

# Linux gyakorlatok

## GKNB\_INTM012 Számítógépek működése

### Tartalomjegyzék

1. Linux telepítés.....	1
2. Ismerkedés a parancssorral.....	1
3. Rendszer-közel parancsok.....	4
4. Archívumok kezelése.....	4
5. Csomagok kezelése.....	6
6. Hálózati beállítások.....	8
7. A GitHub kezelése.....	8
8. Shell-szkriptek.....	10

### 1. Linux telepítés

Az első gyakorlaton feltelepítünk egy Ubuntu Server 16.04-t, amely egy sima konzol alapú operációs rendszer. Ehhez az Oracle Virtual Box virtualizációs szoftvert fogjuk használni. Figyelni kell, hogy a számítógép BIOS-ában bekapcsoljuk a hardveres virtualizáció támogatást, amelyet Intel-es (Intel VT) és AMD-s (AMD-V) gépek esetében is másképp neveznek.

A telepítés folyamata megnézhető [INNEN](#).

### 2. Ismerkedés a parancssorral

A parancssor, vagy más néven terminál, esetleg konzol, egy fekete képernyő, amelybe beírhatjuk a kívánt vezérlő parancsunkat. A parancssorban általában ezt látjuk bal oldalon:

**username@hostname:~\$**

Nézzük, mi mit jelent:

1. a @ előtti név a felhasználónevet mutatja, akivel épp be vagyunk jelentkezve
2. a @ utáni név a hoszt nevet mutatja, ez az a név, ahogyan a számítógépet elneveztük telepítéskor
3. a : utáni ~ (tilde) jel a HOME mappát jelöli, ez az ahol épp tartózkodunk; HOME mappa alatt a home/user mappát kell érteni.
4. a \$ jel az úgynevezett SHELL mark.

Kezdjük az elején! Az egyik legfontosabb parancs a `sudo`, amellyel `root` (adminisztrátori) jogosultságot szerezhethünk magunknak. Használata:

**sudo [OPCIÓK] parancs**

A `sudo`-t tehát nem használjuk önmagában, hanem valamely paranccsal együtt, amelyhez épp szükség van `root` jogra. Ilyen pl. a lemezre írás. A opciók használata opcionális, később még beszélünk róluk.

**sudo su** – átváltás root felhasználóra (super user)

**exit** – visszalépés rootból sima felhasználóba (vagy kilépés)

Ha le szeretnénk kérdezni a jelenlegi munkamappát, azt a **pwd** paranccsal tudjuk megtenni. Ez viszont látszódik a parancssorból is, ezt már részleteztük fentebb. Pl.:

```
user@host:~/mappa1/mappa2$ pwd
Output: home/user/mappa1/mappa2
```

Ha ki szeretnénk listázni egy mappa tartalmát, azt az **ls** paranccsal tudjuk megtenni. Amennyiben a rejtett fájlokat és mappákat is látni szeretnénk úgy használunk kell az **-a** opciót a paranccsal. Pl.:

```
ls # kilistázza a mappa tartalmát
ls -a # kilistázza a mappa összes tartalmát, beleértve a rejtett elemeket is
```

Láthatjuk, hogy itt jelenik meg először használatban az opció, melyet fentebb már említettünk. Minden parancsnak van opciója, több vagy kevesebb. Mivel az összes opciót lehetetlenség észben tartani, ezért hívjuk segítségül a **HELP**-et. Erre is két lényeges lehetőségünk van. Viszont az ebben a dokumentációban szereplő opciókat illik fejből tudni. :-)

```
man parancs_neve # a man csomag a linux telepítésekor kerül telepítésre az alap alkalmazásokkal
parancs --help # a --help opció alapból be van építve a linuxba
```

Mappák létrehozása az **mkdir** paranccsal végezhető el, lássuk hogyan.

```
mkdir mappa_neve # egy mappa létrehozása
mkdir mappa1 mappa2 mappa3 ... # több mappa létrehozása egy lépésben
```

Mappák közötti navigálást a **cd** paranccsal tudunk végezni.

```
cd mappa # közvetlen ugrás a mappába, amennyiben ez a munkamappában található
cd ~/eleresi/utvonal # a teljes útvonal megadásával ugrunk a kívánt mappába
```

Nézzünk meg néhány speciális esetet!

```
cd # egy lépéssel a home mappába ugrik, bárholnán
cd ~ # ugyanazt csinálja mint a fenti
cd .. # egy szinttel feljebb ugrik a mappa-hierarchiában
cd - # egyel vissza, tehát az utoljára tartózkodott mappába ugrik vissza
cd / # egy lépéssel a root mappába ugrik, bárholnán
```

Hozzunk most létre egy szöveg fájlt. Ennek több módja van.

```
touch szoveg.txt # a touch parancs létrehoz egy üres fájlt
```

Ezután valamilyen szövegszerkesztővel tudunk beleírni. Pl. nano, vim, jed, stb.

**nano szoveg.txt**

Ekkor ez a képernyő jelenik meg, ahova beírjuk a szöveget.



A másik módja a szöveg fájl létrehozásának, ha egyből a szerkesztővel hozzuk létre:

**nano szoveg.txt**

Most másoljuk át ezt a fájlt valahova, pl. egy mappába.

```
cp mit hova # a paraméterek sorrendje: 1. mit, 2. hova
cp szoveg.txt mappa_neve # a cp parancs végzi a másolást a kívánt mappába
```

Teljes elérési útvonal megadásával pedig így néz ki:

```
cp szoveg.txt ~/eleresi/utvonal
```

Amennyiben a másolandó elemnek új nevet szeretnék adni az új helyen, akkor:

```
cp szoveg.txt ~/eleresi/utvonal/uj_nev.txt
```

Hasonlóan működik az elemek áthelyezése is:

```
mv mit hova
mv szoveg.txt mappa_neve
mv szoveg.txt ~/eleresi/utvonal
mv szoveg.txt ~/eleresi/utvonal/uj_nev.txt
```

Az mv parancsot szokták használni helyben átnevezésre is:

```
mv szoveg.txt uj_nev.txt
```

Töröljük most ki ezt a fájlt.

```
rm szoveg.txt
```

Vagy ha nem a munkamappánkban van akkor, a teljes elérési útvonal megadásával.

```
rm ~/eleresi/utvonal/szoveg.txt
```

Töröljük most ki a mappákat. Üres mappát az alábbi paranccsal tudunk törölni.

```
rmdir mappa_neve # kizárólag üres mappát töröl
```

Ha a törlendő mappa nem üres, akkor pedig:

```
rm -r mappa_neve # rekurzívan törli ki a mappát tartalmával együtt, erre utal a -r opció is
```

### 3. Rendszer-közel parancsok

```
df # megmutatja a meghajtók állapotát
```

```
du mappanév # megmutatja a mappa méretét
```

```
free # megmutatja a memória állapotát
```

```
top # információval szolgál a Linux rendszerről
```

```
ps # kilistázza a futó folyamatokat
```

```
kill ID # leállítja/megöli az ID sorszámú folyamatot
```

```
clear # törli a képernyő tartalmát
```

```
halt # leállítja a rendszert, tulajdonképpen shutdown
```

```
reboot # újraindítja a rendszert
```

```
shutdown -h # ugyanaz mint halt
```

```
shutdown -r # ugyanaz mint a reboot
```

### 4. Archívumok kezelése

Mik azok az archívumok? Egyszerűen úgy lehet őket jellemezni, hogy sima becsomagolt fájlok, melyeknek a kiterjesztése `.tar`. Az archívumok tehát nem tömörített fájlok! Archívumok kezelése a linuxban a `tar` parancs segítségével történik. Nézzük meg a használatát konkrét példákon keresztül.

Új archívum készítése:

```
tar -cvf fájlnev.tar ~/mit/teszünk/bele
```

# kötelező opciók: `-c`: create new; `-f`: archive fájl;

# opcionális opció: `-v`: verbosely (kiírja a folyamatot)

A `tar` parancs szerencsére nem csak sima archívumokat képes generálni, de tömöríteni is képes. Kétféle tömörített fájl képes létrehozni, egyik a klasszikus GZIP, másik a BZIP. Ezek kiterjesztései

.tar.gz vagy .tar.bz2. Ahhoz, hogy ezeket létrehozzuk, további opciók használata szükséges. Lássuk:

```
tar -cvzf fájlnev.tar.gz ~/mit/teszünk/bele # a -z opció kezeli a GZIP fájlokat
```

```
tar -cvjf fájlnev.tar.bz2 ~/mit/teszünk/bele # a -j opció kezeli a BZIP fájlokat
```

Megerősítés használata az archívum létrehozásakor:

```
tar -czwf fájlnev.tar.gz ~/mappa/* # a -w opció megerősítést kér a parancs végrehajtása előtt, így kiválasztható, hogy a mappából mit szeretnénk becsomagolni; a * a mappában levő összes elemet jelenti; a kiválasztás egy „y” vagy „n” lenyomásával történik.
```

Archívum létrehozása egy másik munkamappába:

```
tar -cvf ~/új/munkamappa/fájlnev.tar ~/mappa/* # ilyenkor a fájl neve elé írjuk a kívánt munkamappa elérési útját
```

Meglévő archívumhoz további elemek hozzáadása az append opcióval. FIGYELEM! Ez csak sima .tar fájlknál működik, tömörített fájlknál már NEM.

```
tar -rvf fájlnev.tar mit_adok_hozzá.txt # a -r opció végzi el a hozzáfűzést.
```

Archívumok tartalmának kilistázása:

```
tar -tf fájlnev.tar # a -t opció kilistázza a tartalmat; ugyanúgy működik ZIP fájlokkal is.
```

```
tar -tf fájlnev.tar | wc -c # ezzel a méretét tudjuk lekérdezni Bájtokban.
```

```
tar -tf fájlnev.tar | wc -l # ezzel pedig a benne levő elemek számát kérdezzük le.
```

Archívumok kicsomagolása:

```
tar -xf fájlnev.tar # a -x (extract) opcióval tudjuk kicsomagolni a fájlokat; ugyanúgy működik ZIP fájlknál is a -z és -j opciók használatával.
```

Ha csak meghatározott elemet/elemeket szeretnénk kicsomagolni, tehát nem az egész archívumot, akkor a parancs végén felsoroljuk, hogy mit szeretnénk kicsomagolni.

```
tar -xf fájlnev.tar mit csomagoljon ki
```

Alapértelmezetten a tar parancs oda csomagolja ki az archívumot, ahol épp tartózkodunk, tehát a munkamappába. Ha máshova szeretnénk kicsomagolni, akkor használnunk kell a -C (change working directory) opciót. Figyeljünk oda, ez most NAGY C betű, nem kis c, mint a create-nél.

```
tar -xf fájlnev.tar -C új/munkamappa/ mit csomagoljon ki
```

Amennyiben csak meghatározott típusú/nevű fájlokat szeretnénk csak kiszedni az archívumból, akkor ezt a `--wildcards` opció és a megadott szabály alapján tudjuk megtenni.

```
tar -xf fájlnev.tar --wildcards "*.txt" # ez a szabály pl. minden .txt kiterjesztésű fájlt kicsomagol; a "abc*" szabály pl. minden abc-vel kezdődő nevű elemet kicsomagol.
```

## 5. Csomagok kezelése

A Debian-alapú linuxok, mint pl. az Ubuntu is, debian csomagokat kezelnek. Ezek kezelése történhet repositoryból, vagy manuálisan is. Az előbbi esetében az `apt-cache` és `apt-get` parancsokkal, utóbbi esetben a `dpkg`-val kezeljük azokat.

### Advanced Packagigng Tool (APT):

```
apt-cache pkgnames # kilistázza a rendszeren elérhető csomagokat
```

```
apt-cache pkgnames csomagnév # kilistázza a „csomagnév”-vel kezdődő csomagokat
```

```
apt-cache search csomagnév # rövid leírással együtt kilistázza a keresett csomaghoz hasonló csomagokat
```

```
apt-cache show csomag_neve # csomaginformációkat szolgáltat
```

```
apt-cache showpkg csomag_neve # megmutatja a csomag telepítésének előkövetelményeit és, hogy azok már telepítve vannak-e
```

```
sudo apt-get update # frissíti a rendszercsomagok listáját
```

```
sudo apt-get upgrade # frissíti az összes rendszercsomagot, melyre elérhető frissítése
```

```
sudo apt-get install csomagnév # telepíti és egyben frissítést keres a kívánt csomagra; tehát: ha nincs telepítve, akkor telepíti, ha más telepítve van, akkor frissítést keres rá
```

```
sudo apt-get install csomagnév1 csomagnév2 ... # több csomag telepítése egy sorban
```

```
sudo apt-get install '*név_részlet*' # több csomagot telepít, amelyek tartalmazzák „név_részlet”-et
```

```
sudo apt-get install csomagnév --no-upgrade # csak telepít; tehát kikapcsoltuk a frissítés funkciót
```

```
sudo apt-get install csomagnév --only-upgrade # csak frissít; tehát kikapcsoltuk a telepítés funkciót
```

**sudo apt-get download csomagnév** # a csomag telepítőjének letöltése

**sudo apt-get remove csomagnév** # törli a feltelepített csomagot a rendszerről

**sudo apt-get purge csomagnév** vagy:

**sudo apt-get remove --purge csomagnév** # mindenestől törli a csomagot (config fájlok, futás közben létrejött cache fájlok, stb...)

**sudo apt-get clean** # merevlemez megtisztítása a felesleges csomagfájloktól

**sudo apt-get --download-only source csomagnév** # a csomag forráskódjának letöltése (csak forráskód letöltés, nem telepítés, nem kicsomagolás)

**sudo apt-get source csomagnév** # a csomag forráskódjának letöltése és kicsomagolása

**sudo apt-get --compile source csomagnév** # a forráskód letöltése, kicsomagolása és fordítása (compile)

**sudo apt-get autoremove** # törli a már elavult/feleslegessé vált csomagokat a rendszerről; általában az upgrade után automatikusan fel szokta ajánlani

**sudo apt-get autoremove csomagnév** # törli a kívánt csomagot a függő csomagokkal együtt

A fentieket kiegészítve meg kell jegyezni, hogy az apt-cache és apt-get parancsok alacsony szintű linux parancsok. Ezek használatának megkönnyítése érdekében létrehozták a magas szintű apt parancsot (eredetileg 2014-ben), mely lényegesen leegyszerűsíti a repository csomagok kezelését. Az apt bejelentését követően mégis csak a 2016-os linux disztribúciókban debütált először.

Egy apró lábjegyzet a Linux Mint felhasználók számára: Néhány évvel ezelőtt a Linux Mint implementált egy Python wrapper-t, melyet szintén apt-nek neveztek el (véletlen egybeesés). Tehát a Linux Mint-es apt, nem egyenlő ezzel a csomagkezelő apt-vel, amit most tárgyalunk. Lássuk tehát az apt parancsokat:

**sudo apt install** # a csomag feltelepítése

**sudo apt remove** # törli a csomagot

**sudo apt purge** # konfigurációs fájlokkal együtt törli a csomagot

**sudo apt update** # frissíti a csomagok listáját

**sudo apt upgrade** # telepíti az elérhető frissítéseket

**sudo apt autoremove** # törli az elavult csomagokat

**sudo apt full-upgrade** # az előkövetelmények automatikus kezelésével telepíti az elérhető frissítéseket

**sudo apt search csomagnév** # rákeres a kérdéses csomagra

**sudo apt show csomagnév** # megmutatja egy csomag részleteit

Újjonnan bevezetett apt parancsok:

**sudo apt list** # Kritériumokkal együtt kilistázza a csomagot (telepített, frissíthető, stb.)

**sudo apt edit-sources** # Módosítja a források listáját

## **DPKG:**

A `dpkg` paranccsal akkor szoktuk manipulálni a csomagokat, ha azok nem repositoryban vannak, hanem valamilyen webtárhelyről letöltjük őket. Ezek a csomagok `.deb` kiterjesztésűek.

**`dpkg -i csomagnév.deb`** # a `-i` opció telepíti (install) a csomagot.

**`dpkg -l`** # kilistázza a rendszerre telepített összes `.deb` csomagot

**`dpkg -l csomagnév`** # megmutatja, hogy a kívánt csomag telepítve van-e vagy sem

**`dpkg -s csomagnév`** # megmutatja a csomag státuszát/információit

**`dpkg -r csomagnév`** # törli (remove) a telepített csomagot

**`dpkg -p csomagnév`** # mindenestől törli a csomagot; hasonló mint az `apt-get purge`

**`dpkg -c csomagnév.deb`** # megmutatja a csomag tartalmát

**`dpkg -L csomagnév`** # megmutatja a csomag telepítési helyét

**`dpkg -R --install mappanév/*`** # telepíti az összes csomagot a mappából (rekurzívan)

**`dpkg --unpack csomagnév.deb`** # kicsomagolja a fájlt, de nem végez rajta semmilyen konfigurációt

**`dpkg --configure csomagnév`** # konfigurálja/rekonfigurálja a sérült/hibás csomagot

**`dpkg --forget-old-unavail`** # „elfelejti” a törölt és/vagy már nem elérhető csomagokat

## **6. Hálózati beállítások**

**`ifconfig`** # kilistázza az összes aktív hálózati interfészt, információkkal együtt

**`ifconfig interfésznév`** # megmutatja a kívánt interfész információit

**`ifconfig interfésznév up`** vagy:

**`ifup interfésznév`** # interfész bekapcsolása; `sudo` kell hozzá

**`ifconfig interfésznév down`** vagy:

**`ifdown interfésznév`** # interfész kikapcsolása; `sudo` kell hozzá

**`ifconfig interfésznév 172.16.25.125`** # interfész statikus IP címének beállítása

**`ifconfig interfésznév netmask 255.255.255.224`** # interfész netmask-jának beállítása.



```
ifconfig interfésznev broadcast 172.16.25.63 # interfész broadcast címének beállítása.
```

```
ifconfig interfésznev 172.16.25.125 netmask 255.255.255.224 broadcast 172.16.25.63 # IP, netmask, broadcast beállítása egy lépésben
```

```
ifconfig interfésznev hw ether AA:BB:CC:DD:EE:FF # MAC cím átírása
```

## 7. A GitHub kezelése

A Repository (továbbiakban repo) tulajdonképpen egy adatszerkezet, amely meta-adatokat tartalmaz. Általában szoftverek/kódok verziókezelésére alkalmazzák, főként amikor többen dolgoznak ugyanazon a feladaton. A verziókezelő rendszerek lehetnek elosztottak (pl. Git), vagy centralizáltak (pl. Subversion), ettől függően a repoban levő információ lehet számos példányba duplikált, vagy egy adott szerveren karbantartott. A gyakorlatokon mi a Git rendszert tanuljuk, azon belül pedig a GitHub szolgáltató ingyenesen igénybe vehető verziókezelő rendszerét.

Először is, hogy használni tudjuk a Gitet a számítógépünkön, fel kell telepíteni a `git` csomagot a szokásos módon.

```
sudo apt-get install git
```

Ezután konfigurálni kell, hogy a git felismerje az identitásunkat.

```
git config --global user.name "Teljes Név vagy username" #  
megadhatjuk a teljes nevünket is amivel regisztráltunk, vagy a felhasználónevet  
git config --global user.email email@email.com # az email cím amivel  
regisztráltunk
```

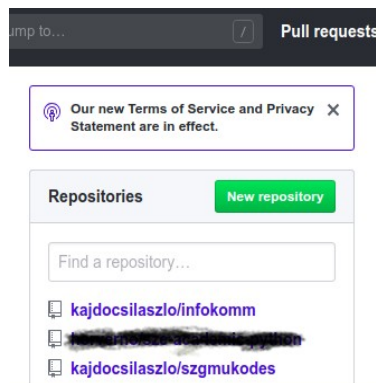
Repo létrehozása a saját localhostunkon két lépésből áll. Létre kell hozni egy mappát, majd azon belül inicializálni kell.

```
mkdir repomappa  
cd repomappa  
git init # ezzel inicializáljuk a repot
```

A másik egyszerűbb módja, amikor leklónozzuk a repot a szerverről a saját localhostunkra.

```
git clone https://github.com/felhasznalonév/reponév.git
```

De, ne szaladjunk ennyire előre. Ahhoz, hogy leklónozzunk egy meglévő repot, először létre kell hozni. Regisztráljunk először a GitHub honlapján, majd a sikeres regisztrációt követően, az alábbi képet láthatjuk a kezdőfelületen:



A zöld "New repository" gombra kattintva hozhatunk létre egy új repot, majd a következő felület tárul elénk:

Adjunk valami nevet a reponak a felső szövegmezőben, majd opcionálisan adhatunk leírást is. Érdemes legalul kipipálni a README fájl inicializálását, ezzel létre hozzuk a repohoz tartozó README fájlt. Választhatunk még .gitignore és LICENCE fájlokat, de ezeket most alapértelmezetten hagyjuk. Végül a "Create repository" gombbal létrehozhatjuk a repot. Ezután már klónozható a repo a saját gépünkre, a fentebb leírt paranccsal.

Miután elvégeztük a saját gépünkön a kívánt módosításokat/frissítéseket, ezeket fel kell töltenünk a szerverre. Ezt az alábbi folyamattal tudjuk megtenni:



```
git add . # hozzáadunk minden módosítást az indexbe
git commit -m "Ide írunk valami kommentet" # véglegesítjük a módosításokat,
azaz commitoljuk a HEAD-ben
git push origin master # végül pusholjuk a szerverre a master branchbe
```

Később, amikor valaki más is módosít a repon, naprakésszé kell tennünk a saját gépünkön levő változatot is, vagyis szinkronizálnunk kell a legfrissebb változatra, ezt az alábbi paranccsal tudjuk megtenni.

```
git pull # ezzel leszívjuk a szerverről a legfrissebb változatokat
```

## 8. Shell-szkriptek

Kezdőlépések:

1. Kell egy Linux terminál/parancssor (Ctrl+Alt+T).
2. Ellenőrizzük, hogy milyen Shell interpreterünk van. Futtassuk le az **echo \$SHELL** parancsot. Általában ezeket a válaszokat várjuk: **/bin/sh** vagy **/bin/bash**. Ha nincs semmilyen, akkor értelemszerűen telepíteni kell.
3. A shell-szkriptben a dollár jel (\$) a shell változókat jelöli.
4. Az **echo** parancs azt írja ki a képernyőre amit legéltünk.
5. A Linux parancsoknak sajátos szintaktikájuk van, a Linux nem tolerálja a hibákat. Ha rosszul írunk egy parancsot, nem kell semmilyen károsodástól tartanunk, egyszerűen csak nem fog lefutni.
6. **#!/bin/bash** – ez az ún. “Shebang”. Mindig ez a szkript első sora, ez alapján találja meg a fordítót.

Shell-szkriptek írása és futtatása:

1. Nyissuk meg a terminált (Ctrl+Alt+T).
2. Navigáljuk **cd** paranccsal oda, ahol dolgozni szeretnénk.
3. Opcionális lépés: Hozzuk létre az üres fájlt: **touch hello.sh** Ne felejtsük a **.sh** kiterjesztést, amely a shell-szkriptek kiterjesztése.
4. Ha átugorjuk a 3-as lépést, akkor egyből szövegszerkesztővel is létrehozhatjuk a szkript fájlt: **nano hello.sh** (Azt a szövegszerkesztőt használjuk amelyiket a legjobban szeretjük. Órákon a nano-t használjuk).
5. Elmentjük a fájlt. Nanoban Ctrl+x és `y`.
6. Ezt követően futtathatóvá kell tennünk a szkriptünket, mivel eddig csak egy sima szövegfájl: **chmod +x hello.sh** (`+` jellel jogosultságod adunk a fájlnek, `-` jellel elveszünk. pl. +rwx vagy -rwx {read; write; execute}).
7. Végül le kell futtatni a szkriptet: **./hello.sh** (Linuxban a futtatható dolgokat `./` paranccsal tudjuk futtatni).

## 1. szkript

Írjunk egy olyan programot amely, a kiírja a terminálra, hogy `Hello World!`

```
#!/bin/bash
# Ez egy megjegyzés. Megjegyzéseket a `#` jelle tudunk a kódba írni. Ezeket a
fordító nem veszi figyelembe.
echo "Hello World!"
exit 0 # az exit 0 mindig a helyes programfutás utáni kilépést jelenti
```

## 2. szkript

Ez a szkript megmondja a felhasználónevet (**username**) és kilistázza az épp futó folyamatokat (**ps**).

```
#!/bin/bash
echo "Hello $USER"
echo "Az éppen futó folyamatok:"
ps # meghívjuk a korábban tanult `ps` parancsot
exit 0
```

## 3. szkript

Shell változó alkalmazása, és input beolvasása billentyűzetről. Kérjük be a felhasználótól a vezetékes- és keresztnévét, majd írjuk ki azt.

```
#!/bin/bash
echo "Hello $USER"
echo -n "Adja meg a vezeteknevet: " # a `-n` kikapcsolja az echo sorvégi
sortörését
read vnev; # beolvassuk az értéket a `vnev` shell változóba, a sor végén
kötelezően pontosvessző! Deklaráláskor nem kell elé a `$` jel.
echo -n "Adja meg a keresztnévet: "
read knev;
echo "Köszönöm kedves $vnev $knev. Viszontlatásra!" # a változók meghívásakor
már kell a `$` jel.
exit 0
```

## 4. szkript

Írjunk egy egyszerű kalkulátor programot. Bekérünk két darab számot a billentyűzetről, majd kiírjuk ezek összegét, különbségét és szorzatát.

```
#!/bin/bash
echo "Egyszeru kalkulator program. Adjon meg ket szamot!"
echo -n "Elso szam: "
read a;
echo -n "Masodik szam: "
read b;
# itt most két lehetőség van: 1. bevezetünk új változókat az eredményeknek, vagy
2. egyből kiíratjuk az eredményt
# az 1. megoldás:
ossz=$((a+b))
kul=$((a-b))
szor=$((a*b))
echo $ossz
echo $kul
```

```

echo $szor
# a 2. megoldás:
echo $((a+b))
echo $((a-b))
echo $((a*b))
exit 0

```

## 5. szkript

Az if ... else if ... else elágazás. Írjunk programot, amely bekér egy számot és attól függően, hogy a szám kisebb, nagyobb vagy egyenlő 5-el, kiírja a képernyőre a megfelelő üzenetet.

```

#!/bin/bash
echo -n "Adjon meg egy számot: "
read szam;
if [ $szam -lt 5 ]; then # az -lt jelentése kisebb (lower than)
echo "A szam kisebb mint 5."
elif [ $szam -gt 5 ]; then # a -gt jelentése nagyobb (greater than)
echo "A szam nagyobb mint 5."
else
echo "A szam pontosan 5."
fi # az if-et mindig fi-vel zárjuk
exit 0
# további logikai operátorok: -le: kisebb vagy egyenlő; -ge: nagyobb vagy
# egyenlő; -eq: egyenlő
# másik szintaktika, ahogyan leírható a feltétel: if (( feltétel )); then ... fi

```

## 6. szkript

While ciklus. Olvassunk be egy 10-nél kisebb számot, mindaddig, amíg nem érkezik helyes input. Hibaüzenettel kísérve kérje be újra az értéket, ha rossz érték érkezik a bemeneten.

```

#!/bin/bash
echo "Olvasson be egy 10-nel kisebb számot."
read szam;
while (( $szam > 9 )) # a while ciklust do és done kulcsszavakkal nyitjuk és
# zárjuk
do
echo -n "Hibas input! Adja meg ujra: "
read szam
done
echo "A megadott szam: $szam"
exit 0

```

## 7. szkript

For ciklus. Kérjünk be egy `n` számot, majd írassuk ki a számokat 0-tól n-ig.

```

#!/bin/bash
echo -n "Adjon meg egy számot: "
read n;
for (( i=0; i<=n; i++ )) # másik lehetőség: for i in $(seq 0 n)
do # hasonlóan itt is do és done-al kezdjük és zárjuk
echo $i
done
exit 0

```

## 8. szkript

Case elágazás. A korábbi kalkulátor programot alakítsuk át úgy, hogy a felhasználó egy menüből válassza ki melyik műveletet szeretné végrehajtani.

```
#!/bin/bash
echo -n "Adja meg az elso szamot: "
read a;
echo -n "Adja meg a masodik szamot: "
read b;
echo "Valasszon az alabbi menupontok kozul!"
echo "1 - osszeadas"
echo "2 - kivonas"
echo "3 - szorzas"
echo "egyeb - kilepes"
read opcio;
case $opcio in # szintaktika: case változó in opciók ... esac
1)echo "Az összeg: " $((a+b));; # megnyomjuk a 1-et, akkor összeadás
2)echo "A különbség: " $((a-b));; # megnyomjuk a 2-et, akkor kivonás
3)echo "A szorzat: " $((a*b));; # megnyomjuk a 3-at, akkor szorzás
*)echo "Kilep!" # megnyomunk bármi mást, akkor kilép
exit;;
esac # a case struktúrát mindig esac zárja
exit 0
```

## Függvények

A függvények olyan szubrutinok, amelyek valamilyen speciális feladatot látnak el. Csak egyszer kell őket definiálni és addig nem csinálnak semmit, amíg nem hívjuk őket meg. Emiatt flexibilitást adnak a kódunknak. Két féle módon hozhatunk létre függvényeket a shell-szkriptekben:

1. **function függvénynév** { # használjuk a function kulcsszót  
... # ide jönnek a parancsok  
}
2. **függvénynév()** { # simán a nevével hozzuk létre, de kell a ()  
... # ide jönnek a parancsok  
}

## 9. szkript

Hozzunk létre egy függvényt, amely függvényhívás esetén kiírja, hogy volt függvényhívás, ellenkező esetben pedig, hogy nem volt. A megoldáshoz használjunk parancssori paramétert az alábbi módon:

```
./szkriptnév.sh paraméter1
```

```
#!/bin/bash
function f1 { # definiáljuk az f1 függvényt a function kulcsszóval
echo "Volt függvényhívás!"
}
if [ $# -ne 1 ]; then # azt teszteli hogy adtunk-e parancssori paramétert; a $#
változó tárolja a megadott paraméterek számát!
echo "Hibás használat! Használjon paramétert!" # ha nem adunk meg paramétert,
akkor jelezzük a hibát
echo "A helyes használat: $0 <1/0>" # a $0 változó a szkript nevét tárolja
else
if [ $1 = 0 ]; then # a $1 az első paraméter értékét tárolja; akárhány
paraméterünk lehet, ezek számozása $1,$2,$3,...,$n-ig. Ha a paraméter=0, akkor
nincs függvényhívás
echo "Nem volt függvényhívás!"
elif [ $1 = 1 ]; then # ha a paraméter=1, akkor meghívjuk a függvényünket
```

```

f1 # ez a függvényhívás
else
echo "Hibás paraméter!" # minden más értékre hiba
fi
fi
exit 0

```

## 10. szkript

Nézzük meg, hogyan működnek a lokális és globális változók. A lokális változó mindig csak onnan elérhető, ahol létrehoztuk, a globális pedig mindenhol hozzáférhető.

```

#!/bin/bash
f2() { # most function kulcsszó nélkül hozzuk létre a függvényt
i=10 # ő a globális változó
local j=20 # lokális változót a local kulcsszóval hozunk létre

echo "i értéke a függvényen belül: $i"
echo "j értéke a függvényen belül: $j" # kiíratjuk a változók értékeit a
függvényen belül
}

echo "i értéke a függvényen kívül: $i"
echo "j értéke a függvényen kívül: $j" # itt még nem lesz értéke egyik
változónak sem, mivel nem történt meg az inicializálás, vagyis még nem hívtuk
meg a függvényt

f2 # most meghívjuk a függvényt, tehát a változók is inicializálódnak

echo "i értéke a függvényen kívül: $i"
echo "j értéke a függvényen kívül: $j" # ismét kiíratjuk, de csak a globális
változó értéke látszódik a függvényen kívül
exit 0

```

## 11. szkript

Függvényrekurzió: akkor beszélünk rekurzióról, mikor a függvény önmagát hívja meg. Írjunk egy programot, amely faktoriális számítást végez függvényrekurzió segítségével.

Első változat mikor egész szám típusú (int) kimenettel dolgozunk. Hátrány, hogy a \$? változó csak 0-255 közötti értékeket képes kezelni, összesen tehát 256 értéket. Emiatt max. csak 5 faktoriálisig működik ez a változat.

```

#!/bin/bash
fact( ) {
local num=$1 # a `num` megkapja a függvény paraméterét, a `$1`-et
if [ $num = 0 ]; then # érték összehasonlítás `=` jellel, nem `-eq`-val
ret=1 # a 0 faktoriálisa 1; nincs mit számolni
else
temp=$((num-1))
fact $temp # itt a rekurzív függvényhívás
ret=$((num*$?)) # a visszatérési érték `num*$?`, ahol a `$?` az utoljára
pufferelt értéket tartalmazza; korlátja, hogy csak 256 értéket tud kezelni
fi
return $ret
}
echo -n "Adjon meg egy számot: "
read szam;
fakt $szam

```

```

echo "A(z) $szam faktoriálisa: $?" # ez a megoldás csak fakt 5-ig működik,
mivel fakt 5=120; fakt 6=720 kellene hogy legyen, ehelyett 208-at kapunk, ami a
többszörös túlcsordulás miatt van: 256+256=512, 720-512=208
exit 0

```

Második változat mikor sztring típusú kimenettel dolgozunk, tehát az eredményt simán kicchozzuk return helyett. Ilyenkor nagyobb számok faktoriálisát is kitudunk számolni.

```

#!/bin/bash
fakt() {
local num=$1
if [ $num = 0 ]; then
ret=1
else
temp=$((num-1))
temp2=$((fakt $temp)) # bevezetünk egy új `temp2` változót, melyhez
hozzárendeljük a `fakt $temp` visszatérési értékét; tehát nem a `$?-t`
használjuk
ret=$((num*temp2)) # a `temp2`-vel szorozzuk be a `num`-ot
fi
echo $ret # return helyett echo
}

echo -n "Adjon meg egy szamot: "
read szam;
echo "A(z) $szam faktoriálisa: $(fakt $szam)" # közvetlenül a `fakt $szam`-ot
íratjuk ki
exit 0

```

## 12. szkript

Fibonacci sorozat kiírása szintén rekurzióval. A sorozat elemei:

0.	1.	2.	3.	...	n.
0	1	1	2	...	(n-1)+(n-2)

```

#!/bin/bash
fib( ) {
n=$1 # a függvény paramétertől várjuk, hogy hány elemet írassunk ki a
sorozatból
if [ $n -lt 2 ]; then
echo $n # ha az `n<2`, akkor az eredmény maga `n`; látható a fenti mintából
else
((--n)) # egyel csökkentem `n` értékét; ún. dekrementálom
a=$((fib $n)) # ő lesz `fib (n-1)`
((--n)) # megint egyel csökkentem `n` értékét
b=$((fib $n)) # ő lesz `fib (n-2)`
echo $((a+b)) # az eredmény `fib (n-1)+fib (n-2)`
fi
}

echo "Adjon meg egy szamot:"
read szam;
for i in $(seq 0 $szam) # ez a for ciklus másik fajta alkalmazása; fentebb
említettük
do
out=$((fib $i))
echo $out
done
exit 0

```