERE tERE vana kERE'. The terminal shows the use of echo, sed, ls, and head commands.

```
~/uuskaust/pildid % echo "Tere tere vana kere" > tere.1.txt
~/uuskaust/pildid % sed 's/ere/ERE/' tere.1.txt > tere.2.txt
~/uuskaust/pildid % sed 's/ere/ERE/g' tere.1.txt > tere.3.txt
~/uuskaust/pildid % ls
tere.1.txt tere.2.txt tere.3.txt
~/uuskaust/pildid % head *
==> tere.1.txt <==
Tere tere vana kere

==> tere.2.txt <==
TERE tere vana kere

==> tere.3.txt <==
TERE tERE vana kERE
~/uuskaust/pildid %
```

Joonis 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.

2. `sed 's/ere/ERE/g' tere.1.txt > tere.3.txt` muudab kõik vasted samal real
3. `head *` näitab pärast kõiki kolme faili üksteise järel, 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

`awk -F:` valib väljade eraldajaks kooloni. `$1` ja `$2` tähistavad esimest ja teist välja; komaga eraldatud väljad prinditakse vaikimisi tühikuga.

```
printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1}'
printf 'Mari:20\nJaan:21\n' | awk -F: '{print $1, $2}'
```

Perli `-e` võtab koodi käsurealt, `-p` loeb ja väljastab read automaatselt ning `-n` jätab väljastamise sinu koodi otsustada. `-F:` määrab eraldaja, `-a` jagab rea massiivi `@F`, `-l` haldab reavahetusi; `$F[0]` on esimene väli.

```
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]'
```

Edasijõudnule: algarvud regulaaravaldisega

Järgmine näide ei ole algarvude leidmiseks kõige praktilisem, kuid näitab hästi, kui võimsad võivad Perli üherealised käsud olla:

```
seq 2 200 | perl -n1E '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 plokest, 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.

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.

Spikker

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.

`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
<code>find ...</code>	leia kirjeid tingimuste järgi	leitud failide ja kataloogide teed
<code>xargs</code>	anna teed järgmisele käsule	järgmise käsu väljund
<code>-print0</code>	eralda nimed nullmärgiga	masinloetav voog, mitte tavalised read
<code>xargs -0</code>	loe nullmärgiga nimesid	tühikutega nimed jäävad terveks
<code>-exec ... {} +</code>	käivita käsk otse <code>find</code> sees	järgmise käsu väljund

Vale eraldusviis lõpeb sageli veateatega `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

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 käsult `find` 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 alustada ohutu eelvaatega:

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

ja alles siis lisada lõppu faile muutev tegevus.

Veel parem on alustada mittehävitava käsuga, näiteks:

```
find . -type f -name '*.log' -print0 | xargs -0 ls -lh
```

Päris näide: loenda kõik data/ tekstifailid kokku

Repo kataloog data/ on 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: 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.

Tühi tulemus ja suured failihulgad. find ... -exec käsk {} + ei käivita käsku, kui vasteid pole. xargs -0 korral sõltub tühja sisendi käitumine süsteemist; GNU xargs võib käsu siiski käivitada. GNU valik -r väldib seda, kuid pole igal pool sama tähendusega. Suure hulga korral võib nii -exec ... + kui ka xargs jagada failid mitmeks käivituseks: näiteks wc kokkuvõte on siis partii, mitte tingimata kogu puu summa.

`read -r -d ''` näide eeldab Bashi või zsh-d, mitte suvalist POSIX `sh`-d. `IFS=` säilitab nime alguse ja lõpu tühikud, `-r` kurakriipsud ning tühi `-d` määrab nullmärgi eraldajaks.

Minitest

1. Loo vähemalt üks fail, mille nimes on tühik.
2. Leia see fail `find`-iga.
3. Näita seda faili `ls -l` abil läbi töövoog `-print0 | xargs -0`.
4. Seleta ühe lausega, miks `-print0` ja `-0` koos käivad.
5. Loenda kataloogi `data/` tekstifailide read käsuga `find ... -print0 | xargs -0 wc -l`.
6. Tee sama töövoog uuesti kujul `while IFS= read -r -d '' fail; do ...; done`.

Spikker

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` — `find`: ainult tavafailid
- `-name '*.txt'` — nime järgi
- `-print0` — `find`: failinimede vahel nullmärk
- `xargs -0` — `xargs`: eralda argumentid nullmärgiga
- `-exec käsk {} + —` käivita otse
- `read -r -d ''` — Bash/zsh: loe nullmärgini, säilita kaldkriipsud

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

Esimene shelliskript

Shelliskript on käske sisaldav tekstifail. Teeme ühe skripti, mis tervitab kõiki kasurealt antud nimesid ja teatab veast, kui ühtegi nime ei antud.

Kuidas skript käivitub

Esimene rida ehk **shebang**, näiteks `#!/usr/bin/env bash`, määrab tõlgendaja faili otse käivitamisel. `env` otsib Bashi `PATH`-ist. Käivitusõigus ja tõlgendaja valik on eri asjad:

Kuju	Mis juhtub
<code>bash tervita.sh Mari</code>	Bash loeb faili; käivitusõigust pole vaja
<code>chmod u+x tervita.sh, seejärel ./tervita.sh Mari</code>	fail käivitub shebang-rea järgi
<code>sh tervita.sh Mari</code>	valitakse sh , isegi kui shebang nimetab Bashi
<code>bash -n tervita.sh</code>	kontrollitakse süntaksit, käske ei käivitata

`sh` ei pruugi toetada Bashi erisusi, näiteks massiive ja `[[ ... ]]` tingimust. **Süntaksikontroll ei ole ohutuskontroll**: korrektne skript võib siiski teha soovimatuid muudatusi. Ka hilisem süntaksiviga ei taga, et varasemad käsud pole jõudnud käivituda.

Argumendid, tingimus ja kordus

Enne terviknäidet selle ehitusklotsid:

Kirjapilt	Tähendus
<code>\$0</code>	skripti käivitamisel kasutatud nimi, mitte jooksva shelli nimi
<code>\$#</code>	argumentide arv
<code>\$1</code>	esimene argument
<code>"\$@"</code>	kõik argumendid eraldi, säilitades tühikud nende sees
<code>["\$#" -eq 0]</code>	kontroll: kas argumentide arv on null; sulgude ümber on tühikud
<code>if ...; then ...; fi</code>	tingimuse õnnestumisel käivitatav haru
<code>for nimi in "\$@"; do ...; done</code>	korda tegevust iga argumentiga
<code>>&2</code>	suuna teade veaväljundisse
<code>exit 1 / exit 0</code>	lõpeta veaga / edukalt

Loo fail

Tööta eraldi kataloogis. **Here-doc** `<<'EOF'` annab järgmised read käsu sisendiks kuni eraldi real oleva `EOF`-ini. Jutumärgid takistavad muutujate laiendamist

faili loomise ajal; need peavad laienema alles skripti käivitamisel. > kirjutab olemasoleva `tervita.sh` üle.

```
mkdir -p ~/tmp/skripti-naide
cd ~/tmp/skripti-naide
cat > tervita.sh <<'EOF'
#!/usr/bin/env bash
if [ "$#" -eq 0 ]; then
    printf 'Kasuta: %s nimi...\n' "$0" >&2
    exit 1
fi
for nimi in "$@"; do
    printf 'Tere, %s!\n' "$nimi"
done
exit 0
EOF
```

`printf` asendab `%s` järgmise argumendi tekstiga ning `\n` lõpetab rea.

Proovi edu ja viga

Kontrolli süntaksit. Vaikse eduka kontrolli järel anna omanikule käivitusõigus:

```
bash -n tervita.sh
chmod u+x tervita.sh
./tervita.sh Mari 'Jaan Tamm'
```

Tulemus on kaks tervitust, mitte kolm:

```
Tere, Mari!
Tere, Jaan Tamm!
```

Proovi ka puuduvat sisendit:

```
./tervita.sh
echo $?
```

Näed kasutusjuhust ja arvu 1. `$?` sisaldab **vahetult eelmise käsu** lõpetuskoodi: mõni vahele käivitatud käsk asendaks selle oma tulemusega.

Shebang töötab ka teiste keeltega: `#!/usr/bin/env python3` valib Pythoni, kuid faili sisu peab siis olema Pythoni kood. Selline fail on Pythoni skript, mitte shelliskript. Pikem võrdlus on kompileerimise ja käivitamise peatükis.

Minitest

1. Käivita skript ühe, kolme ja null argumendiga. Kontrolli iga kord lõpetuskoodi.
2. Anna nimeks `'Mari Maasikas'`. Miks on jutumärgid vajalikud nii käsureal kui skripti `"$@"` ümber?

3. Lisa skripti enne tsükli rida, mis kuvab argumentide arvu. Testi süntaksit enne käivitamist.
4. Muuda veajuhist nii, et kasutaja näeks üht korrektset käivitusnäidet. Veateade peab endiselt minema veaväljundisse.

Spikker

Eesmärk: esimese shelliskripti tuum on viis asja: shebang, käivitatavus, argumentid, 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-rida

Olulisemad lipud, märgid ja kiirnupud

- `#!/usr/bin/env bash` — vali Bash
- `$1` — esimene argument
- `"$@"` — kõik argumentid
- `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.

cron ja ajastatud tööd

Ajastatud töö käivitub kindlal ajal ilma sinu kohalolekuta. cron sobib näiteks perioodilise mõõtmise või varundusskripti käivitamiseks. Töö ei ava terminaliakent: salvesta väljund logisse.

Loe ajamustrit

Kasutaja cronitabeli reas on viis ajavälja ja seejärel käsk:

```
*/15 * * * * /bin/date >> "$HOME"/cron-naide.log 2>&1
```

Väli vasakult	Väärtused ja näite tähendus
minut	0–59; */15 tähendab siin minuteid 0, 15, 30 ja 45
tund	0–23; * tähendab iga tundi
kuu päev	1–31; * tähendab iga päeva
kuu	1–12; * tähendab iga kuud
nädalapäev	0–7, kus 0 ja 7 on pühapäev; * tähendab iga nädalapäeva

Seega ei tähenda näide „15 minutit pärast lisamist“, vaid järgmisi kellaaegu, mille minut jagub 15-ga. >> lisab väljundi logi lõppu ja 2>&1 ühendab sinna veaväljundi.

Kontrolli käsku enne ajastamist

```
/bin/date >> "$HOME"/cron-naide.log 2>&1
tail -n 5 "$HOME"/cron-naide.log
```

Logisse peab lisanduma praegune kuupäev ja kellaaeg. Croni PATH ja keskkond võivad erineda sinu terminali omadest; eelista programmi ning sisendfailide täis- teid. Oma virtuaalkeskonnas Pythonit saab kutsuda otse selle täisteega, ilma interaktiivse aktiveerimiseta.

Lisa üks töö seniseid säilitades

crontab -l näitab olemasolevaid töid, **crontab -e** avab need redaktoris. Kui töid pole, võib loendi asemel tulla vastav teade.

```
crontab -l
crontab -e
```

Lisa ülaltoodud näidisrida **seniste ridade lõppu**, salvesta ning kontrolli uuesti **crontab -l** abil. Kui tahad ainult mustrit õppida, ära rida päriselt paigalda.

crontab fail asendab kogu senise kasutaja cronitabeli, mitte ei lisa failist üht rida. Samuti eemaldab **crontab -r** kogu tabeli. Katse lõpetamiseks ava **crontab -e** ja eemalda ainult enda lisatud näidisrida.

Kui logisse ei teki uut rida

Kontrolli, kas määratud kellaaeg saabus, masin oli sisse lülitatud, croniteenus töötas ja kasutajal oli logifailile kirjutamisõigus. Tavapärane cron ei käivita magamise või väljalülitamise ajal vahele jäänud tööd automaatselt tagantjärele.

Linuxis kasutatakse ka **systemd** ajasteid, macOS-is **launchd**-d. Süsteemi aja- võond ja suvejale üleminek mõjutavad kellaaegadel põhinevaid ajastusi. Lisaks

käsitleb cron käsureas märki % eriliselt; näiteks `date +%F` vajab cronireas kuju `date +\%F`.

Minitest

1. Mida tähendavad ajad `0 8 * * *` ja `*/15 * * * *`?
2. Katseta logisse kirjutavat käsku käsitsi, ajastust lisamata.
3. Kuidas eemaldada ainult oma katsetöö, jättes teised alles?
4. Miks terminalis töötav käsk ei pruugi cronis sama tulemuse anda?

Spikker

Eesmärk: Ajasta läbiproovitud käsk; kasuta täisteid ja talleta väljund logisse.

Käsud ja näited

- `crontab -l` — vaata praeguse kasutaja ajastusi
- `crontab -e` — muuda tabelit, säilitades muud tööd
- `/bin/date >> "$HOME"/cron-naide.log 2>&1` — testi ajastatavat käsku käsitsi
- `tail -n 5 "$HOME"/cron-naide.log` — kontrolli, kas töö lisab kirjeid

Märgid ja valikud

- `*/15 * * * *` — minutitel 0, 15, 30 ja 45
- `0 8 * * 1-5` — tööpäeviti kell 08.00 süsteemi ajavööndis
- `crontab fail` — asendab kogu kasutaja ajastustabeli
- `crontab -r` — eemaldab kogu kasutaja ajastustabeli

Pane tähele: Ära kasuta `crontab fail` ühe rea lisamiseks. Cron ei käivita tavaliselt arvuti uneajal vahele jäänud töid tagantjärele.

Git, GitHub ja töövoog

Loogika

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

Git nimetab üht loogilist ajalookirjet sõnaga `commit`; selles peatükis kasutame selle kohta ka eestikeelset sõna **versioonikirje**.

Kõige tähtsam töövoog on:

1. vaata, mis seisus repo on
2. vaata, mis täpselt muutus
3. vali järgmise versioonikirje sisu
4. salvesta loogiline muutus versioonikirjena
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
<code>git status</code>	vaata repo seisu	muutunud, vahealal olevad ja jälgimata failid
<code>git diff</code>	vaata tööpuu muutusi	- ja + read jälgitud failides
<code>git add fail</code>	pane fail vahealale	edukal juhul sageli vaikne
<code>git diff --cached</code>	vaata vaheala sisu	järgmisse versioonikirjesse minevad muudatused
<code>git commit -m '...'</code>	loo versioonikirje	uue kirje lühikood ja kokkuvõte
<code>git log --oneline</code>	vaata ajalugu	versioonikirjed ühe rea kaupa
<code>git restore fail</code>	eemalda tööpuu muudatus	fail taastatakse vaheala seisu; salvestamata tööpuumuudatus kaob
<code>git restore --staged fail</code>	eemalda fail vahealalt	sisu jääb alles, vaheala muutub
<code>git switch -c haru</code>	loo ja ava uus haru	liigud uuele harule
<code>git pull --ff-only</code>	uuenda puhas haru serverist	edasinihe (fast-forward) või selge keeldumine
<code>git push -u origin haru</code>	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, vaheala ja versioonikirje
- 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

Kolm olekut: tööpuu, vaheala ja versioonikirje

Giti õppimine läheb palju lihtsamaks, kui eristad kolme kohta.

Koht	Tähendus	Kontrollkäsk
tööpuu	sinu failid praegu kettal	<code>git status</code> , <code>git diff</code>
vaheala	järgmise versioonikirje ettevalmistus	<code>git diff --cached</code>
versioonikirje (<code>commit</code>)	salvestatud loogiline muutus	<code>git log --oneline</code>

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, et muudatus läheb järgmisse versioonikirjesse.

Esimene harjutusrepo

Harjutamiseks vali uus tühi kataloog, mitte olemasolev repo. Näite `git init -b main` loob hoidla põhiharuga `main`; `-b` määrab algse haru nime.

```
mkdir -p ~/tmp/git-naide
cd ~/tmp/git-naide
git init -b main
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 versioonikirjesse.

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

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

Need seaded kehtivad selles repos; `--global` muudaks vaikeseadet kõigis sinu repodes. Nimi ja e-post salvestatakse versioonikirjetesse ning avaliku repo pu-

hul muutuvad avalikuks. See ei ole GitHubi sisselogimine. Seejärel proovi `git commit` uuesti.

Muudatuse ülevaatamine

Kui fail on juba Gitis 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 käsku `git commit` kontrolli ka vaheala:

```
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 tõmbekutse (`pull request`) avamist: 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 põhiharu nimi on `master`, kasuta nendes näidetes `main` asemel `master`.

Seejärel tee muudatused. Enne versioonikirje loomist:

```
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 põhiharu ainult siis, kui seda saab teha edasinihkena
- `git switch -c parandus` hoiab muudatuse eraldi harus
- `git diff` ja `git diff --cached` vähendavad juhusliku versioonikirje riski

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

Haru ehk branch

Haru (branch) on sama projekti eraldi arenguliin.

Tüüpiline mõtteviis:

- `main` on põhirida
- väike parandus tehakse eraldi harus
- hiljem ühendatakse valmis haru tõmbekutse (`pull request`) kaudu põhiharuga

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 algusega `https://...` kloonimisaadressi.

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 ning käskude `pull` või `rebase` kasutamist kontrolli:

```
git status
```

Järgmine käsk kustutab faili tööpuus tehtud, vahealale lisamata muudatused. `git restore fail.txt` taastab faili vahealalt, mitte tingimata viimasest versioonikirjest. Vaata enne `git diff` ja jäta vajalik töö alles:

```
git restore fail.txt
```

Kui tahad faili vahealalt eemaldada, kuid 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
```

Vaikimisi jätab `stash` jälgimata failid kõrvale; nende kaasamiseks on valik `-u`. `stash pop` võib tekitada konflikti ning jätab sel juhul kirje alles. `stash` on ajutine sahtel, mitte pikaajaline hoiukoht. Kui töö on sisuline, salvesta see pigem käsuga `git commit`, mitte ära jäta muudatusi nädalateks `stash`-i.

.gitignore

Kõiki faile ei tasu Giti lisada. Paroolid, privaativõtmed ja kohalikud saladused ei kuulu ka privaatsesse reposse. Kui saladus on juba avaldatud, tuleb see tühistada või vahetada; failist kustutamine ei eemalda seda ajaloost.

Tüüpiliselt jäetakse välja:

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

Lihtne näide:

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

Kui fail on juba Giti lisatud, siis ainult `.gitignore` ei eemalda seda 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
<code>merge</code>	säilitab mõlema haru ajaloo	valmis haru ühendamisel
<code>rebase</code>	tõstab sinu versioonikirjed teise haru lõppu	oma parandusharu korrastamisel

Näide: oled harus `parandus` ja tahad sellesse ühendada värske haru `main`:

```
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 parandusharus võib enne tõmbekutse esitamist sageli kasutada käsku `rebase`

- ära kasuta käsku **rebase** harus, millele teised juba oma tööd rajavad, 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

Konfliktimärgid algavad päris failis rea algusest. Ülal on ühendamise (**merge**) näide. Ümbertöstmise (**rebase**) ajal võib HEAD tähistada hoopis alust, mille peale sinu versioonikirjet uuesti rakendatakse: loe mõlemat varianti, mitte ära vali neid sildi „minu“ järgi.

Lahendamine:

1. ava fail redaktoris
2. tee valmis õige lõpptekst
3. kustuta konfliktimärgid
4. kontrolli faili sisu
5. lisa fail vahealale

Näiteks:

```
git status
nano fail.txt
git add fail.txt
```

Kui konflikt tekkis käsu **git merge** ajal:

```
git commit
```

Kui konflikt tekkis käsu **git rebase** 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 tõesti vajalik, peaks see olema meeskonnas kokku lepitud ja piirduma tavaliselt oma parandusharuga, mitte ühise haruga `main`.

GitHub: issue ja pull request

GitHubis seotakse muudatus tavaliselt arutelukoha või ülesandega.

Põhimõisted:

- **issue** ehk probleemikirje on ülesanne, viga või aruteluteema
- **pull request** ehk tõmbekutse esitab muudatusettepaneku ülevaatamiseks ja ühendamiseks
- **review** ehk koodiülevaatus on teiste tagasiside tõmbekutse kohta
- **checklist** ehk kontroll-loend on Markdowni märkeruutudega loend

Lihtne probleemikirje 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
- [] koostamine õnnestub

Hea tõmbekutse ehk muudatusettepanek vastab kolmele küsimusele:

- mis probleem lahendati

- kuidas seda lahendati
- kuidas kontrolliti, et lahendus töötab

Tüüpiline GitHubi töövoog:

1. vali probleemikirje või kirjuta probleem lühidalt lahti
2. loo selle jaoks haru, näiteks `fix-cat-less`
3. tee väike loogiline muudatus
4. loo selge sõnumiga versioonikirje
5. saada haru käsuga `git push` GitHubi
6. ava tõmbekutse (`pull request`)
7. vasta ülevaatuskommentaaridele uute versioonikirjetega
8. pärast ühendamist kustuta tööharu

Versioonikirje või tõmbekutse kirjelduses saab probleemikirjele viidata:

```
Fixes #12
Closes #12
Refs #12
```

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.

Spikker

Git-käskude visuaalne kaart ja pikem tekstiline meelepea on lisas Lisa B: spikrite register.

Minitest

1. Selgita, mis vahe on tööpuul, vahealal ja `commit`-kirjel ehk versioonikirjel.
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 tõmbekutse (`pull request`) lühikirjeldus kujul: probleem, lahendus, kontroll.

Lisalugemine

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

Spikker

Eesmärk: Git hoiab muutuste ajalugu; erista tööpuud, vaheala ja versioonikirjet ning vaata muudatused enne üle.

Põhikujud

- `git status` — vaata seis
- `git diff` — muutused tööpuus
- `git diff --cached` — muutused vahealal
- `git add fail.txt` — lisa vahealale
- `git commit -m '...'` — loo versioonikirje
- `git switch -c parandus` — loo haru
- `git pull --ff-only` — uuenda puhtalt
- `git push -u origin parandus` — saada haru
- `git restore --staged fail.txt` — eemalda vahealalt

Olulisemad lipud, märgid ja kiirnupud

- `HEAD` — praegune tipp
- `main` — põhirida
- `origin` — kaugrepo
- `vaheala` — järgmise versioonikirje sisu

Pane tähele: Uus jälgimata fail ei ilmu tavalisse `git diff` vaatesse; pärast `git add` kontrolli esimese versioonikirje sisu käsuga `git diff --cached`.

Pythoni venv ja eraldatud keskkonnad

`venv` hoiab projekti Pythoni paketid teistest projektidest eraldi. Kui üks projekt vajab paketi üht ja teine teist versiooni, ei pea need sama paigaldust jagama. Tavaliselt on igal projektil oma `.venv` kataloog.

Loo ja aktiveeri

Vaja on töötavat `python3` käsku. `python3 -m venv` käivitab selle Pythoni mooduli `venv`; viimane argument on loodava keskkonna kataloog.

```
mkdir -p ~/tmp/python-naide
cd ~/tmp/python-naide
python3 -m venv .venv
source .venv/bin/activate
```

Loomine on tavaliselt vaikne. **Aktiveerimine muudab praeguse shelli PATH-i**, nii et **python** viitab nüüd projekti keskkonnale. Viipa võib ilmuda (**.venv**), kuid kontrolli tegelikku tõlgendajat:

```
python -c 'import sys; print(sys.executable)'
python -m pip --version
```

-c käivitab jutumärkides antud Pythoni koodi. Mõlemas väljundis peaks tee viitama projekti **.venv**-ile. See ei muuda teise avatud terminali ega IDE tõlgendajat automaatselt.

Paigalda ja katseta paketti

python -m pip kasutab just selle Pythoni paketihooldurit. Järgmine käsk laeb paketi ja selle sõltuvused internetist:

```
python -m pip install requests
python -c 'import requests; print(requests.__version__)'
```

Teine käsk kinnitab, et pakett on imporditav, ja näitab versiooni. **python -m pip list** kuvab paigaldatud paketid. **Ära kasuta virtuaalkeskkonnas sudo pip-i**.

Salvesta sõltuvused, mitte keskkonnakataloog

pip freeze väljastab paigaldatud pakettide versioonid. Uues katseprojekti saad need salvestada:

```
python -m pip freeze > requirements.txt
cat requirements.txt
```

> kirjutab faili üle: olemasolevas projektis ära asenda sõltuvusfaili enne selle otstarbe kontrollimist. **freeze** sisaldab ka kaudseid sõltuvusi; see pole garantii, et keskkond töötab muutmata igas operatsioonisüsteemis või iga Pythoni versiooniga.

Teine inimene loob oma **.venv** ja paigaldab paketid kujul **python -m pip install -r requirements.txt**. -r tähendab, et paketinimed loetakse failist. Giti läheb sõltuvusfail, mitte **.venv/**; ära eelda, et keskkonnakataloogi kopeerimine teise masinasse töötab.

Väljumine ja järgmine tööpäev

deactivate taastab shelli varasema keskkonna. See ei kustuta pakette ega projekti. Järgmisel korral liigu projekti ja aktiveeri olemasolev keskkond; uut **venv-i**

ei ole iga kord vaja luua.

Aktiveerimine pole kohustuslik: `.venv/bin/python skript.py` valib sama tõlgendaja otse. See on kasulik ajastatud töödes ja IDE seadistuses.

Piirid ja levinud vead

- **No module named venv või ensurepip puudub:** Linuxis jaotus võib vajada eraldi `python3-venv` paketti. Vaata oma paketihooldust, mitte ära asenda süsteemi Pythonit juhuslikult.
- **externally-managed-environment:** paigaldad tõenäoliselt süsteemi Pythonisse. Loo või vali virtuaalkeskkond; ära lülita süsteemi kaitset välja.
- **IDE ei leia paketti:** vali IDE-s projekti `.venv/bin/python` ja võrdle selle teed terminali omaga.
- **Pakett vajab süsteemiteeki:** `venv` ei paigalda uut operatsioonisüsteemi ega isoleeri süsteemiteeki. Selline sõltuvus võib vajada paketihooldurit või Dockerit.
- **Võõras kood:** virtuaalkeskkond ei ole turvaliivakast. Programm saab endiselt lugeda sinu faile ja kasutada võrku sinu õigustega.

Minitest

1. Kontrolli Pythoni täisteed enne aktiveerimist ja pärast seda.
2. Paigalda katsepakett ning tõesta impordiga, et kasutad õiget keskkonda.
3. Lõpeta aktiveerimine. Kas saad sama paketti kasutada käsuga `.venv/bin/python`?
4. Miks jagatakse `requirements.txt`, mitte kogu `.venv` kataloogi?

Lisalugemine

Pythoni `venv`-juhend ja pakettide paigaldamise juhend selgitavad loomist, aktiveerimist ja piiranguid.

Spikker

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.

Docker'i alused

Loogika

Docker aitab panna rakenduse koos käivituskeskonnaga ühte korratavasse pakki.

Kõige tähtsam mõttejärjekord on:

1. vali või koosta konteinerkuju (`image`)
2. käivita sellest **konteiner**
3. seo vajadusel port või kataloog hostsüsteemiga
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 asub lähtekood hostsüsteemi Giti hoidlas ning Docker annab koodile ühtse käivituskeskonna.

Kiire orientiir

Mõiste või käsk	Milleks	Mida tavaliselt näed
konteinerkuju (<code>image</code>)	käivitusmall	nimi ja märgend, näiteks <code>python:3.13-slim</code>
konteiner	konteinerkujust käivitatud protsess	töötav või peatatud eksemplar
Dockerfile	konteinerkuju koostamise retsept	<code>FROM</code> , <code>WORKDIR</code> , <code>COPY</code> , <code>RUN</code> , <code>CMD</code>
<code>docker build</code>	koosta konteinerkuju	koostamiskihid ja kuju nimi
<code>docker run</code>	käivita konteiner	programmi väljund või konteineri ID
<code>docker ps</code>	vaata töötavaid konteinereid	konteinerite tabel
<code>docker logs</code>	loe konteineri väljundit	programmi logiread
<code>docker exec</code>	käivita käsk töötavas konteineris	käsu väljund konteineri seest

Mõiste või käsk	Milleks	Mida tavaliselt näed
<code>docker compose</code>	käivita seotud teenused	teenuste logid ja seis

Tüüpilised algaja vead

- aetakse segi konteinerkuju (`image`) ja konteiner
- arvatakse, et konteineri sees käsitsi muudetud failid jäävad alati alles
- kasutatakse `docker exec` konteineri käivitamiseks, kuigi see töötab ainult juba töötavas konteineris
- eeldatakse, et `depends_on` tähendab andmebaasi täielikku valmisolekut
- pannakse konteinerisse `venv` ainult harjumusest, kuigi konteiner ise juba isoleerib keskkonda
- kustutatakse käsuga `docker compose down -v` kogemata andmebaasi andmekõide

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 katsetamiseks mõeldud konteinerkuju. Esimesel käivitamisel võib Docker selle internetist alla laadida.

macOS-is ja Windowsis tähendab Docker tavaliselt Docker Desktopi või muud taustal töötavat virtuaalmasinakihti. Kui käsk ütleb, et Dockeri daemon ei tööta, ava Docker Desktop ja proovi uuesti.

Konteinerkuju (`image`) ja konteiner

Konteinerkuju (`image`) on valmis käivitusmall. Konteiner on sellest mallist käivitatud konkreetne eksemplar.

Ühest konteinerkujust võib korraga töötada mitu konteinerit:

```
docker run --rm alpine echo tere
docker run --rm alpine uname -a
```

Siin:

- `alpine` on konteinerkuju
- `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 hostsüsteemi kataloogis, nimega andmeköites (`named volume`) või Giti hoidlas.

venv vs Docker

`venv` ja Docker lahendavad eri probleemi.

Olukord	Tavaliselt sobib
ainult Pythoni paketid peavad olema eraldi	<code>venv</code>
vaja on kindlat Linuxi keskkonda või süsteemipakette	Docker
projekt vajab andmebaasi, vahemälu või mitut teenust	Docker Compose
tahad kiiresti väikest kohalikku skripti teha	<code>venv</code>
meeskonnal peab sama rakendus eri masinates sarnaselt käivituma	Docker

Dockeris sees ei ole eraldi `venv` enamasti vajalik. Kui üks konteiner käitab üht rakendust, on konteiner ise eralduskiht.

Esimene väike Dockerfile

Kõige selgem esimene näide on väike programm, mille konteinerkuju 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 töökataloogi konteineri sees
- `COPY app.py .` kopeerib faili konteinerkujusse
- `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 tähistab koostamiskonteksti (`build context`): kataloogi, mille sisu Docker koostamise ajal näeb.

.dockerignore

Koostamiskontekst ei peaks sisaldama kõike, mis projektikataloogis on.

Tüüpiline `.dockerignore`:

```
.git  
.venv  
.env  
.env.*  
__pycache__/  
dist/  
*.log
```

See kiirendab koostamist ja vähendab ohtu, et konteinerkujusse 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 töötab server, tuleb port hostsüsteemiga siduda.

```
docker run --rm -d --name pyweb -p 127.0.0.1:8000:8000 python:3.13-slim python -m http.server
```

Siin:

- `-d` paneb konteineri taustale
- `--name pyweb` annab konteinerile nime
- `-p 127.0.0.1:8000:8000` seob ainult kohaliku arvuti pordi 8000 konteineri pordiga 8000; ilma `127.0.0.1`-ta võib port olla avatud ka võrgust

Kontroll:

```
docker ps  
docker logs pyweb
```

Brauseris peaks teenus olema aadressil:

`http://localhost:8000`

Kui port on juba kasutusel, vali vasakul pool teine hosti port, näiteks `-p 127.0.0.1:8080:8000`.

logs ja exec

Kui konteiner on juba käimas, alusta veaotsingut logidest.

```
docker logs pyweb
docker logs -f pyweb
```

`-f` jälgib logi jooksvalt. Katkesta jälgimine `Ctrl-C`-ga enne järgmist käsku; see ei peata konteinerit.

Käsk konteineri sees:

```
docker exec pyweb pwd
docker exec -it pyweb sh
```

`docker exec` ei loo uut konteinerit. See käivitab käsu olemasolevas töötavas konteineris.

Interaktiivsest `sh`-st välju käsuga `exit`. Kui brauserikatse, logid ja `exec` on proovitud, peata hosti terminalis konteiner käsuga `docker stop pyweb`. Valiku `--rm` tõttu eemaldatakse see siis ka nimekirjast. Väikeses konteinerkujus ei pruugi `bash`-i olla; `sh` on sagedamini olemas.

Hostitee sidumine konteinerisse

Koostatud konteinerkuju sobib valmis rakenduse proovimiseks. Arenduses tahad sageli, et konteiner näeks kohe sinu praegust projektikataloogi.

```
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 kataloog hostsüsteemis
- `dst=/app` või `/app` on sama kataloog konteineri sees
- `-w /app` teeb sellest töökataloogi

Konteineri sees käivita varem loodud `app.py`, seejärel välju. Pakettide paigaldamine ja testimine vajavad projekti sõltuvusfaili ning teste; neid esimene näide veel ei sisalda:

```
python app.py
exit
```

Hostitee sidumise (`bind mount`) korral muutub hostsüsteemi fail kohe. See on arenduses mugav, kuid tähendab ka, et konteineris tehtud muudatus võib muuta sinu töökataloogi.

Konteineri peatamine üksi ei kustuta selle kirjutatavat kihti. Andmed kaovad konteineri eemaldamisel, näiteks `--rm` või `docker compose down` korral, kui need pole seotud hostitee või andmeköitega.

Nimega andmeköide

Hostitee sidumine ühendab konteineri hostsüsteemi kindla faili või kataloogiga. Nimega andmeköide (`named volume`) on Dockeri hallatav püsiv andmeala.

Kõige tavalisem kasutus on andmebaasi andmed:

```
volumes:
  pgdata:
```

Järgmine on ainult teenusekirjelduse **katkend**, mitte iseseisev käivitata fail. Täielik fail koos parooli ja teenustega tuleb allpool:

```
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 nimega andmeköited
- andmebaasi andmeköite kustutamine tähendab sageli andmete kustutamist

Arenduse tööjaotus

Hea algaja tööjaotus on:

Kus	Mida tee
hostsüsteemis	muuda faile, salvesta muutatusi
konteineris	Gitiga, hoia dokumentatsiooni käivita rakendus, paigalda sõltuvusi, testi, silu keskkonda
andmeköites (<code>volume</code>)	hoia andmeid, mis peavad konteineri vahetamisel alles jääma

Välldi 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`. Järgmine variant näitab paigalduse asukohta Dockerfile'is. Versioonita `requests` valib paigaldushetkel sobiva versiooni; korduvaks koostamiseks salvesta oma läbiproovitud versioonid, nagu `venv`-peatükis.

`requirements.txt`

`requests`

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 koostamisvahemällu
- `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:
      - "127.0.0.1:8000:8000"

  db:
    image: postgres:16
    environment:
      POSTGRES_DB: opik
      POSTGRES_USER: opik
      POSTGRES_PASSWORD: opikpass
    volumes:
      - pgdata:/var/lib/postgresql/data
```

```
volumes:
  pgdata:
```

See kohalik katse käivitab kaks teenust, kuid HTTP-failiserver **ei kasuta veel andmebaasi**. Parool **opikpass** on ainult harjutuseks; andmebaasi porti ei avaldata hostile. Failiserver jagab praegust katsekataloogi, seega ära kasuta saladusi sisaldavat projekti.

Käivita teenused taustal:

```
docker compose up -d
docker compose logs db
```

Kui logi järgi võtab andmebaas ühendusi vastu, kontrolli seda. `psql` on PostgreSQL-i klient, `-U` määrab kasutaja, `-d` andmebaasi ja `-c` täidetava SQL-käsu:

```
docker compose exec db psql -U opik -d opik -c 'SELECT 1;'
```

Tulemuses näed väärtust 1. Ava HTTP-teenus brauseris ja lõpeta katse:

```
docker compose down
```

Logi saab pikemalt jälgida käsuga `docker compose logs -f app`; enne järgmisi käske katkestada jälgimine `Ctrl-C`-ga. Interaktiivsest `docker compose exec app sh` shellist välju `exit`-iga.

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 korduskatsete loogikat
- andmebaasi tervisekontrolli (**healthcheck**)
- käivituskripti, 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 konteinerkujus	docker run --rm alpine uname -a
kontrolli valmis rakenduse konteinerkuju	docker build ja docker run
arenda koodi konteineri sees, kuid failid jäävad hostsüsteemi	hostitee sidumine
käivita rakendus koos andmebaasi või muu teenusega	docker compose
IDE avab kogu tööruumi konteineris	arenduskonteiner ehk devcontainer

See valik vähendab segadust: ära kasuta Compose'i ainult ühe lühikese käsu käivitamiseks ega kirjuta üht väga pikka **docker run** käsku, kui projektis on mitu teenust.

Ohutus ja privaatsus

Konteiner ei ole maagiline turvasein.

Ole ettevaatlik, kui:

- konteinerkuju on tundmatust allikast
- seod konteineriga hostsüsteemi katalooge
- paned keskkonnamuutujatesse paroole või võtmeid
- näed juhendit, mis seob konteineriga Dockeri sokli, näiteks **/var/run/docker.sock**
- käsk kasutab **--privileged**

Algaja hea reegel: ära käivita võõrast konteinerkuju nii, et seod sellega oma kodukataloogi või salajased failid.

Kettaruumi kontroll ja koristamine

Docker võib aja jooksul koguda konteinerkujusid, peatatud konteinereid ja koostamisvahemälu.

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 konteineriga
- projektikataloog seotakse tööruumina
- terminal, testid ja keele tööriistad töötavad konteineris

Arenduskonteiner ei asenda Dockeri põhimõtteid. Ta teeb sama loogika mugavamaks, eriti siis, kui projekt vajab keerulisemat keskkonda.

Minitest

1. Mis vahe on konteinerkujul (`image`) ja konteineril?
2. Miks on `--rm` kasulik lühikeste katsete puhul?
3. Mida teeb `-p 127.0.0.1:8000:8000`?
4. Mis vahe on hostitee sidumisel ja nimega andmeköitel (`named volume`)?
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.

Spikker

Eesmärk: Konteinerkuju (`image`) on käivitusmall; konteiner võib töötada või olla peatatud. Konteineri eemaldamisel kaob selle kirjutatav kiht, mitte seotud hostifailid ega püsivad andmeköited.

Põhikäsud

- `docker --version` — kontrolli olemasolu
- `docker run --rm alpine echo tere` — ohutu esimene käivitus
- `docker build -t rakendus .` — koosta konteinerkuju
- `docker run --rm rakendus` — käivita konteiner

- `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 ja eemalda konteinerid ning võrk

Olulisemad lipud, märgid ja kiirnupud

- `image` — konteinerkuju ehk käivitusmall
- `konteiner` — käivitatud või peatatud eksemplar
- `Dockerfile` — konteinerkuju retsept
- `compose.yaml` — mitu teenust
- `volume` — püsiv andmeköide
- `registry` — konteinerkujude hoidla
- `--rm` — kustuta lühikatske konteiner pärast lõppu
- `-p 127.0.0.1:8000:8000` — `docker run`: avalda port ainult oma arvutile
- `--mount type=bind,src="$PWD",dst=/app` — jaga praegune kataloog konteinerisse
- `-w /app` — määra konteineri töökataloog
- `docker compose down -v` — peatab ja kustutab ka andmekööited

Pane tähele: Kui konteiner ei tööta, vaata enne `docker logs`; ära seo tundmatu konteinerkujuga oma kodukataloogi ega salajasi faile.

IDE-d ja arenduskeskkonnad

IDE (*Integrated Development Environment*) ehk integreeritud arenduskeskkond ühendab redaktori, käivitamise ja silumise. Kontrolli, et selle tööriistad kasutaksid sama projekti ja keelekeskkonda.

Kiirülevaade

Eesmärk on panna terminal, Git, tõlgendaja ja vajadusel konteinerid töötama sama projektiga.

Kontroll või valik	Milleks	Mida tavaliselt näed
<code>--version</code> käsud	kontrolli tööriista olemasolu	üks lühike versioonirida
IDE tõlgendaja	vali projekti Python, Node vms	valitud keskkond IDE seadetes
sisseehitatud terminal	käivita kāske projekti kontekstis	sama töökataloog ja projektifailid
devcontainer või Docker	anna projektile eraldi käivituskeskkond	konteineri terminal ja tööriistad

Kontroll või valik	Milleks	Mida tavaliselt näed
puuduv tööriist	vajab paigaldust või PATH-i parandust	veateade <code>command not found</code> või selle vaste

Tüüpilised algaja vead

- arvatakse, et IDE „teab ise“, millist Pythoni või Node'i kasutada
- terminal ja IDE kasutavad eri keskkondi
- proovitakse korraga liiga palju tööriistu, kuigi alguses piisab lihtsast ter-
vikust

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 ja hea Remote SSH toega
- PyCharm: tugev Pythoni tugi
- Vim või Neovim: kiire klaviatuuripõhine töö
- JetBrainsi tööriistad üldisemalt: tugev projektitugi

Mida IDE-s jälgida

- sisseehitatud terminal
- Giti lõiming
- 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, tõlgendaja ja tööriistad töötavad konteineris

See aitab eriti siis, kui:

- projekt vajab kindlat Linuxi keskkonda
- mitmel arendajal peab olema sama töölaud
- hostsüsteemi ei taheta liigsete tööriistadega täita

Lühidalt:

- `venv` aitab Pythoni pakettidega
- Docker aitab kogu keskkonnaga
- arenduskonteiner ühendab selle IDE kasutusmugavusega

Minimaalne `devcontainer.json`

Näide eeldab töötavat Dockerit, VS Code'i laiendust Dev Containers ning projekti juures olemasolevat `requirements.txt` faili. `postCreateCommand` käivitatakse konteineri loomisel; see on päris käsk, mitte pelgalt kirjeldus.

`.devcontainer/devcontainer.json`

```
{
  "name": "Python Dev Container",
  "image": "python:3.13-slim",
  "workspaceFolder": "/workspace",
  "workspaceMount": "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 konteinerkuju (`image`)
- kohalik projekt seotakse kataloogiga `/workspace`
- pärast loomist paigaldatakse sõltuvused
- VS Code saab automaatselt vajalikud laiendused

Tüüpiline töövoog arenduskonteineriga

1. paigalda VS Code'is laiendus Dev Containers
2. ava projektikataloog
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 töötab Python või Node
- kus tehakse `pip install` või `npm install`
- kas avatud terminal on hostis või konteineris

Kontrolli vajalikke tööriistu

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 tõlgendajaks selle keskkonna Python
3. paigalda sõltuvused sinna, mitte süsteemi suvalisse asukohta

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`

`npm install` paigaldab sõltuvused; `npm run dev` ja `npm test` töötavad ainult siis, kui projekt on need skriptid määranud. Vaata enne `package.json` sisu: paigaldus võib käivitada projekti või pakettide skripte.

Tüüpilised käsud:

```
npm install
npm run dev
npm test
```

Seadistuse ametlik kirjeldus: VS Code: töökausta sidumine konteineriga.

Minitest

1. Pane kirja, millist redaktorit või IDE-d sa kasutad.
2. Kontrolli, kas IDE terminal kasutab sama shelli nagu tavaline terminal.
3. Uuri, kuidas selles IDE-s Giti versioonikirjet (`commit`) luua.
4. Kontrolli, kuidas valida IDE-s Pythoni tõlgendaja või Node.js-i projekt.
5. Kui kasutad VS Code'i, uuri, kus menüüs on **Reopen in Container**.

Spikker

Eesmärk: heas arenduskeskkonnas osutavad terminal, Git, tõlgendaja ja vajaduse korral konteiner samale projektile.

Kontrollid ja valikud

- `python3 --version` — kontrolli Pythonit
- `python3 -m pip --version` — kontrolli pip-i
- `git --version` — kontrolli Giti
- `docker --version` — kontrolli Dockerit
- `node --version` — kontrolli Node'i
- `npm --version` — kontrolli npm-i

Olulised mõisted

- `tõlgendaja` — koodi 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 kui terminal.

Andmeteaduse eelteadmised käsurea vaates

Loogika

Kui räägitakse andmeteaduse eelteadmistest, siis tavaliselt mõeldakse vähemalt neid plokkide:

- programmeerimine, eriti Python
- andmebaasid, SQL ja relatsiooniline mõtteviis
- failivormingud nagu CSV, JSON ja XML
- statistika, tõenäosusteooria ja matemaatiline mõtlemine

Käsuri ei asenda neid oskusi, kuid aitab neid omavahel siduda.

Just siin on käsurea suur väärtus:

- näed kiiresti, mis failid sul üldse on

- saad kontrollida andmete kuju enne suurema keskkonna avamist
- 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	skriptid ja korduvad käivitused
SQL ja andmebaasid	andmetöötlussamme 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 üks ei asenda
head, column, jq, wc, sqlite3	tee esimene kontroll	faili kuju, read, veerud ja mahud
Python, SQL, notebook-keskkond	tee põhjalikum 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 töövoogu korrastatuse tähtsust

See õpik katab tugevamalt:

- käsurea loogika
- failide ja voogude töötlust
- Pythoni keskkonnad
- SQLite'i ja SQL-i alguse
- arendustöövoogu, Giti ja Dockerit

See õpik ei püüa eraldi õpetada põhjalikult:

- statistikat
- tõenäosusteooriat
- lineaaralgebrat
- R-i

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

Sageli ei ole probleem 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 „käsku andmebaasile“, vaid ka kindlat 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.

Tööjaotus on järgmine:

- käsurida aitab andmeid ette valmistada
- Python või R aitab neid töödelda
- statistiline mõtlemine aitab tulemusi mõista

Kõik kolm on eri asjad.

Näited

Tüüpiline väike andmetöö võib kulgeda nii:

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

Vastav töövoog võib olla:

`fail -> head/less/grep -> column/cut/tr -> sqlite3 -> python3`

Seetõttu on käsurida andmeteaduse alguses kasulik:

- see aitab väikeste sammudega kiiresti ülevaate saada
- see ei sunni kohe suurt keskkonda avama
- see muudab 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.

Spikker

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 märkmiku (notebook) juurde enne, kui oled vähemalt korra kontrollinud, mis kujul andmed sul tegelikult on.

CSV, JSON ja XML käsurreal

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 edasi SQLite'i või Pythoni juurde

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
<code>head</code>	vaata faili algust	esimesed read
<code>column</code>	tee lihtne tabel loetavamaks	joondatud veerud
<code>python3 -m json.tool</code> , <code>jq</code>	vaata JSON-i	vormindatud või filtreeritud JSON
<code>xmllint</code>	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 koosnevad lihtsatest ühetaolistest võtmeväärtuse paaridest

Kõige tavalisemad tööriistad

CSV

Lihtsa CSV puhul on väga kasulikud:

- `head`
- `column`
- `cut`
- `sort`
- `wc`

Tähtis hoiatus:

- CSV võib sisaldada jutumärke, väljade sees komasid ja reavahetusi
- siis ei pruugi `cut -d`, olla enam piisavalt täpne

Seega:

- kiireks vaatamiseks on shell väga hea
- keerulise CSV jaoks on parem kasutada Pythonit või selleks mõeldud 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, kuid see 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` võimaldab kiiresti märgendeid otsida
- `xmllint --format` teeb struktuuri palju loetavamaks

Kui `xmllint` puudub, ei ole see üllatav: tööriist ei ole igas süsteemis vaikinisi olemas.

CSV kirje ei pruugi olla üks tekstirida. Jutumärkides välja sees võib olla reavahetus; `wc -l` loendab reavahetusi, mitte CSV kirjeid. Selliste andmete jaoks kasuta CSV-parserit, mitte `cut -d,.`

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 keeruka CSV-faili töötlemiseks piisata.

Lisalugemine

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

Spikker

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 filtriiks

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.

Andmebaasi algus: SQLite ja Python

SQLite salvestab andmebaasi ühte faili ega vaja eraldi serverit. Kasutame seda küsimuse lahendamiseks: kuidas siduda õppijad nende tulemustega ja arvutada keskmine, kordamata igal real õppija põhiandmeid?

Mõiste	Tähendus selles näites
tabel	ühe teema kirjed, näiteks õppijad
rida / veerg	üks õppija / üks omadus, näiteks nimi
primaarvõti (<i>primary key</i>)	rea kordumatu tunnus <code>id</code>
võõrvõti (<i>foreign key</i>)	tulemuserea viide õppija tunnusele
JOIN	seo tabelite read võtme järgi
GROUP BY	koonda read ühise tunnuse järgi

Turvaline harjutuskoht ja tööriist

Vaja on käsku `sqlite3`. Pythoni moodul `sqlite3` võib olla olemas ka siis, kui eraldi käsureaprogramm puudub. Vajadusel vaata paketi haldust.

mktemp -d loob uue kordumatu nimega kataloogi ja väljastab selle tee. Käsuasendus `$(...)` salvestab tee muutujasse, nii et näide ei kirjuta varasemat andmebaasi üle:

```
katse=$(mktemp -d)
cd "$katse"
pwd
```

Jäta tee meelde. Ajutises kataloogis olevat tööd võib süsteem hiljem koristada; see on katse, mitte püsivate andmete asukoht.

`sqlite3 andmed.db` avab interaktiivse SQL-käsurea; väljumiseks kirjuta `.quit`. Punktiga käsud, näiteks `.tables` ja `.schema`, on SQLite'i kliendi käsud, mitte SQL.

Loo õppijate tabel

`CREATE TABLE` loob tabeli, `INSERT INTO` lisab kirjed ja `SELECT` loeb neid. `INTEGER` on täisarv, `TEXT` tekst ning `NOT NULL` keelab puuduva väärtuse. SQL-laused lõpevad semikooloniga.

Shelli `<<'SQL'` annab käsule mitmerealise sisendi kuni eraldi real oleva SQL-ini. Valikud `-header` `-column` kuvavad tulemuse veerupealkirjadega tabelina.

```
sqlite3 -header -column andmed.db <<'SQL'
CREATE TABLE students (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  city TEXT
);
INSERT INTO students VALUES
  (1, 'Mari', 'Tartu'),
  (2, 'Jaan', 'Tallinn'),
  (3, 'Liis', 'Narva');
SELECT * FROM students;
SQL
```

Näed kolme õppijat. Tabeli loomine ja lisamine ise tavaliselt teksti ei väljasta. Ära käivita loomise plokki samas andmebaasis uuesti: tabel on juba olemas. Uueks katseks loo uus ajutine kataloog, mitte ära kustuta seniseid tabeleid.

Struktuuri saad hiljem kontrollida shellist käsuga `sqlite3 andmed.db '.schema students'`.

Lisa tulemused eraldi tabelisse

Ühel õppijal võib olla mitu tulemust. `results.student_id` viitab tabeli `students.id` väärtusele. `PRAGMA foreign_keys = ON` lülitab viidete kontrolli sisse; SQLite'is tuleb seda teha igas andmeid muutvas ühenduses, kui kontroll pole juba lubatud.

```
sqlite3 andmed.db <<'SQL'
PRAGMA foreign_keys = ON;
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 results (student_id, subject, score) VALUES
    (1, 'matemaatika', 91), (1, 'python', 95),
    (2, 'matemaatika', 84), (2, 'python', 79),
    (3, 'matemaatika', 88), (3, 'python', 90);
SQL

```

Viidete kontroll takistab tulemuse lisamist olematule õppijale. See ei kontrolli automaatselt näiteks punktisumma mõistlikku vahemikku; sellised reeglid tuleb eraldi määrata.

Seo ja koonda

Järgmises päringus on **s** ja **r** tabelite lühinimed. **ON** määrab seose ja **ORDER BY** tulemuse järjekorra:

```

sqlite3 -header -column 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;
"

```

Näed kuut tulemust koos õppijanimedega. See **JOIN** jätab välja õppijad, kellel tulemust pole; kõigi õppijate säilitamiseks oleks vaja **LEFT JOIN**-i.

Keskmiseks rühmita õppija järgi. **AVG** arvutab keskmise, **ROUND(..., 1)** ümardab ühe kümnendkohani, **AS** annab tulemuse veerule nime ja **DESC** järjestab kahanevalt:

```

sqlite3 -header -column andmed.db "
SELECT s.name, ROUND(AVG(r.score), 1) AS keskmine
FROM results r
JOIN students s ON s.id = r.student_id
GROUP BY s.id, s.name
ORDER BY keskmine DESC;
"

```

Tulemused on Mari 93.0, Liis 89.0 ja Jaan 81.5. Päring ei muuda salvestatud hindeid.

Sama andmebaas Pythonist

Salvesta järgmine kood faili **naide.py** samas kataloogis. Käivita **python3 naide.py**. Päringu ? on **parameetri kohatäitja**: kasutaja väärtusi ei liideta SQL-i stringi sisse.

```
import sqlite3

conn = sqlite3.connect("andmed.db")
try:
    for name, in conn.execute(
        "SELECT name FROM students WHERE city = ?", ("Tartu",)
    ):
        print(name)
finally:
    conn.close()
```

Näed Mari. SQL valib andmed; Python saab tulemuse edasi töödelda. `finally` sulgeb ühenduse ka vea korral.

Minitest

1. Leia kõigi õppijate asemel ainult Tallinna õppija nimi.
2. Miks ei tohiks seost teha nime järgi, kui kahel õppijal võib olla sama nimi?
3. Mis muutuks keskmise päringus, kui eemaldada `GROUP BY`?
4. Miks ei piisa SQLite'is ainult võõrvõtme kirjutamisest tabeli kirjeldusse?
5. Muuda Pythoni näites linna parameetrit, jättes SQL-teksti samaks.

Lisalugemine

SQLite'i võõrvõtmete juhend selgitab viidete kontrolli. Üldised viited on andmevormingute ja SQLite'i lisalugemises.

Spikker

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 ridade
- `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.

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 natiivne binaar
- mõni tulemus vajab eraldi käivituskeskkonda või virtuaalmasinat

See teema seob kokku:

- shebang-rida ja käivitavad skriptid
- **PATH**-i ja käskude leidmise
- Dockeri ja koostamistööriistad
- LaTeX-i kompileerimise idee

Kiirülevaade

Eesmärk on mõista, miks eri keelte programmide käivitamine erineb: mõnda tõlgendatakse käivitamise ajal, mõni koostatakse enne eraldi artefaktiks.

Kuju või tööriist	Milleks	Mida tavaliselt näed
tõlgendiga skript kompilaator	käivita lähtekood otse tee lähtekoodist uus artefakt	programmi väljund kohe binaar või koostamiskataloog
<code>go build</code> , <code>cargo</code>	juhi koostamissamme tööriistaga	koostamislogi ja väljundfailid
Java	kompileeri ja käivita eri sammudena	<code>.class</code> 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 koostamis- ja käivituskäsk
- unustatakse vaadata, millised failid pärast koostamist tekkisid

Üks kiire võrdlustabel

Keel	Lähtefail	Tüüpiline käivitus	Mis tekib
Shell	hello.sh	sh hello.sh või ./hello.sh	tavaliselt eraldi koostamisväljundit ei teki
Python	hello.py	python3 hello.py	lähtefail, mõnikord __pycache__
C	hello.c	./hello-c pärast kompileerimist	natiivne binaarfail
C++	hello.cpp	./hello-cpp pärast kompileerimist	natiivne binaarfail
Go	hello.go	go run hello.go või ./hello-go	natiivne binaarfail
Rust	src/main.rs	cargo run või binaarfail	natiivne binaarfail ja kataloogis
Java	Hello.java	target/ java Hello	kataloog target/.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 käivitavat 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 sobib käsuraõppeks hästi, sest koostamise loogika on lihtne.

```
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 käsuga
- `go build` teeb natiivse binaari

Go näitab hästi erinevust käivitamise ja eraldi binaari koostamise vahel.

Rust: koostamistööriist 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 näitab, et mõnes keeles ei koosne koostamine ainult ühest kompilaatorikäsust, vaid tervest töövoost.

Java: kompileerimine ja eraldi käivituskeskkond

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 klassifaili, mida käitab Java käivituskeskond

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.

Spikker

Eesmärk: Erista lähtekoodi, koostatud väljundit ja selle käivitajat. Vali üks paigaldatud tööriistaga keel.

Käsud ja näited

- `sh hello.sh` — tõlgenda POSIX-shelliskript
- `python3 hello.py` — käivita Pythoni lähtefail
- `cc hello.c -o hello-c` — kompileeri C lähtefail käivitatavaks failiks
- `./hello-c` — käivita äsja koostatud C programm
- `c++ hello.cpp -o hello-cpp` — kompileeri C++
- `go build -o hello-go hello.go` — koosta Go binaarfail
- `cargo run` — Rust: koosta ja käivita Cargo projekt
- `javac Hello.java` — Java: loo baitkoodifail `Hello.class`
- `java Hello` — käivita klass JVM-is, ilma `.class`-laiendita

Pane tähele: `./fail` vajab sinu süsteemile sobivat käivitavat faili; lähtekoodi ega teise platvormi binaari ei saa nii lihtsalt käivitada.

LaTeX käsurealt

Loogika

LaTeX on selles raamatus pigem järgmine samm pärast Markdowni, kui tahad jõuda väga hea PDF-väljundini.

See teema on seotud dokumentide koostamise ja vormindamisega, mitte Linuxi põhioskustega.

Kiirülevaade

Eesmärk on teha tekstifailist kvaliteetne PDF, kus vormindus, valemid ja viited on korratavalt ehitatavad.

Käsk või fail	Milleks	Mida tavaliselt näed
<code>pdflatex fail.tex</code>	kompileeri PDF-iks	pikk logi ja <code>.pdf</code> fail
<code>xelatex fail.tex</code>	kasuta tänapäevasemat fondituge	pikk logi ja <code>.pdf</code> fail
<code>latexmk</code>	korda vajalikke samme automaatselt	mitu kompileerimissammu
<code>.tex</code>	lähtefail	jääb tekstifailiks
<code>.aux</code> , <code>.log</code>	abifailid	tekivad PDF-i kõrval

Tüüpilised algaja vead

- eeldatakse, et üks kompileerimiskäik lahendab alati kõik ristviited
- ehmutakse logi pikkusest, kuigi suur osa sellest on tavapärane koostamis-teave
- muudetakse väljundfaile, kuigi tegelik allikas on `.tex`

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 Ubuntu:

```
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
```

Pakette `basictex` ja `mactex-no-gui` ei paigaldata tavaliselt koos.

Minitest

1. Loo fail `tere.tex`.
2. Kompileeri see PDF-iks.
3. Kontrolli, millised väljundfailid LaTeX tekitab.

Lisalugemine

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

Spikker

Eesmärk: LaTeX-i mõte on lihtne: lähtetekst jääb tekstifailiks, aga buildist sünnib väga täpselt juhitud 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.

Lisa A: kopeeritavad minitestid

Siin on lühikesed, peatükkide kaupa jaotatud kopeeritavad harjutused.

Need plokid on teadlikult lühikesed ja kasutavad juba mõnda varem seletatud lühikuju:

- `~` tähendab kodukataloogi
- `mkdir -p` loob puuduva tee koos vahekataloogidega

Vali katseteks uus tühi kataloog. Kui näitenimi on juba kasutusel, muuda seda;
> võib olemasoleva faili üle kirjutada.

Failid ja kataloogid

```
mkdir -p ~/tmp/test1
cd ~/tmp/test1
touch a.txt b.txt
mkdir kaust
mv b.txt kaust/
ls -la
```

Kontroll: `a.txt` on praeguses kataloogis, `b.txt` peab olema alamkataloogis `kaust`. Kontrolli teist käsuga `ls kaust`.

Torud ja suunamine

```
printf 'üks\nkaks\nkolm\n' | wc -l
printf 'tere\n' > proov.txt
printf 'juurde\n' >> proov.txt
cat proov.txt
```

Kontroll: esimese käsu vastus on 3; faili lõplik sisu on kaks rida, `tere` ja `juurde`.

grep ja sort

```
printf 'kass\nkoer\nkass\n' > loomad.txt
grep 'kass' loomad.txt
sort loomad.txt | uniq -c
```

Kontroll: `grep` annab kaks rida; loendus näitab kaks kassi ja üht koera.

Python ja venv

```
mkdir -p ~/tmp/venvtest
cd ~/tmp/venvtest
python3 -m venv .venv
source .venv/bin/activate
python -V
deactivate
```

Kontroll: aktiveeritud keskkonnas peab `python -m pip --version` viitama `.venv`-ile. Tee see kontroll enne `deactivate` käsku; väljumine keskkonda ei kustuta.

Spikker

Eesmärk: Kontrolli tulemust, mitte ainult käsu edukat käivitumist.

Harjutuse kontroll

- `ls kaust` — kas liigutatud fail on õiges kohas?

- `cat proov.txt` — kas vana ja lisatud rida on mõlemad alles?
- `sort loomad.txt | uniq -c` — kaks kassi, üks koer
- `python -m pip --version` — kas aktiivne Python kasutab projekti `.venv-i`?

Pane tähele: Kasuta uut katsekataloogi; näidete `>` suunamine kirjutab vana sisu üle.

Lisa B: spikrite register

Kiirviited tööülesande järgi. Failinimed ja serveriaadressid asenda oma töö omadega; tähenduse ja ohutuse kohta loe vastavat peatükki.

Failid

- `pwd` kus ma olen
- `ls -la` mida siin näha on
- `ls -lt | head` vaata, mis viimati muutus
- `du -ak . | sort -nr | less` leia suurimad failid ja kaustad
- `du -k -d 1 . | sort -nr` võrdle alamkatalooge, ka peidetuid (Linux/macOS)
- `cp` kopeeri
- `mv` liiguta või nimeta ümber
- `rm fail` kustuta fail prügikastita; kontrolli enne sihti
- `rmdir kaust` kustuta tühi kataloog
- `rm -r kaust` kustuta kataloog koos sisuga
- `mkdir -p ~/tmp/proov` loo puuduva teega harjutuskataloog; olemasolev sisu säilib
- `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`-is 25% liigu failis 25% kaugusele
- `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` sorteeeri
- `uniq -c` loenda kõrvutised võrdsed read
- `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 skript mitmerealise sisendi abil
- `cc hello.c -o hello-c` kompileeri C programm
- `go build -o hello-go hello.go` koosta Go binaarfail
- `cargo run` koosta 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 .` koosta konteinerkuju (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

Git-spikker: käsud töövoo järgi

Alusta seisust, vaata muutust, salvesta loogiline samm ja sünkrooni alles siis.

Repo algus

loo, too või ühenda kaugrepo

git init uus kohalik repo

git clone URL kopeeri repo

Seis ja muutus

enne mõista, siis muuda

git status mis seisus repo on

git diff mis tööpuus muutus

Vaheala

vai järgmise kirje sisu

git add fail lisa vahealale

git diff --cached kontrolli

Versioonikirje

salvesta loogiline samm

git commit -m sõnumiga kirje

git commit --amend paranda viimast

Harud

eraldi arenguliinid ja tööharud

git branch näita harusid

git switch main ligu olemasolevale harule

git switch -c parandus loo uus haru

git branch -d haru kustuta ühendatud haru

Kaugrepo

GitHub, GitLab või muu server

git remote -v näita aadresse

git fetch origin too info, ära ühenda

git pull --ff-only uuenda puhas haru

git push -u origin haru saada haru

Ühendamine

pane harud kokku või tõsta oma tööd

git merge haru salita harude ajalugu

git rebase origin/main tõsta oma haru

git merge --abort katkesta merge

git rebase --continue jätku pärast konflikti

Ajalugu ja kontroll

vaata, mis juhtus ja kus harud on

git log --oneline lühike ajalugu

git log --graph --all harude pilt tekstis

git show HEAD viimane kirje

git blame fail rea päritolu

Taaste ja tühistamine

vai rahulikult, mida tagasi võtad

git restore fail võta tööpuu muudus tagasi

git restore --staged fail eemalda vahealt

git revert COMMIT tee vastupidine kirje

git reflog otsi kadunud HEAD-i liikumist

Ajutine töö ja koristus

pane pooleli olev töö korraks kõrvale

git stash push -m pane kõrvale

git stash list näita sahtlit

git stash pop too tagasi ja eemalda

git clean -fd kustuta jälgimata failid

Ettevaatlikult

need käsud võivad ajalugu või faile tugevalt muuta

git reset --hard **git push --force-with-lease** **git branch -D**

GitHubi töö

probleemikirje annab põhjuse, tõmbekutse annab ülevaate koha

issue ülesanne või viga **pull request** muudatuseettepanek

rütm status -> diff -> add -> diff --cached -> commit -> fetch/pull või push

Joonis 9: Git-spikker: käsud töövoo järgi

Git: visuaalne spikker

Kaart paigutab käsud töövoogu. Üksikasjad ja katsetatav näide on Giti peatükis.

Ettevaatust: `git clean -fd` kustutab jälgimata faile, `commit --amend` kirjutab viimase kirje ümber. Pildil lühendatud `-m` vajab sõnumit, näiteks `git commit -m 'Paranda juhend'`. Enne muutmist vaata `git status` ja `git diff --cached`.

Repo algus

- `git init` loo uus repo praeguses kataloogis
- `git clone URL` too olemasolev repo oma arvutisse
- `git config user.name "Nimi"` seadista selle repo versioonikirjete autorinimi
- `git config user.email "nimi@example.com"` seadista selle repo autori e-post; `--global` muudaks kasutaja vaikeseadet kõigis repodes
- `git remote -v` näita kaugrepo aadresse

Harud

- `git branch` näita kohalikke harusid
- `git switch main` liigu põhiharule
- `git switch -c parandus` loo uus haru ja liigu sinna
- `git branch -d parandus` kustuta ühendatud kohalik haru
- `git branch -D parandus` kustuta haru jõuga; kasuta ainult siis, kui oled kindel

Kaugrepo

- `git fetch origin` too serverist värske info, kuid ära ühenda seda praegusesse harusse
- `git pull --ff-only` uuenda kohalik haru ainult siis, kui seda saab teha edasinihkena
- `git push -u origin parandus` saada uus haru kaugreposse ja seo see ülesvoolu haruga
- `git push` saada juba seotud haru uued versioonikirjed kaugreposse
- `git remote prune origin` eemalda kohalikud viited kaugrepo harudele, mida serveris enam pole

Ühendamine ja korrastamine

- `git merge haru` ühenda teine haru praegusesse harusse
- `git rebase origin/main` tõsta oma praeguse haru versioonikirjed värske põhiharu lõppu
- `git merge --abort` katkesta pooleliolev ühendamine
- `git rebase --abort` katkesta pooleliolev rebase

- `git rebase --continue` jätkka rebase'i pärast konflikti lahendamist

Ajalugu ja ülevaatus

- `git log --oneline` näita ajalugu lühidalt
- `git log --graph --decorate --all -n 20` näita harude ja siltide paiknemist
- `git show HEAD` näita viimast versioonikirjet
- `git diff HEAD` võrdle tööpuud viimase versioonikirjega
- `git blame fail` näita, millisest versioonikirjest faili read pärinevad

Taaste ja tühistamine

- `git restore fail` taasta tööfail vaheala sisust; salvestamata muudatus kaob
- `git restore --staged fail` eemalda fail vahealalt, kuid jäta faili sisu alles
- `git revert COMMIT` loo uus versioonikirje, mis võtab vana muudatuse tagasi
- `git reflog` näita, kus HEAD hiljuti käis
- `git restore --source=COMMIT -- fail` taasta tööfail nimetatud versioonikirjest; vaheala ei muutu

Ajutine töö

- `git stash push -m 'kirjeldus'` pane jälgitud failide töö kõrvale; `-u` kaasab ka jälgimata failid
- `git stash list` näita kõrvale pandud töid
- `git stash show` vaata kõrvale pandud töö kokkuvõtet
- `git stash pop` too töö tagasi; eduka rakendamise järel eemalda sahtlist, konflikti korral säilita
- `git stash apply` too töö tagasi, kuid jäta see sahtlisse alles

Andme- ja arendusprojektides kasulik

- `git lfs install` lülita Git LFS kasutaja jaoks sisse
- `git lfs track '*.psd'` määra tulevastele PSD-failidele LFS-reegel; salvesta ka loodud `.gitattributes`, varasem ajalugu ei teisendu
- `git check-ignore -v fail` kontrolli, milline `.gitignore` reegel faili peidab
- `git diff -- . ':(exclude)*.ipynb'` võrdle tööpuu muudatusi, jättes märkmikufailid kõrvale
- `git grep 'muster'` otsi jälgitud failidest kiiresti

Käsud, mida ei tasu paanikas käivitada

- `git reset --hard` viskab tööpuu ja vaheala muudatused ära

- `git clean -fd` kustutab jälgimata failid ja kataloogid
- `git push --force-with-lease` kirjutab kaugrepo haru ajalugu ümber
- `git branch -D` haru kustutab kohaliku haru ka siis, kui Git peab seda ühendamata tööks

Kui plaanid kasutada mõnda neist käskudest, tee enne `git status`, loe väljund läbi ja salvesta vajadusel töö koopia eraldi kataloogi.

Loogika

- `käsk1 ; käsk2` käivita käsud järjest
- `|` suuna väljund edasi
- `>` kirjuta väljund faili; vana sisu kirjutatakse üle
- `>>` 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
- `&&` jätkka ainult edu korral
- `||` jätkka vea korral
- `echo $?` kuva eelmise käsu lõpetuskood
- `set -o pipefail` ära peida toru sees tekkinud vigu
- `type nimi` vaata, kas nimi on alias, sisekäsk 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` jätkka tööd 1 taustal
- `fg %2` too töö 2 ette
- `wait` oota taustatöö lõppemist
- `nohup käsk > fail 2>&1 &` jäta töö sessioonist vähem sõltuvaks
- `disown %1` eemalda töö shelli tööde nimekirjast
- `ps aux | sort -nrk 3 | head` vaata suurima protsessorikasutusega protsesse
- `ps aux | sort -nrk 4 | head` vaata suurima mälu kasutusega protsesse

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

Spikker

Eesmärk: kõige lühemad meespead 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, koostamine
- **Loogika** — torud ja vead
- **Protsessid** — tööd ja signaalid

Pane tähele: See lisa on kõige lühem meespea; kui mõni käsk tundub hägune, mine tagasi vastava peatüki juurde.

Lisa C: sõnastik ja terminoloogia

Levinud mõistete lühiseletused. Inglisekeelseid vasteid vajad käsurea abi ja dokumentatsiooni lugemisel; täieliku teemade kaupa võrdluse leiad inglise-eesti terminoloogiast.

Üldmõisted

- **terminal:** tekstipõhine keskkond, kus kāske sisestatakse
- **CLI** ehk **command-line interface:** käsurealiides; viis arvutiga suhelda kāske 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
- **kāsurida:** üks konkreetne käsk koos argumentidega
- **shell:** käsutõlk, mis loeb kāsurida ja käivitab kāske
- **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

- **standardsisend** ehk **stdin**: programmi vaikimisi sisendvoog
- **standardväljund** ehk **stdout**: programmi tavaline väljundvoog
- **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
- **puhvri tühjendamine** ehk **flush**: puhvri kohene tühjendamine, et väljund jõuaks kohe nähtavale või järgmisse kohta
- **toru**: kuju |, millega ühe käsu väljund suunatakse teise käsu sisendiks
- **übersuunamine**: väljundi või sisendi suunamine faili või mujale
- **lõpetuskood** ehk **exit code**: käsu tulemust kirjeldav kood; tavaliselt tähendab 0 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**: käivitataivate programmide katalooginimi; 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
- **valik -f** ehk **force**: sunnib paljudes käskudes toimingu lõpule või jätab kinnituse küsimata; täpne tähendus sõltub käsust
- **räsi**: lühike sõrmejälg, mis kirjeldab faili sisu
- **krüptoräsi**: räsi, mida kasutatakse tervikluse kontrolliks, näiteks SHA-256

Shell ja tekstitöötlus

- **globbing**: shelli muustrilaiendus kujudele nagu *****, **?**, **[]**
- **jutumärkidega kaitsmine** ehk **quoting**: erimärkide mõju piiramine jutumärkide abil

- **paomärgiga kaitsmine** ehk **escaping**: ühe märgi erikäitumise väljalülitamine, tavaliselt \ abil
- **shellimuutuja**: aktiivse shelli sees hoitav muutuja
- **keskkonnamuutuja**: muutuja, mis antakse edasi alamprotsessidele
- **alias**: lühinimi mõnele pikemale käsule
- **shellis sisseehitatud käsk** ehk **sisekäsk (builtin)**: käsk, mis on shelli enda sees, mitte eraldi programmina kettal
- **reserveeritud sõna** ehk **keyword**: shelli süntaksi osa, näiteks **if**, **then**, **for**, **do**, **done**
- **shellis funktsioon**: shellis defineeritud käsuplokk, mida saab nimega käivitada
- **regulaaravaldis**: mustrikeel tekstis vastete leidmiseks
- **sõne** ehk **string**: märkide jada; sõneliteraali annab selle koodis otseselt kirja
- **filter**: käsk, mis loeb ridu ja väljastab neist ainult vajaliku osa

Võrk ja kaugkasutus

- **host**: võrgus olev sihtmasin; sageli serveri nimi või 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
- **pordiedastus** ehk **port forwarding**: võrguühenduse suunamine ühest pordist teise
- **WSL** ehk **Windows Subsystem for Linux**: viis käitada Windowsis Linux kasutajaruumi

Git ja GitHub

- **repo**: Giti hoidla või repositoorium
- **haru**: eraldi arendusjoon Gitis
- **commit** ehk **versioonikirje**: loogiline muudatuse salvestus Gitis
- **kaugrepo** ehk **remote**: väline repo, millega kohalik repo suhtleb
- **origin**: vaikimisi peamise kaugrepo nimi
- **märgend** ehk **tag**: nimetatud tähis mõne versioonikirje juures
- **väljalase** ehk **release**: teadlikult välja antud versioon, tavaliselt seotud kindla Giti märgendiga (**tag**)
- **hetktõmmis** ehk **snapshot**: säilitamiseks tehtud väljundikoopia, mida järgmine koostamine üle ei kirjuta
- **verstapost**: oluline seis, mis tasub eraldi nime all alles hoida
- **pull request** ehk **tõmbekutse**: GitHubi kaudu esitatud muudatusettepanek harust teise
- **diff**: muudatuste vaade enne või pärast versioonikirje loomist

- **vaheala** ehk **stage**: Giti ala, kuhu valitakse järgmisse versioonikirjesse 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 projekti sõltuvuste 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 käivituskeskkond või virtuaalmasin
- **käivituskeskkond** ehk **runtime**: keskkond, mida programm tööks vajab
- **JVM** ehk **Java virtuaalmasin**: keskkond, 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** ehk **konteinerkuju** (AKITis ka **konteinerpakett**): valmis konteineri aluskihtide kogum

- **konteiner**: konteinerkujust loodud eraldatud käivituskeskkond koos kirjutatava kihiga; võib olla töötav või peatatud
- **registrihoidla** ehk **registry**: koht, kus konteinerkujusid hoitakse ja kust neid alla laaditakse
- **hostitee sidumine** ehk **bind mount**: hostsüsteemi faili või kataloogi sidumine konteineri teega
- **nimega andmekõide** ehk **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

Spikker

Eesmärk: leia mõiste tähendus ning tunne ära ingliskeelne vaste dokumentatsioonis

Põhimõisted

- **CLI** — käsurida
- **GUI** — graafiline liides
- **viip** ehk **prompt** — ootab sisestust
- **shell** — tõlgendab käske
- **lipp** — käsu valik
- **repo** — Giti hoidla

Pane tähele: Täielik inglise-eesti terminoloogia on ka eraldi sõnastikulehel.

Lisa D: edasised võimalused ja lugejate soovid

Milline päris ülesanne jäi õpiku abil lahendamata? Saada soovitus või murekoht, lisades peatüki lingi, kasutatud süsteemi ja koha, kus takerdusid. Ära lisa paroole, võtmeid ega isikuandmeid sisaldavat logi.

Võimalikud süvendused

Vajadus	Edasiõppimise teema
Meeskonnas muudatusi kooskõlastada	Giti koodiülevaatus, fork ja upstream, sõltuvad harud

Vajadus	Edasiõppimise teema
Teha korduvast käsujadast vastupidav tööriist	funktsioonid, argumentide kontroll, ajutised failid ja weakäsitlus
Taastada töö pärast viga või riket	versioonidega varundus, eraldi andmekandja ja taastamiskatse
Ajastada Linuxis teenuseid	systemd teenuse ja ajasti koostöö
Viia andmed interaktiivsesse analüüsi	R või Jupyter; millal valida shell, SQL või märgmik
Mõista keerukamaid otsingumustreid	regulaaravaldiste ülesanderada ja parserite kasutamine

Need on võimalikud jätkuteemad, mitte algaja kohustuslik eelteadmiste loend. Seniste teemade süvendamiseks vaata usaldusväärse lisalugemise kogu.

Spikker

Eesmärk: vali jätkuteema päris ülesande järgi ja saada konkreetne soovitus

Tugevad järgmised teemad

- R ja märgmikud (notebook) — 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: Tagasisides lisa peatüki link, kasutatud süsteem ja takerdumise koht; ära saada paroole ega privaatvõtmeid.

Lisa E: usaldusväärsed viited ja lisalugemine

See lisa koondab teemade kaupa ametlikud ja usaldusväärsed lisalugemise allikad.

Järjekord on üldiselt selline:

1. ametlik alustamisjuhend või ülevaade
2. ametlik teatmik või käsiraamat
3. standard või spetsifikatsioon
4. üks pikem ja hea õppematerjal, kui see on laialt tunnustatud

Kui tahad edasi õppida, alusta iga teema esimesest viitest. Kui vajad täpset üksikasja, vaata teatmikku või käsiraamatut.

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

Spikker

Eesmärk: ametlikud käsiraamatud ja alustamisjuhendid shelli, SSH, Giti, 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 põhikujutus on olemas.

Lisa F: shelli seadistusfailid Bash ja zsh jaoks

Seadistusfailist saad uuele shellile anda ajaloo, viiba ja oma lühikäsud. **Vali kõigepealt oma shelli põhiseadistus; lisa ühised mugavused ainult üks kord.** Kogu faili asendamise asemel täienda olemasolevat seadistust.

Milline shell ja milline fail?

`ps -p $$` näitab praegust shelliprotsessi. `echo "$SHELL"` näitab tavaliselt kontole seadistatud sisselogimisshell; need võivad erineda.

Shell	Interaktiivse seadistuse asukoht
zsh	<code>~/.zshrc</code>
Bash	<code>~/.bashrc</code>
Bashi sisselogimisshell, sageli macOS-i terminalis	<code>~/.bash_profile</code> , mis võib laadida <code>~/.bashrc</code>

Need on tavapärase asukohad; zsh `ZDOTDIR` võib seadistuste kataloogi muuta. Bash loeb sisselogimisel esimese olemasoleva faili järjestusest `.bash_profile`, `.bash_login`, `.profile`.

Enne muutmist tee olemasolevast failist koopia. Näiteks zsh puhul:

```
cp -i ~/.zshrc ~/.zshrc.backup
nano ~/.zshrc
```

`-i` küsib enne olemasoleva varukoopia ülekirjutamist. Kui lähtefaili pole, pole ka midagi varundada. Ära käsitle muid lugemis- või õigusevigu tõendina faili puudumisest.

Bashi põhiseadistus

Lisa faili `~/.bashrc` ainult read, mida seal veel samaväärsel kujul pole:

```
HISTSIZE=5000
HISTFILESIZE=10000
HISTCONTROL=ignoredups
shopt -s histappend
```

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

`HISTSIZE` piirab mälu hoitavat ajalugu, `HISTFILESIZE` ajaloofaili suurust. `ignoredups` väldib vahetult korduvaid kirjeid ning `histappend` lisab shelli lõppedes ajaloo faili senist sisu asendamata.

Kui kasutad Bashi sisselogimisshellina ja sinu profiil veel `.bashrc`-d ei laadi, lisa olemasoleva `~/.bash_profile` lõppu:

```
if [ -f "$HOME/.bashrc" ]; then
. "$HOME/.bashrc"
fi
```

See kontrollib faili olemasolu ja loeb selle praegusesse shelli. Ära kustuta profiilist muid käivituseks vajalikke ridu.

zsh põhiseadistus

Lisa faili `~/.zshrc`:

```
HISTFILE="$HOME/.zsh_history"
HISTSIZE=5000
SAVEHIST=5000
setopt APPEND_HISTORY
setopt HIST_IGNORE_DUPS
```

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

`SAVEHIST` määrab faili salvestatavate ajalookirjete arvu. Viibas tähendavad `%n` kasutajat, `%m` arvutiname, `%~` lühendatud tööteed ja `%#` õigustele vastavat lõpumärki.

Kui tahad eraldi terminalide ajalugu jooksvalt jagada, lisa `setopt SHARE_HISTORY`. See muudab ka seda, millised käsud ülesnoolega ette tulevad; alguses pole seda vaja.

Ühised lühikäsud

Järgmine plokk sobib nii **Bashi** kui ka **zsh** interaktiivsesse seadistusfaili:

```
alias ll='ls -lh'
alias la='ls -la'
alias ..='cd ..'
alias h='history | tail -n 20'
alias dus='du -k -d 1 . | sort -nr'
alias biggest='du -ak . | sort -nr | head -n 10'
export EDITOR=nano
export LESS='-R'
```

`dus` näitab Linuxis ja macOS-is kataloogide mahte ühe taseme kaupa; `biggest` otsib kogu puust. Mõlemad kaasavad peidetud kataloogid ega peida lugemisvigu. `du -d` puudumisel kasuta `biggest`-i.

`EDITOR` annab seda järgivatele programmidele redaktori eelistuse; vali ainult masinas olemasolev redaktor. `LESS=-R` lubab lehitsejal kuvada toetatud värvikoode, mitte kõiki terminali juhtkoode.

Valikuliselt võid lisada kinnitust küsivad aliased:

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
```

Need on mugavus, **mitte varukoopia ega üldine turvagarantii**. Alias ei kehti igas skriptis, selle saab vahele jätta ning teised valikud võivad käitumist muuta.

Oma programmide otsingutee

PATH sisaldab katalooge, kust shell programme otsib. Ära asenda tervet väärtust ühe kataloogiga.

Bashis lisa olemasolev ~/bin üks kord:

```
if [ -d "$HOME/bin" ]; then
    case ":$PATH:" in
        *"$HOME/bin:*)" ;;
        *) PATH="$HOME/bin:$PATH" ;;
    esac
fi
export PATH
```

case valib mustri järgi haru: esimene ei tee midagi, kui tee juba esineb, teine lisab selle ette.

zsh-s on path massiiv seotud muutujaga PATH; typeset -U eemaldab korduvad kirjed:

```
typeset -U path
if [ -d "$HOME/bin" ]; then
    path=("$HOME/bin" "${path[@]}")
fi
export PATH
```

Kataloogi ette lisamine annab selle programmidele eelisjärjekorra. Ära lisa PATH-i juhuslikku allalaadimiste kataloogi.

Arendaja valikulised lisad

Lisa ainult kasutatavate tööriistade lühikäsud:

```
alias gs='git status --short'
alias gd='git diff'
alias gdc='git diff --cached'
alias gl='git log --oneline --decorate -10'
alias venv='python3 -m venv .venv'
alias va='source .venv/bin/activate'
alias dc='docker compose'
alias json='python3 -m json.tool'
alias serve='python3 -m http.server --bind 127.0.0.1 8000'
```

serve jagab praeguse kataloogi sisu ainult selle arvuti kaudu. Käivita see katsekataloogis, mitte seal, kus hoiad saladusi. Lõpeta Ctrl-C-ga.

mkcd ühendab kataloogi loomise ja sinna liikumise. Järgmine funktsioon nõuab täpselt üht argumenti; && väldib liikumist, kui loomine ebaõnnestus:

```
mkcd() {
    if [ "$#" -ne 1 ]; then
        printf 'Kasutus: mkcd kataloog\n' >&2
        return 2
    fi
    mkdir -p "$1" && cd "$1"
}
```

Funktsiooni kasutatakse näiteks `mkcd "$HOME/tmp/uus projekt"`. `return` lõpetab funktsiooni, mitte kogu shelli.

Kontroll ja taastamine

Kontrolli **oma shelli** faili süntaksit enne selle sisselugemist:

```
bash -n ~/.bashrc
```

zsh puhul:

```
zsh -n ~/.zshrc
```

Vaikne tulemus tähendab, et süntaksiviga ei leitud, mitte seda, et kõik käsud on ohutud. Seejärel loe õige fail sisse: `source ~/.bashrc` või `source ~/.zshrc`. `source` käivitab faili sisu praeguses shellis.

Kui seadistus rikub töö, käivita `bash --noprofile --norc` või `zsh -f`. zsh loeb ka sel juhul süsteemi `zshenv`-faili, kuid jätab kasutaja tavalise interaktiivse seadistuse vahele. Paranda fail redaktoris või taasta **ainult oma shelli** varukoopia, näiteks `cp -i ~/.zshrc.backup ~/.zshrc`. Uus tavaline terminaliaken loeb vigase seadistuse uuesti, seega pelgalt uue akna avamine ei pruugi aidata.

Minitest

1. Võrdle `ps -p $$` ja `echo "$SHELL"` vastust. Kas need peavad kattuma?
2. Varunda oma seadistus ja lisa üks alias; kontrolli süntaksit ning proovi aliaist.
3. Miks ei piisa käsu `rm` muutmisest kinnitust küsivaks aliasena?
4. Kuidas pääsed shelli, kui sinu interaktiivses seadistusfailis on viga?

Spikker

Eesmärk: Hoiu üks baasprofiil oma shellile ja lisa ainult vajalikke aliaseid või funktsioone.

Käsud ja näited

- `ps -p $` — kontrolli praegu töötavat shelli

- `cp -i ~/.zshrc ~/.zshrc.varu` — tee olemasolevast failist varu; ära kirjuta varu pimesi üle
- `zsh -n ~/.zshrc` — kontrolli zsh süntaksit ilma käivitamata
- `bash -n ~/.bashrc` — kontrolli Bashi süntaksit ilma käivitamata
- `source ~/.zshrc` — käivita muudetud zsh seadistus praeguses shellis
- `zsh -f` — ava zsh kasutaja rc-faile lugemata
- `bash --noprofile --norc` — ava Bash profile lugemata

Märgid ja valikud

- `~/.zshrc` — interaktiivse zsh seadistus
- `~/.bashrc` — interaktiivse mitte-login Bashi seadistus
- `~/.bash_profile` — login-Bash; võib laadida `.bashrc`
- `PATH` — programmide otsinguteed; järjekord loeb

Pane tähele: Ära kirjuta olemasolevat profiili kogu näidisfailiga üle. `source` käivitab kõik failis olevad käsud.