S Z A K D O L G O Z A T

Perjési András

Debrecen
2009

Debreceni Egyetem
Informatikai Kar
Informatikai Rendszerek és Hálózatok tanszék

PowerShell az oktatásban és gyakorlatban

Konzulens:
Dr. Sztrik János
egyetemi tanár,
MTA doktora

Készítette:
Perjési András
informatika tanár szak

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

Az Egyetemen, a korábbi felsőoktatási intézményben, illetve a munkakörnyezetben eltöltött idő úgy gondolom megfelelő lehetőséget adott arra hogy képet alkothassak az Informatikáról, erről a modern és világunkat sűrűn átszövő gyakorlati tudományról. Mégis sokszor kell szembesüljek azzal a ténnyel, hogy bármennyire is próbálom tanulással és gyakorlattal követni ezt a fejlődését, valójában folytonos lemaradásban vagyok. Az élet bármely területén felvetődő problémák hatására nap mint nap születik egy új ötlet, egy új technológia, egy új szabvány ami ezt a gyorsan megújuló szerkezetet működteti, mozgatja. Mindezek megalkotói azok az emberek akiket érdekel a problémák informatikai szemléletű megoldása: a *fejlesztők*é akik elkészítik az eszközt; az *üzemeltetők*é akik működtetik a rendszert; a *felhasználók*é akik a tényleges alkalmazás során élvezik ennek előnyeit. Ez a folyamatos fejlődés, dinamizmus az, ami szakmai fejlődésemben sarkall illetve megerősít abban, hogy érdemes volt ezt a szakmát választanom és továbbképeznem magam benne. Ez a folyamatosan motiváló helyzet azonban némi hátrányokkal is jár. Jártasságunk ellenére ki nem találkozott volna már olyan szituációval mikor egy rutinfeladatnak tűnő probléma megoldása során, csak kerülővel vagy bonyodalmak árán sikerült feloldani a hiba okát – köszönhetően annak hogy épp nem olvastunk utána valaminek, nem tájékozódtunk egy szabvány pontosításáról, és ismereteink nem voltak elég aktuálisak. A helyzet feloldása tehát nem könnyű, de munkánk eredménye annál fontosabb, hiszen ma már alig van olyan területe az életnek ahova nem szökött be az informatika – megkönnyítve, hatékonyabbá téve ezzel a közvetlen környezet életét.

Az előzőek természetesen vonatkoznak az informatikai témájú szakdolgozatokra

is. Véleményem szerint főképp kétféle nehézséget görget elénk a szakdolgozati témaválasztás kérdése illetve a téma kidolgozása. Az egyik: könnyű a témák rengetegében elveszni. Legtöbbször szeretnénk minél több területen ismeretekhez jutni, ami arra az eredményre vezet, hogy egyik témával kapcsolatban sem rendelkezünk szakdolgozat íráshoz alkalmas mélységben megfelelő tapasztalattal, érdeklődéssel, feldolgozó képességgel. A másik véglet, hogy a végzős hallgatók – tisztelet a kivételeknek – ismereteik, idejük, lelkesedésük hiányában kisebb hangvételű „könnyebb" témát választanak. Ekkor az esetek egy részében újra és újra visszatérő témájú szakdolgozatot kaphatunk, máskor esetleg egy jobb feldolgozást ami egy újabb gondolatot indít el egy ismert problémával kapcsolatban. Bármelyik eset is áll fenn a szakdolgozati témaválasztás és a téma feldolgozása mindig sarkalatos része a tanulmányoknak.

Ez járt a fejemben akkor is mikor elhatároztam: diplomamunkámban a Windows PowerShell-t oktatási és alkalmazási aspektusból kívánom tárgyalni. Az informatikai szakterületek rengetege egy dologban bizonyosan egyezik: közvetve vagy közvetlenül mind valamilyen életszerű gyakorlati probléma megoldására törekszenek. Csak néhány általánosat kiragadva, a *kliens programozás* egy programot, programrendszert hoz létre ami közvetlen gyakorlati alkalmazása során válik hasznossá. A *webfejlesztés* kimenete valamilyen böngészőben futtatható – az előzőhöz hasonló feladatokra szánt – alkalmazói eszköz, programokat segítő szolgáltatás vagy publikáló rendszer. Az *adatbázisrendszerek* középpontjában az adatok minél biztonságosabb hatékonyabb kezelése áll. A PowerShell-lel egy olyan informatikai eszköz oktatását és alkalmazását kívánom tárgyalni, mely közvetlenül nem a felhasználóknak, inkább az üzemeltetőknek ad módot a rendszereik rugalmas, gyors és legfőképp egyszerű kezelésére, karbantartására. A szakdolgozattal fő célom dokumentálni és tárgyalni a PowerShell gyakorlati

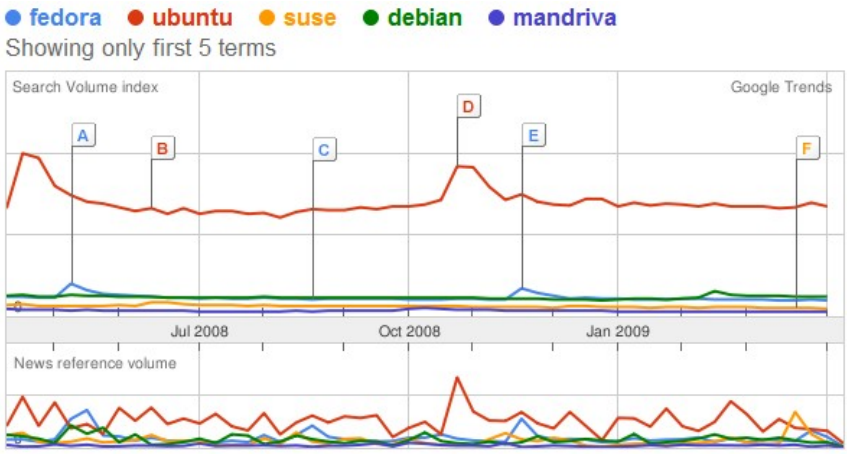
alkalmazásának és tanításának általam jónak gondolt módszereit, elveit.

Elsőként az olyan fontos információkat szeretném tárgyalni melyeket a szkriptnyelvvel kapcsolatban mindenképp célszerű tudni, ilyen a PowerShell általános megközelítése, kialakulása, és az újdonságok, funkciók rövid áttekintése. Ezzel célom úgymond egy általános képet kialakítani. Ez után a második részben konkrétan szeretném részletezni a PowerShell környezet gyakorlatban használható képességeit. Legvégül a szakdolgozat utolsó harmadik részében pedig a úgynevezett *commandlet-ek* tanításához begyakorlásához kívánok egy fajta útmutatót adni, illetve javaslatot tenni ezek alkalmazására. A konkrét példák során Windows Vista SP1 környezet alatt dolgozok majd, ezért mind a leírásokban, mind a képernyőnézetekben, elérési utakban és egyéb rendszer függő beállításoknál az ehhez kapcsolódó konvenciókat fogom követni.

2. A kezdetek

Mindenki számára ismerős fogalom a 'parancssor'. Ha meghalljuk ezt a szót a régi DOS-os felületek jutnak eszünkbe, vagy az SSH terminál a $-végű prompttal. Egy fekete hátterű karakteres felületre gondolunk, ahol meghatározott parancsokkal „vehetjük rá" számítógépünket a kért művelet végrehajtására. Ezek kezdetekben többnyire egyszerű fájlműveletekben nyilvánultak meg a hétköznapi felhasználó számára – nem véletlen a **D**isk **O**perating **S**ystem elnevezés a DOS esetében. Ez akkoriban éppen elég volt arra, hogy a felhasználó saját mappáit, fájljait elérje, módosítsa, kitallózza vagy futtassa. Az idő előrehaladtával azonban felmerült az igény a rendszerek egyre kifinomultabb működtetésére. A helyi rendszertől is egyre többet vártunk, több perifériát kezeljen, hálózaton kommunikáljon. Az operációs rendszerek kezdtek moduláris jelleget ölteni, egyre jobban testre lehetett szabni őket, ezáltal a parancssor is eszközgazdagabb lett, egyre több feladatra lett alkalmas. A kijelzők technológiai fejlődésének és az újabb szoftveres megoldásoknak köszönhetően megkezdődött a grafikus felhasználói felületek térhódítása – ezzel egy időben pedig a karakteres felületek háttérbe szorulása. A parancsokat leváltotta az egérkurzorral történő vezérlés, mellyel a felhasználók sokkal intuitívabb módon használhatták számítógépüket. Mindamellett azonban hogy egyedüli felhasználói felületként szinte ma már csak bizonyos célhardvereken(Cisco router-ek *Cisco IOS operációs rendszere*), vagy alacsony erőforrásokkal rendelkező eszközön(Linksys WRT54GL – *OpenWRT*) használjuk a parancssort, a rendszerüzemeltetés terén továbbra is szükség volt rá. Olyan feladatoknál például ahol kötegelt adminisztrációs műveleteket kellett végrehajtani, a grafikus felületen csak nehézkes és hosszú munka árán érhettük el célunkat, ellentétben a karakteres felületek jól paraméterezhető és iterálható

eszközeinek alkalmazásával. Bár igaz, hogy kezelésük egészen más gondolkodásmenetet és felhasználási stratégiát követel meg, a ma is használt parancssorok tulajdonképpen napjainkban is a rendszerfelügyelet terén tanúsított alkalmazhatóságuknak köszönhetik létjogosultságukat. Mindezt nagyon könnyen beláthatjuk: igen kevés olyan operációs rendszert találunk mely nem rendelkezik valamilyen teljes értékű[1] parancssoros kezelő felülettel – legyen az egy emulált Windows Parancssor(cmd.exe), egy .NET Framework-re épülő PowerShell, vagy egy olyan napjainkban népszerű Linux disztribúció[1][2], mint a Debian alapú Ubuntu, vagy a SUSE Linux.



1. ábra: Google Trends diagramja az öt legnépszerűbb Linux disztribúció térhódításáról

A Microsoft korábbi próbálkozása mellett 2006 novemberére jutott el egy igen rugalmas és korszerű parancssor kiadásáig. Ugyan előtte a Windows 98-cal érkező Windows Scripting Host funkciókban nagy előrelépésnek számított a klasszikus Windows parancssorhoz képest és kezdte felvenni a versenyt a rivális operációs rendszereken futó héjakkal, egy idő után azonban több aspektusból is rossz választásnak bizonyult. [3] Egyrészt sebezhetőségi szempontból kifogásolható volt , mivel a kevés beépített biztonsági lehetőség mellett könnyen lehetett vele programokat, processzeket ütemezni. Másrészt mivel a WSH nem egy önálló

1 Teljes értékű ~ az operációs rendszerrel grafikus felületen elvégezhető bármely művelet kiváltására alkalmas

parancssor hanem csak egy hívható programkönyvtár formájában létezett, a konkrét alkalmazásához valamilyen egyéb szkriptnyelvre(VBScript, Jscript, Python) volt szükség, ezzel pedig egyúttal vált nehézkessé az implementáció, illetve veszett el az egységes szintaxis lehetősége – ami nagyon fontos lesz a szkriptjeink megírásánál, értelmezésénél. Mialatt tehát a Microsoft a 80-as évektől kezdve grafikus felületével nagy sikereket aratott, parancssoros felületének népszerűsége elhanyagolható volt a Linux rendszereken futó parancsértelmezőkhöz képest. Az új parancssor híre, melyet *Monad* névre kereszteltek el és Jeffrey Snover vezető mérnök irányításával fejlesztettek ki, 2003 elején kezdett el terjedni. Snover nyilatkozatai szerint egy alapjaiban új parancssort terveztek, mely feledi a WSH gyermekbetegségeit, s a felhasználói visszajelzések alapján igyekszik majd egyesíteni a hasznos funkciókat és magasabb szintre emelni a Microsoft új parancsértelmezőjének képességeit. A kétségeket véglegesen 2006 november 14-én oszlatták el az új parancssor bejelentésével, melyet a Windows Powershell-nek kereszteltek.

Megjelenés éve	Verzió
2005 június 17.	Monad(Béta)
2005 szeptember 11.	Monad(Béta 2)
2006 január 10.	Monad(Béta 3)
2006 április 25.	Új név: Windows PowerShell 1.0 (RC 1)
2006 szeptember 26.	Windows PowerShell 1.0 (RC[1] 2)
2006 november 11.	Windows PowerShell 1.0 (RTW[2])
2007 január 30.	Windows PowerShell 1.0 for Vista
2007 november 6.	Windows PowerShell 2.0 (CTP[3])
2008 május 2.	Windows PowerShell 2.0 (CTP2)
2008 december 23.	Windows PowerShell 2.0 (CTP3)

1. táblázat: PowerShell történelem[4]

1 Release Candidate
2 Release To Web
3 Customer Technology Preview

Megjelenésekor már lehetett tudni, hogy az új funkcióknak[4] köszönhetően erősen támaszkodni fog rá a Microsoft új kommunikációs platformja a Microsoft Exchange Server 2007. Ennek eredményeképp az Exchange Microsoft Management Console-on keresztül elérhető műveletek mögött valójában PowerShell szkriptek munkálkodtak.

Jelenleg a Windows PowerShell 2.0 CTP3 verziójánál tart. Ez a 1.0-hoz képest is számos új funkcióval rendelkezik, ezekről a szakdolgozat végén ejtek majd néhány szót. Mivel ez a változat azonban még nem számít végleges verziónak, ezért a jelenlegi stabil verziót választottam eszközül munkám során.

A fejlesztők tehát teljesítették ígéretüket és Windows PowerShell-t teljesen új alapokra helyezték. Éppen ezért nem is szeretném az újdonságok tárgyalása során a *cmd.exe*-hez viszonyítani. Megfigyelhetők olyan új tulajdonságok, melyek ugyan a korábbi Windows-os parancssorokban nem voltak meg, viszont a Linux-os környezetben már évek óta bevált eszközként működtek. Ilyen a parancsok egyedi névvel történő azonosítása(*alias*) vagy a csövezési technológia, mely során az egyik parancs kimenetét egy másik parancsnak lehet továbbadni feldolgozásra. Van azonban ezek mellett számos újdonság, amivel legelőször a PowerShell-ben találkozhatunk, ilyen az objektumorientált működés – köszönhetően a .NET Framework 2.0-nek –, a különböző szoftveres komponensek – registry, tanúsítványtár – meghajtóként történő kezelése. Mindezek leírására a szakdolgozat kereteit figyelembe véve a következő fejezetekben vállalkozom.

2. ábra: Windows PowerShell logója és ikonja

4 A hosting funkcióval a powershell felügyeleti szoftverekbe építhető be

3. A gyakorlat

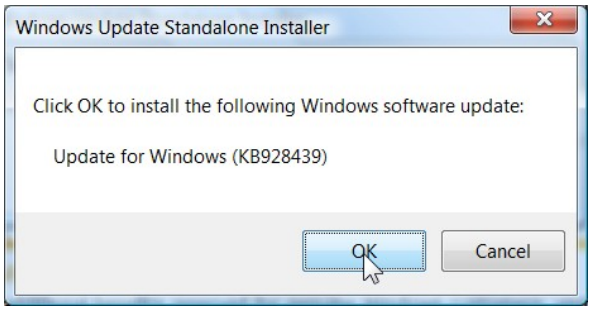
Ahogyan korábban említettem, a grafikus felületekkel szemben a karakteres rendszerhéj egészen más gondolkodást igényel ahhoz, hogy benne hatékonyan tudjunk dolgozni. Míg a grafikus felhasználó felületek esetén a végrehajtáshoz, a feladatok elvégzéséhez az információ ikonok, menük, vizuális elemek formájában mindig könnyen elérhetőek, a parancssor esetén alapértelmezés szerint legfeljebb egy villogó prompttal és az aktuális könyvtár elérési útjával kell beérjük. Ahhoz tehát hogy elérjük a kívánt cél, a fejünkben kell legyen, mely eszközöket, parancsokat használhatjuk fel a megoldáshoz. Az alapértelmezett PowerShell esetén 129 „beépített parancs”[1] áll rendelkezésünkre. Attól függően, hogy milyen szinten vesszük igénybe napi munkánk során a Windows PowerShell nyújtotta lehetőségeket, a „parancsok” közül legalább 10-15 darabot rutinosan kell forgatnunk ahhoz, hogy hatékonyan jussunk eredményre. Ez a szám a gyakoribb használattal exponenciálisan növekedhet, ami sajnos – az emberi memória korlátait ismerve – egy idő után oda vezet, hogy bizonyos kapcsolók, parancsok szintaxisára már nem biztos hogy jól emlékszünk. Szerencsére erről a fejlesztők a környezet kialakítása során nem feledkeztek el és jól kitalált módszerekkel próbálták orvosolni a problémát. Az egyik módszer a konzisztens, következetes működés és szintaxis, a másik pedig a „parancsok” jól kitalált elnevezése. Fontos tehát, hogy először is megismerjük mire képes a parancssoros környezet, majd megtanuljuk a használathoz mindenképp szükséges „alap parancsokat”[2]. Miután pedig ezen ismereteket magabiztosan tudjuk használni, továbbléphetünk a „bonyolultabb” és specifikusabb parancsok irányába. Lássuk tehát mi az amit célszerű tudnunk ha a „türkizkék parancssorral” szeretnénk dolgozni.

1 Nem a klasszikus értelemben vett parancsokról van szó, később ezekre úgynevezett *cmdlet, commandlet* vagy *parancsocska* névvel fogok hivatkozni
2 Ezeket a PowerShell *CoreCommand* néven emlegeti

3.1 Letöltés és használatbavétel

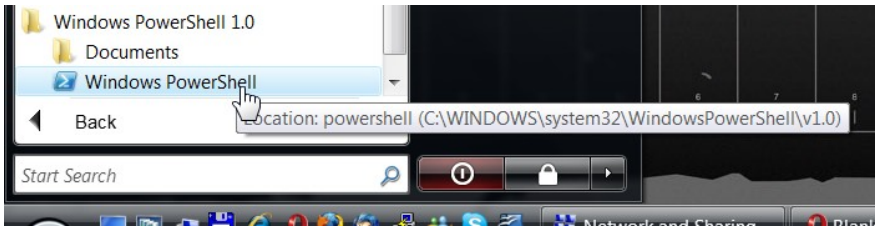
A parancssor ingyenesen letölthető és használható. Ugyanazok a használati feltételek vonatkoznak rá mint az őt futtató környezetre. Mind 32 mind 64 bites architektúrák esetén telepíthető Windows XP SP2/SP3, Windows Server 2003, Windows Vista, Windows 7(béta) operációs rendszerekre. A telepítés szükséges feltétele legalább 2.0 verziójú .NET keretrendszer. Látni fogjuk, hogy tulajdonképpen folyamatosan a .NET osztályaival fogunk kommunikálni, tehát ezen nincs is mit csodálkozni. A felsorolt operációs rendszerek közül sajnos csak a Windows 7(béta) tartalmazza beépített módon a PowerShell-t, a többi platform esetén nekünk kézzel kell telepíteni. A letöltésnél érdemes figyelnünk arra, hogy a megfelelő platformhoz való környezetet töltsük le, máskülönben telepítés előtt hibaüzenettel kell számolnunk. A telepítéshez, akárcsak a parancssor első használatához számos erőforrás és leírás található a PowerShell oldalán.

A telepítés lépéseivel illetve részleteivel ebben a dokumentumban nem szeretnék foglalkozni. Ha valaki az informatikában kicsit is járatos, vagy felkeresi az imént megadott URL-t, a letöltés után kapott *.msi(*Vista esetén *.msu)* állományt futtatva könnyen telepítheti rendszerére a Windows PowerShell megfelelő verzióját. A PowerShell telepítés után egy frissítésként fog megjelenni a *'Vezérlőpult/ Programok Hozzáadása és eltávolítása'* ablak listájában arra az esetre, ha később upgrade céljából a régi verziót el szeretnénk távolítani.



3. ábra: Telepítés során kapott párbeszédablak Vista operációs rendszeren

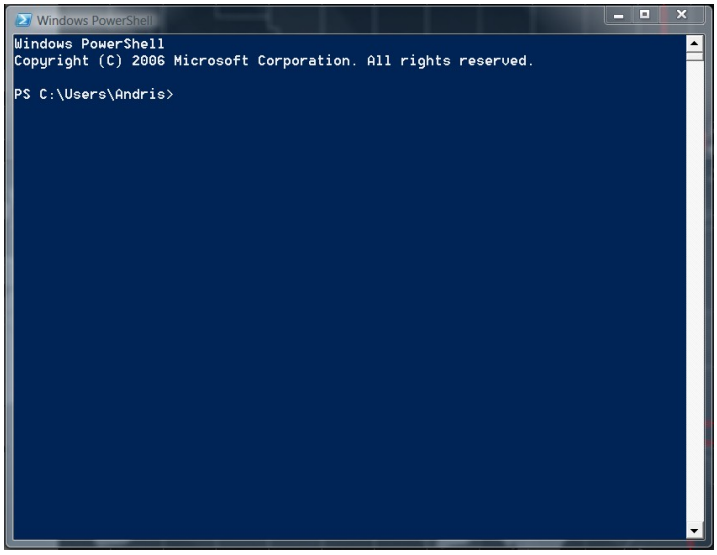
A telepített parancssorunkat a *Start menü/Minden Program/Windows PowerShell* menüje alól érhetjük el legegyszerűbben. A környezetet másrészről a *C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe* útvonalon találhatjuk meg.



4. ábra: Windows PowerShell a Start menüben

3.2 Konzolablak, olyan mint régen

A PowerShell konzolablak működését tekintve szinte teljesen megegyezik a régi *cmd.exe*-vel. A régi módszerhez hasonlóan most is a jobb felső sarokban található '*Ablakvezérlő-menü*[1]*/Tulajdonságok*' menüpontja alól adható meg a buffer és ablak mérettől, a betűfonton, és háttéren keresztül, a szerkesztési módon át szinte minden fontos beállítás. Az első kellemes értelemben szemet szúró különbség az ergonomikus türkizkék színséma. Nyomtatóbarát dokumentum előállítása céljából a későbbi példáimhoz ezt fekete-fehérre változtatom, illetve követhetőség miatt a promptot is módosítom. Tehát először és utoljára ebben a dokumentumban lássuk a PowerShell konzolablakot alapértelmezett „pompájában”:

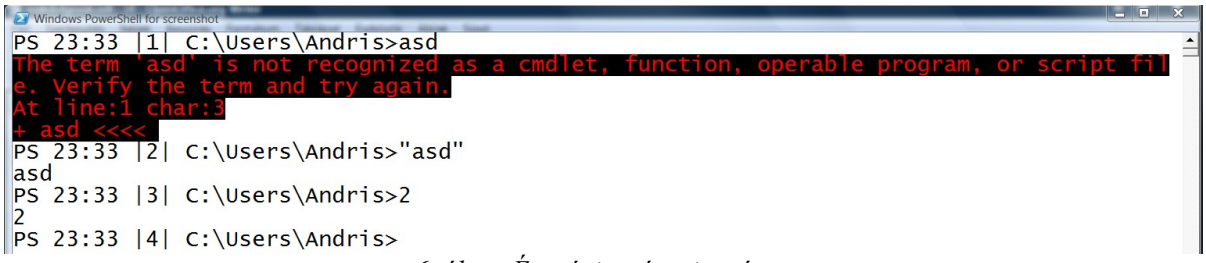


5. ábra: Windows PowerShell konzolablak

1 Alt+Space billentyűkombinációval bármikor elérhető

3.3 Alapvető tudnivalók, ismerkedés a PowerShell-lel

Miután sikerült megbarátkozni az új konzolablakunkkal, felmerül bennünk a kérdés hogy a kék háttér és a kissé megváltozott prompt tényleg valami új dolgot takar-e, és ha igen, vajon elérhetőek-e mindemellett a régi parancsok. Kiadva a jó régi *dir* vagy *cd* parancsokat, örömmel tapasztalhatjuk hogy igenis minden olyan mint régen. Sőt el tudjuk indítani az aktuális könyvtárban vagy bármely *%PATH%* környezeti változó elérési útjain található futtatható állományt, legyen az egy *notepad.exe* vagy egy olyan komoly játékprogram mint az *Aknakereső (minesweeper.exe).* Miután megnyugodtunk, hogy valójában nem is vagyunk nagyon eltévedve, hiszen mindent tudunk amit a régi parancssorral tudtunk. Felmerül bennünk a kérdés, hogy hol vannak az újdonságok. Az alaposabban szemlélődő – és rutinosabb – szakemberek már a *dir* parancsnál is felfigyelhettek egy érdekes eltérésre a parancskimenetben. A kimenet első sora „*Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\...”.* A *Microsoft.PowerShell.Core* sor igen sokat sejtet. A pontokkal elválasztott azonosítók a névterek, osztályok és metódusok elválasztási módjára emlékeztetnek Ha tovább próbálkozunk, arra is igen könnyen rájöhetünk, hogy míg régen a parancssorban elgépelt karakterláncokra a nem túl szép „*rossz parancs vagy fájlnév...*” választ kaptunk, most *cmdlet*-eket(commandlet), *function*-öket(függvény) és *script file* -okat (szkriptfájl vagy héjprogram) említ a PowerShell. Ha pedig számokat vagy karakterláncokat gépelünk be, a parancssor tovább „viccelődik” velünk visszaismételve a begépelt sort.



6. ábra: És mégis más mint régen

Természetesen a korábban tapasztalt jelenségek nem a fejlesztők humorérzékét, sokkal inkább a PowerShell helyes működését tükrözik. Nézzük tehát miről is van itt szó. A PowerShell prompt fogadni tud *commandlet-eket(későbbiek* során: *cmdlet), függvényeket, szkripteket* és további futtatható fájlokat. Mindemellett képes konstans értékek értelmezésére és kezelésére[1], mely származhat akár egy változóból, vagy a sztenderd bemenetről. Mindezt tovább fokozza azzal, hogy a .NET Framework Class Library bármely osztályát vagy abból létrehozott példányt képes kezelni. Így lehetővé válik vele szinte minden adminisztratív feladat végrehajtása. A végrehajtás során a PowerShell alapértelmezés szerint *case-insensitive*, azaz betűnagyság érzéketlen működést mutat.

A következőkben lépésről lépésre megismerkedünk fent említett *cmdlet-ekkel*. Szó lesz arról, hogy mi az a *provider* és mire használhatjuk a PowerShell meghajtóit. Végül a függvényekkel és héjprogramokkal kapcsolatos ismeretekben mélyülünk el.

3.3.1 A cmdlet

A *cmdlet* vagy másként *commandlet* a PowerShell környezetben létező speciális .NET osztály, mellyel a parancssoron belül utasításokat adhatunk ki. Nem lehet tehát sem a *netsh.exe* vagy a *format.exe* programokhoz, sem korábbi parancssorok olyan belső parancsaihoz hasonlítani mint a *dir* vagy a *copy* – bár olyan téren kapcsolható lehet az utóbbi kategóriához, hogy a parancssoros környezeten kívül nem léteznek, így csak a PowerShell segítségével hívhatók meg. Ilyen *cmdlet-ekkel* listázhatunk egy könyvtárt, lekérdezhetjük az eseménynaplót, szűrhetjük a kimenetünket vagy akár objektumokat hozhatunk létre. Minden *cmdlet* egy *<ige>"-"<főnév>* alakban van megadva. A hatékonyságot továbbfokozza

1 Valójában a parancsok kimenetei is értékek melyeket a powershell – ha másképp nem rendelkezünk – egy kiíró parancsnak ad tovább

az, hogy egy olyan *cmdlet* ami könyvtárakat képes listázni, általában képes lesz majd arra is, hogy a tanúsítványokat megjelenítse a *tanúsítványtárunkban*, vagy *registry* kulcsok felsorolását adja. A *cmdlet-ek* megalkotásakor tehát nagyban törekedtek arra, hogy az általa hordozott funkció minél általánosabb körben, minél konvencionálisabb módon alkalmazható legyen.

A különböző *cmdlet-ek* általában valamilyen bemenet alapján végzik dolgukat. Ezt a bemenetet két módszerrel tudják fogadni. Az első módszer szerint a végrehajtáshoz szükséges információkat(elérési utat, számértéket, vagy valamilyen azonosítót) kötőjeles kapcsolók segítségével adjuk meg. A *cmdlet-től* függően az egyes kapcsolókat nem mindig kötelező feltüntetni. Ekkor a bemenetekre érkező adatokat azok sorrendje alapján fogja a *cmdlet* kiértékelni és a megfelelő kapcsolóhoz kötni[1]. A kapcsolók esetében öt tulajdonságot tudunk meghatározni: 1) kötelező-e a használata(nem a kiírása!); 2) csak névvel vagy sorrend alapján is azonosítható-e; 3) van-e alapértelmezett értéke(például könyvtár listázásnál aktuális elérési út); 4) tudja-e fogadni a csővezetéken érkező bemenetet; 5) értelmezi-e a *jocker* vagy *wildcard* karaktereket. A bemenet feldolgozásának másik módszere szerint az adatokat egy korábbi parancs eredményeként kapjuk. A *cmdlet* ilyenkor mindig a csővezeték karakter('|') jobb oldalán áll, és a végrehajtás mikéntje az előbb említett kapcsoló tulajdonságok, illetve a bemenetre érkező érték(ek) alapján implicit módon dől el.

A *cmdlet-ek* rendelkeznek egy úgynevezett általános kapcsoló készlettel – a PowerShell ezt *Common Parameters* néven említi. Bizonyos feltételek szerint ezeket a kapcsolókat vagy részben, vagy egészben bármely *cmdlet* használata során alkalmazhatjuk. A sorból eszerint a '-*Whatif*' és a '-*Confirm*' kapcsolók fognak

1 A kapcsolóknak lesz egy speciális típusa mely nem vár értéket, egyedül csak egy opció bekapcsolását jelzi, innen is az elnevezés 'switch parameter'

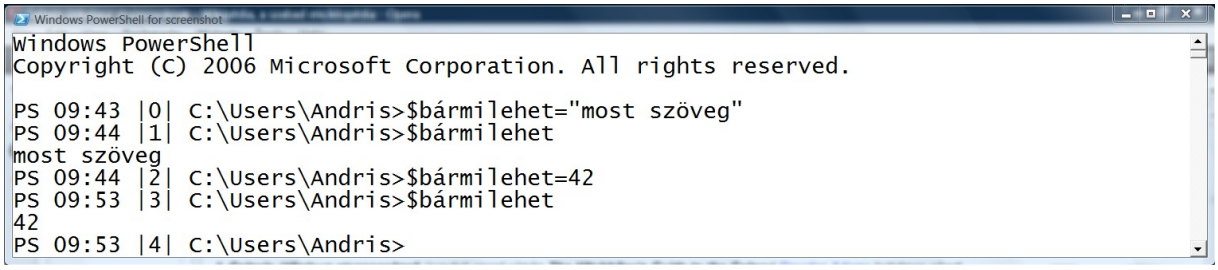
kilógni. Ezeket a kapcsolókat ugyanis csak olyan *cmdlet-eknél* használhatjuk, melyek megváltoztathatják a rendszer állapotát. Lássuk hogy az egyes kapcsolókat milyen esetekben alkalmazzuk.

- *Verbose*: amennyiben a *cmdlet* generál nyomkövetési adatokat, a kapcsoló megadásával ezt be lehet kapcsolni
- *Debug:* amennyiben a *cmdlet* generál programozási szintű nyomkövetési adatokat, a kapcsoló megadásával ezt be lehet kapcsolni.
- *ErrorAction*: a végrehajtás során keletkező hiba események alapértelmezett válaszát adhatjuk meg. A kapcsolónak a *Continue*(alapértelmezés), *Stop*, *Silently, Continue*, *Inquire* felsorolás valamely elemét várja.
- *ErrorVariable*: megadható, hogy a hibaadatokat mely változóban szeretnénk rögzíteni
- *OutVariable*: megadható, hogy a végrehajtás során keletkező kimenetet milyen változóban szeretném tárolni.
- *OutBuffer*: egész számként megadható, hogy hány kimeneti objektum esetén adja át a végrehajtás eredményét a csővezeték soron következő *cmdlet*-ének
- *WhatIf*: a kapcsoló használata esetén a *cmdlet* szövegesen kiírja, hogy mi a várható eredménye a parancsvégrehajtásnak(csak olyan *cmdlet*-ek esetén mikor az változást idéz elő)
- *Confirm*: a kapcsoló használata esetén a *cmdlet* szövegesen kiírja, hogy mi a várható eredménye a parancsvégrehajtásnak, és megerősítést vár a végrehajtáshoz.(csak olyan *cmdlet*-ek esetén mikor az változást idéz elő)

3.3.2 Változók

Gyakori hogy munkánk során a *cmdlet-ek* kimenetét, a paraméternek szánt értékeket vagy valamilyen kifejezés operandusait tárolni szeretnénk. Az értékek tárolására ilyenkor változókat használunk. Minden változónak lesz egy *azonosítója* mellyel hivatkozhatunk rá, egy *értéke* melyet benne tárolunk és egy *típusa*, mely

meghatározza a változóval végezhető műveletek körét. A PowerShell változói alapértelmezés szerint nem szigorúan típusosak. Vagyis egy változó ami egyik pillanatban egy karakterláncot tárol, másik pillanatban akár egy fájl objektumot is gond nélkül értékül kaphat. Előre deklarálásuk nem kötelező, amennyiben viszont szeretnénk, van módunk arra, hogy meghatározzuk a típusát. Mindezt akkor kell megtennünk, mikor először adunk neki értéket. Innentől fogva az adott változóban csak a deklarálás szerinti típus értékeit tárolhatjuk. Ha mégis más típusú konstanst próbálunk neki értékül adni, a PowerShell hibát fog jelezni. PowerShell-ben a változó típusát a változó neve előtt szögletes zárójelek között feltüntetve kell megadni. Természetesen típusként használható bármely .NET Framework-ben példányosítható osztály minősített névvel, vagy megadhatjuk rövid alakban néhány gyakoribb típus nevét – ezeket a függelék Típusok rövidnévvel és minősített névvel feliratú táblázatában találhatjuk meg. A változók azonosítója minden esetben dollárjellel('$') kezdődik és tartalmazhat kis-nagy betűt, számot illetve egyéb speciális jeleket tetszőleges sorrendben. A PowerShell betűként értelmezi a helyi nyelvi karaktereket is – a biztos átjárhatóság miatt szkriptjeinknél javasolt ennek elkerülése.

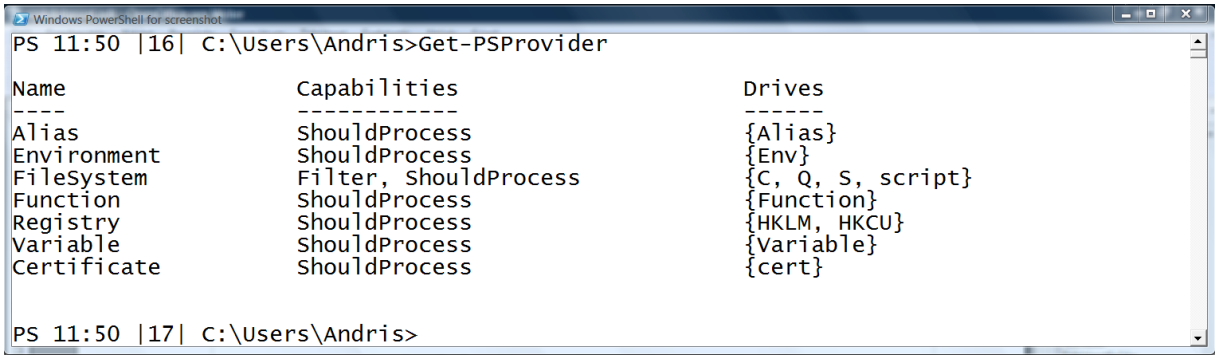


7. ábra: Nem szigorúan típusos változók

A PowerShell a speciális karakterek használatát valószínűleg úgynevezett automatikus és környezetfüggő változói miatt tartja érvényes felhasználható karakternek a változó azonosítókban. Ennek ellenére helyi nyelvi karakterekhez hasonlóan, a speciális karakterek ilyen módú felhasználását csak különleges esetekben tartom célszerűnek.

3.3.3 *Provider-ek* és meghajtók

Természetesnek hat, hogy mikor legújabb parancssorunkat elindítjuk benne azonnal kezelni tudjuk fájlrendszerünket, tallózhatunk a mappák között, fájlokat érhetünk el. Mivel PowerShell-ről van szó, és tudjuk hogy mint olyan igen erősen támaszkodik .NET Framework keretrendszerre, ezért azon sem csodálkoznunk sokat, hogy meghajtóinkat is a .NET lehetőségeinek kiaknázásval érjük el. Az viszont érdekesebben hangzik, hogy a PowerShell-ben nem csak valamilyen fájlrendszer gyökere lehet meghajtó. Hogyan lehetséges ez? A válasz a PowerShell *provider-ekben* vagy szolgáltatókban rejlik. A *cmdlet-ekhez* hasonlóan a *provider-ek* is lefordított .NET osztályok, ezek feladata azonban az, hogy egységes és legfőképp egyszerű módon férhessünk hozzá a különböző adattárolóinkhoz. A *cmdlet-einket* és *provider-einket* úgynevezett PowerShell *SpanIn-ek* fogják össze, ezek segítségével épülnek be a környezetbe és válnak használhatóvá. Léteznek alapértelmezett *SnapIn-ek*, de van lehetőségünk saját magunknak hozzáadni ilyen modulokat PowerShell környezetünkhöz, bővítve ezzel a parancssor és szkriptjeink felhasználási körét. A *SnapIn-ek* hozzáadásakor nem árt szem előtt tartanunk azt, hogy jelenleg teljes eltávolításuk nincs megoldva. Ez azt jelenti, hogy feltelepítés után ugyan a parancssorban – vagy szkriptjeinkben – van módunk kikapcsolni őket, de legközelebbi indításkor ez az állapot elveszik és újra az összes *SnapIn* betöltődik – a benne található összes *provider-rel* és *cmdlet-tel.* Mint látható alapértelmezés szerint hét darab szolgáltató áll rendelkezésünkre. A három oszlopba rendezett



8. ábra: PowerShell szolgáltatók listája

kimenet megadja a szolgáltató nevét, képességeit és azon meghajtókat, melyek épp használják az adott *provider-t*. A képességeknél a *Filter* tulajdonság határozza meg hogy *provider* környezetében futtatott *cmdlet-ek* szűrhetik a kimenetüket – már amennyiben van nekik *'-filter'* kapcsolójuk. A *Shouldprocess* tulajdonság lehetővé teszi hogy a parancsocskák végrehajtásuk előtt – amennyiben ezt a '-*WhatIf*' vagy '-*Confirm'* kapcsolókkal jelezzük – információkkal szolgáljanak a végrehajtás utáni állapotról, illetve megerősítést kérjenek a végrehajtáshoz, vagy késleltessék a futtatást. A legutóbbi képernyőnézeten láthattuk hogy a *provider-ekre* épülő meghajtók nevekkel rendelkeznek, ez a név – mint látható – több karakterből is állhat. Tekintsük át a következőkben hogy mely provider-ekhez milyen meghajtók tartoznak alapértelmezés szerint:

- A *FileSystem(*fájlrendszer) *provider* meghajtói rendszerint – de nem szükségszerűen – egy betűből állnak, melyek a merevlemez meghajtóbetűjeleit jelölik. Segítségével végrehajtható minden fájlt vagy könyvtárat érintő művelet.
- A *Registry* meghajtói, a *HKLM* és *HKCU*, rendre a *HKEY_LOCAL_MACHINE* és a *HKEY_CURRENT_USER* rendszeradatbázis kulcsaira mutatnak. A provider lehetővé teszi számunkra, hogy böngésszük és szerkesszük az adatbázis kulcsait és bejegyzéseit.
- A *Certificate*(tanúsítvány) egyetlen meghajtója *Cert* néven hivatkozható. Ennek segítségével tallózhatóvá válnak a tanúsítványtárolóink és manipulálhatók a tanúsítványaink. Ennek különös hasznát vesszük szkriptjeink aláírásakor.
- A *Variable*(változó) *provider* meghajtója már kevésbé kézzelfogható. A *'Variable:'* meghajtót kiválasztva listázhatjuk és manipulálhatjuk a parancssorban létrehozott változókat.
- Az *Environment*(környezet) meghajtója *'Env:'* néven hivatkozható, itt a rendszerünk környezeti változóihoz férhetünk hozzá. Bár végleges

környezeti változó létrehozása viszonylag sok körbejárást és egy .NET osztályt igényel majd.

- A *Function*(függvény) *provider* a használható függvényeket tárolja és meghajtója ugyanezzel a névvel érhető el.
- Az *Alias*(álnév) szolgáltatóval a parancsainkhoz rendelt álneveket tudjuk kezelni – a hozzárendelt egyetlen meghajtó neve itt is megegyezik a *provider* nevével.

A PowerShell meghajtóit *meghajtónév:'* formában hivatkozhatjuk. A hierarchikus szerkezetű meghajtók – *FileSystem, Registry, Certificate* –, a hierarchiában alul/felül rendeltségi viszonyt az elérési útban visszafelé per-jellel('\') jelölik.

3.3.4 OOP

Bár az Objektum Orientált Programozási paradigma nem csak a PowerShell-lel kapcsolatba hozható fogalom, azonban most érkezett el annak az ideje, hogy néhány szóban foglalkozzunk ezzel a fontos témával. Tudjuk hogy a PowerShell *cmdlet-ekkel* dolgozik, és a tároló komponensek eléréséhez *provider-eket* használunk. Ha csak ezeket tekintjük, tudjuk hogy mindkettőt objektum-osztályokkal valósítjuk meg. Konkrétabb példa azonban a változók esete, ahol is szinte akármilyen .NET osztály felhasználásával tárolhatunk objektumokat. Mindez nem sokat érne akkor, ha mindig mi kézzel adnánk meg a változók értékeit, mindig mi magunk példányosítanánk. Szerencsére ez nem is így van. Ugyanis valahányszor végrehajtunk egy *cmdlet-et*, a visszaadott kimenet egy objektum lesz. Ez az jelenti hogy használhatjuk az eredmény objektum minden elérhető metódusát, kiolvashatjuk mezőit, amivel nagy tehertől szabadulunk meg. Gondoljunk csak arra, hogy egy könyvtárlistázás során konzolra kapott szöveges kimenet karakter alapú feldolgozásától, mennyivel egyszerűbb az a módszer, ha a visszakapott fájl objektumok létrehozási dátumát az objektum metódusaival olvasom ki. Nem is beszélve arról, hogy a fájlok sokszor tulajdonossal, hozzáférési szabályokkal és egyéb olyan

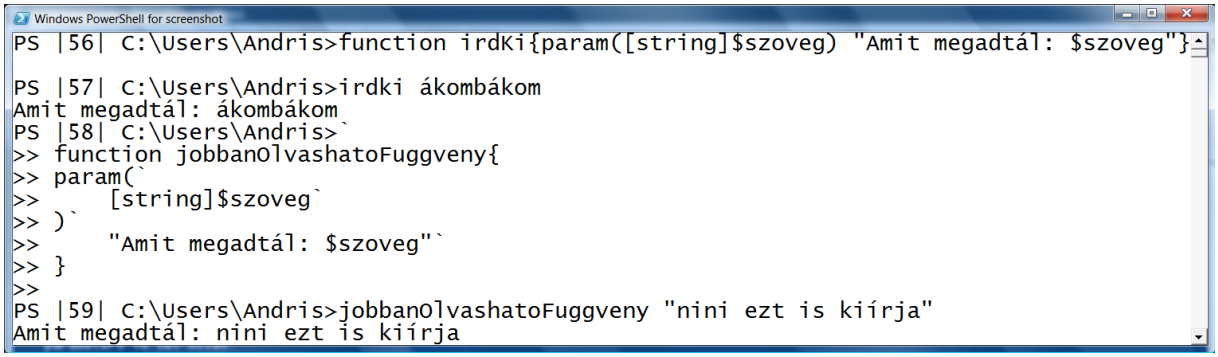
tulajdonságokkal is rendelkeznek, melyeket egyébként a parancssor nem szokott megjeleníteni. Ezzel az egyszerű példával is könnyen belátható hogy a PowerShell egyik legnagyobb erőssége az, hogy minden esetben objektumokkal dolgozik. Teszi ezt akkor is, mikor a konzolablakban megjelenít valamit, ekkor ugyanis a *cmdlet* kimeneteként kapott objektumokat alapértelmezés szerint a *Write-Output cmdlet-be* irányítja, ami aztán szöveges formában megjeleníti az eredményt. Az eredményül kapott objektumok mezőire metódusaira a már jól ismert módon '*objektum.metódus()*'[1], míg tulajdonságai esetén '*objektum.tulajdonság*' formában hivatkozhatunk. Mi a helyzet akkor ha valamely metódus csak osztály referenciával hivatkozható? Ekkor egyszerűen szögletes zárójelek között kell jelölni az osztályt, majd két kettőspont(::) után hivatkozhatunk is az adott osztálymetódusra vagy osztálymezőre, valahogy így: *[Névtér.osztály]::osztálymetódus().*

3.3.5 Függvények

Eddig nem foglalkoztunk összetettebb problémák megoldásával. A PowerShell-lel dolgozó szakemberek idejének nagy részét viszont ilyen feladatok teszik ki. Gyakori hogy a kidolgozott kódot többször is újra kell hasznosítani, ennek egyik módja a PowerShell utasításaink függvényként való implementálása. A függvények, ugyanúgy ahogy azt a *cmdlet*-ek tették, várhatnak paramétereket melyek alapján aztán elvégezhetnek valamilyen feladatot és/vagy visszaadhatnak valamilyen eredményt. A paramétereket egyszerűen a függvény hívásakor, a függvény neve után szóközökkel elválasztva adhatjuk meg. Amennyiben több paraméterrel is dolgozunk és még egyértelműbbé szeretnénk tenni, hogy mely érték mely paraméterhez tartozik, az érték előtt '*-paraméternév*' formában jelezhetjük ezt – hasonlóan mint a *cmdlet-eknél.* Függvényt definiálni a *function függvénynév*

1 Fontos hogy a metódus mögött kitegyük a zárójelpárt másképp a metódust mint objektumot kaptuk eredményül

{param(paraméter1, paraméter2...) függvény kódja} formában tudunk. Erre egyszerű példát a következő oldali képernyőnézeten látunk.



9. ábra: Példa függvény definícióra sor folytonosan és backtick escape karakterrel több sorban

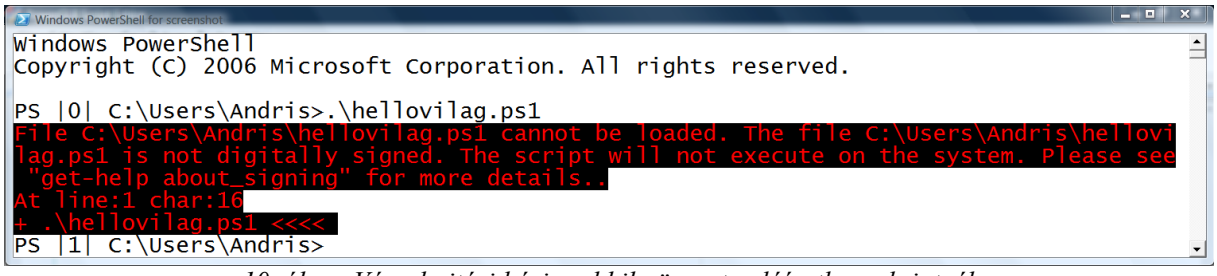
Az *irdKi* függvény paraméternek kap egy szöveget, amit változatlan formában ki is ír. Látható, hogy a paraméternek én típust is definiáltam. A definiált paraméterek használata nem kötelező, nélkülük a függvény belsejéből az *$args[]* tömb elemeire hivatkozva is tudjuk fogadni a paraméter(ek)nek szánt bemenetet. A függvények ilyen jellegű sor folytonos definíciója nem túl könnyen követhető, ezért előfordul hogy parancsunkban sortöréseket hozunk létre. A sortörésekben a parancsaink végrehajtása során alkalmazott ENTER-t használjuk egy backthick(`) karakterrel hatástalanítva[1]. Bár függvény szerkesztésénél persze továbbra sem ideális ilyen környezet, többsoros cmdlet-láncnál jól jöhet ez a módszer. Minden olyan esetben tehát amikor egy speciális karaktert „hatásától megfosztva” csupán sima karakterként szeretnénk értelmezni, ezt a visszafelé aposztróf jelet kell előtte feltüntetni.A függvények parancssori definiálása ideiglenes hatású, azaz ha bezárjuk a konzolablakot, következő indításkor a függvény nem használható fel újra. Erre nyújtanak megoldást a PowerShell profilok. Ezeket a speciális szkriptfájlokat ugyanis minden indításkor automatikusan lefuttatja a PowerShell. Ahhoz hogy a szkriptfájlokról többet megtudhassunk tekintsük a következő fejezetet.

1 Az ilyen karaktereket escape karaktereknek nevezzük

3.3.6 Szkriptfájlok

A bonyolult programkódok rögzítésének másik módja, mikor az utasításainkat szkriptfájlokban írjuk meg. A szkriptfájlok *.ps1* kiterjesztésű sima szöveges (*plaintext)* állományok, melyek PowerShell utasításokat tárolnak. A függvényekkel szemben rendelkeznek azon jó tulajdonságokkal, hogy hordozhatóak és tartósak – bár a hordozhatósághoz egyéb negatív tulajdonságok is társulhatnak. Az által, hogy a PowerShell az operációs rendszerünket szinte teljes mértékben ki tudja használni, a szkriptek számítógépünket támadás célpontjává is tehetik. Mivel ez nem egy új keletű probléma, erre a fejlesztők is felkészültek. A probléma megoldására a PowerShell-ben végrehajtási házirendeket definiáltak, illetve lehetővé tették a szkriptek aláírását, valamint az aláírások ellenőrzését. Kedvenc parancssorunkban található végrehajtási házirenddel háromféle biztonsági szint állítható be:

- *Restricted*: egyáltalán nem futtathatóak szkriptek
- *Allsigned*: kizárólag csak digitálisan aláírt és megbízható szkriptek futtathatók
- *RemoteSigned*: a helyi készítésű szkriptek digitális aláírás nélkül is futtathatók, de a webről letöltött, hálózati meghajtóról indított szkriptek csak érvényes aláírással indíthatók
- *UnRestricted*: bármely szkript bármikor futtatható



10. ábra: Végrehajtási házirend hibaüzenete aláíratlan szkriptnél

Az alapértelmezett beállítás minden telepítés után az *AllSigned* opció, ezért a PowerShell-lel újonnan ismerkedő felhasználók első – aláíratlan – szkriptjük futtatásakor gyakran az előző oldali üzenettel szembesülnek.

Ilyenkor két dolgot tehetünk, az egyik módszer egy kódaláíró tanúsítvány beszerzése, majd szkriptünk aláírása az újonnan kapott tanúsítvánnyal[1]. Mivel azonban a kezdő felhasználó nem feltétlenül rendelkezik az ehhez szükséges háttér információkkal, vagy esetleg épp a tanúsítvány beszerzése jelent neki problémát, célszerű ha egy „engedékenyebb" - de nem túl megengedő – házirendet állítunk be. Ilyen esetekre kompatibilitási szempontból megfelelő – és mellette kellően biztonságos – házirend a *RemoteSigned* beállítás.

A végrehajtási házirendeket a *Set-ExecutionPolicy cmdlet-tel* rendszergazda jogosultság birtokában változtathatjuk meg. Ezt figyelembe véve a sebezhetőség Windows XP operációs rendszereken továbbra is fenn áll, legtöbb helyen a mindennapok során használt helyi fiókunk egyben a Rendszergazdák csoportnak is tagja. Emiatt a végrehajtási házirend akár egy sima .bat állományból is módosítható. Windows Vista-n és az őt követő rendszereken szerencsére ez már nem ilyen egyszerű.

Mivel gyakori, hogy szkriptek írásába komolyabb feladatok esetén vágunk bele, valamint a megoldó algoritmust azért írjuk meg egyszer, hogy ezzel a későbbiek során ne kelljen még egyszer vesződni, a szkriptek írásánál kell felhívnom a figyelmet arra, hogy az utasításaink „érthetősége" legalább olyan fontos szempont mint a megírt kód hatékonysága. Ahhoz, hogy ezt elérjük minimális körbejárásra és minimális plusz időre van szükség, amivel a jövőben viszont órákat spórolhatunk meg, mikor kölcsönadjuk a szkriptet valamelyik kollégánknak, vagy épp magunk javítjuk tovább a programlogikát.

Az elkészített szkripteket a teljes elérési úttal hivatkozva tudjuk futtatni. Amennyiben elég mélyen vagyunk a fájlrendszer hierarchiában, vagy egyszerűen nem szeretnénk sokat gépelni és a héjprogram az aktuális könyvtárban található, úgy az aktuális elérési utat a „.\" formában is jelezhetjük

3.3.7 Cmdlet-ek konkrétan

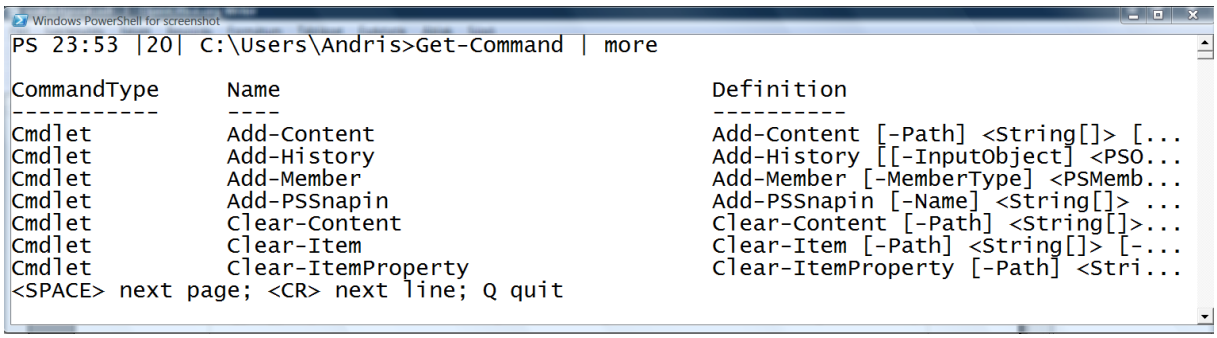
A korábbi fejezetekben már volt szó a *cmdlet*-ekről, akkor viszont nem

1 Későbbi fejezetekben részletesen tárgyaljuk

tárgyaltuk konkrét alkalmazásukat. Ezt a hiányt szeretném most pótolni azzal, hogy felsorolom és példákkal magyarázatot adok alkalmazásukra. A példák során sokszor hosszas parancskimenetet kapok, nem szeretném viszont a dokumentum folytonosságát megszakítani, ezért előfordul, hogy a dokumentum végén található függelék ábráira fogok hivatkozni. A PowerShell az alap *cmdlet-eket* egy bizonyos *CoreCommands* csoportba sorolja. Ezek közül a következőkben nézzük meg, melyek azok a *cmdlet-ek* melyekre gyakrabban lehet szükségünk.

3.3.7.1 Get-Command

Először is fontos tudnunk, hogy milyen *cmdlet-eket* használhatunk egyáltalán. A környezetben használható *commandlet-ek,* illetve – tágabb értelemben használva – bármely közvetlenül futtatható parancs listázását a '*Get-Command*' *cmdlet* valósítja meg. A PowerShell 129 darab cmdlet-et tartalmaz, így a parancsot kiadva egy terjengős lista szalad végig a szemünk előtt. Ha viszont a parancs mögé fűzzük egy csővezeték jel('|') után a más parancssorokból is ismert *more* terminál-lapozó nevét – ami mellékesen PowerShell-ben egy függvény lesz –, akkor a kapott kimenet soronként, vagy akár oldalanként lapozható lesz.



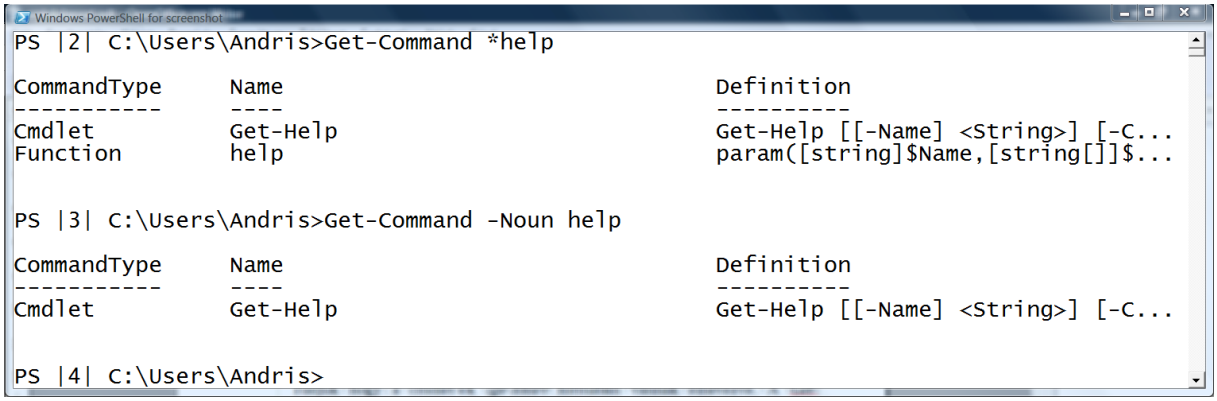
11. ábra: 'Get-Command' kimenete 'more' terminállapozó alkalmazásával

A lista jól áttekinthető, látszik hogy az elemek típusa *cmdlet*, megfigyelhető a ige-főnév elnevezési konvenció az azonosítókban, illetve kapunk egy szintaxis információkkal szolgáló *'Definition'* oszlopot – amiből egyelőre[1] túl sokat nem

1 Formázó cmdlet-eknél később orvosoljuk ezt a problémát

látunk. Az előző lista tehát a környezetben használható összes *cmdlet-et* visszaadta.

Tudjuk hogy a *cmdlet*-ek *ige-főnév* formában vannak elnevezve. A *Get-Command cmdlet* akkor nyújt igazán sok segítséget, ha *cmdlet*-et keresünk valamihez. Ilyenkor vagy *wildcard* karakterekkel adjuk meg a keresett *cmdlet* nevét, vagy használjuk a *-Noun* vagy a *-Verb* kapcsolókat melyekkel kifejezetten a *cmdlet* nevének főnév vagy ige része szerint tudunk keresni.



12. ábra: Help kulcsszót tartalmazó cmdlet-ek keresése Get-Command-dal

3.3.7.2 Get-Help

Most már tudjuk hogy milyen *cmdlet-eket* állnak rendelkezésünkre, használatukról azonban nem sok információ áll rendelkezésre. Ennek megállapításához lesz hasznunkra a következő *Get-Help* cmdlet. A *Get-Help cmdlet-et* pontosan akkor forgatjuk gyakrabban, ha valamiről információkat szeretnénk megtudni. Ha a *Get-Help* parancsot paraméter nélkül futtatjuk le, akkor a *Get-Help* önmaga használatáról ad információkat. Mindebből kiderül, hogy a *Get-Help* alkalmas parancsocskákkal kapcsolatos információk, illetve help-fájlok tartalmának megjelenítésére, valamint teljes vagy szűrt listát kérhetünk vele arról, hogy milyen témákkal, *cmdlet*-ekkel kapcsolatban rendelkezik információkkal[1].

Cmdlet-ek esetén az eredményül kapott kimenet a használati leírás mellett konkrét alkalmazási példát és szintaktikai leírást is tartalmaz. A parancs kimenet

1 Ezen információk egy részét egyébként – angol nyelvű telepítés esetén a '*C:\Windows\System32\WindowsPowerShell\v1.0\en-US*'mappában található fájlokból veszi.

alapértelmezés szerint gyors használatra alkalmas rövidebb kimenetet produkál, beszédesebb segítséget a *-detailed* vagy a *-full* kapcsolókkal érhetjük el. A súgótémák mind *'about_'* kulcsszóval kezdődnek így ha csak a lehetséges súgótémákat szeretnénk megkapni akkor a *Get-Help about_** parancsot kell kiadni.

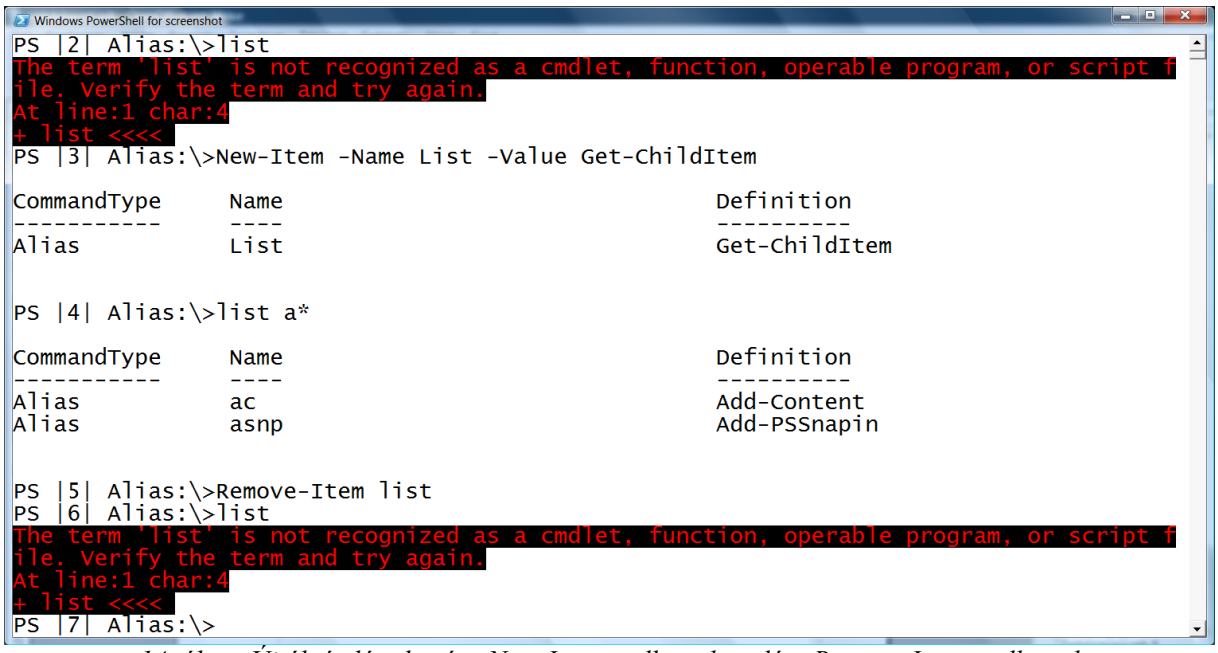
3.3.7.3 Get-Childitem

Bármilyen meghajtót is használunk, legyen az egy fájlrendszer vagy registry típusú, az elemek listázását a *Get-ChildItem* cmdlet-tel tudjuk elérni. A *cmdlet wildcard* karakterekkel a kimenet közvetlen szűrésére.

13. ábra: HKEY_CURRENT_USER kulcs kitallózása

3.3.7.4 Set-Location

Ha szeretnénk meghajtót váltani, a fájlrendszer hierarchiában mozogni, vagy ki akarunk választani egy új tároló objektumot, akkor a *Set-Location cmdlet*-et alkalmazzuk. A fenti képernyőnézeten az látható ahogyan a *Set-Location cmdlet*-tel kiválasztjuk a HKEY_CURRENT_USER *registry* kulcsot mint meghajtót.



14. ábra: Új álnév létrehozása New-Item cmdlet-tel, törlése Remove-Item cmdlet-tel

3.3.7.5 New-Item

Az aktuális környezet *provider-étől* függően *New-Item* cmdlet-tel a névtér egy új

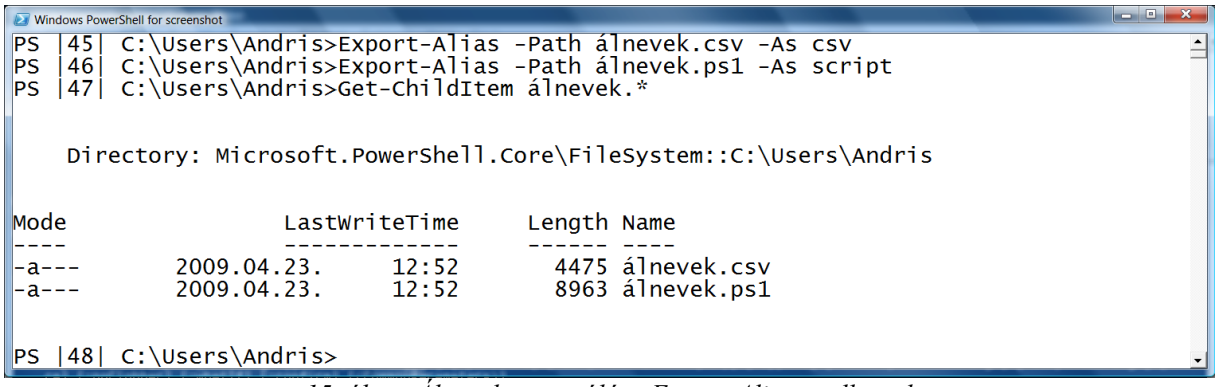
elemét hozhatjuk létre. Amennyiben épp fájlrendszer jellegű tárolóban vagyunk, fájlt vagy mappát kreálhatunk. Ha a *registry-ben* vagyunk, ott egy új kulcs képezhető vele, ha az *alias*: meghajtó az aktuális környezet, ott új álnevet hozhatunk létre(lásd: előző oldali képernyőnézet). Használata mindig *provider*-től függ. Alkalmazásához a fájlrendszerrel kapcsolatos parancsokat szemléltető képernyőnézeten is láthatunk példát.

3.3.7.6 Remove-Item, Copy-Item, Move-Item, Rename-Item

A PowerShell-ben gyakran hajtunk végre valamilyen objektum manipulációs parancsot. Az „objektum" szóhasználat szándékos volt, hiszen ugyanúgy mint a *Get-ChildItem* esetében, a címben szereplő három *cmdlet* is meghajtótól „függetlenül" használható. A '*Remove-Item'* parancsocskát törlésre, a *'Copy-Item'* és *'Move-Item'* parancsocskát másolásra illetve mozgatásra használjuk, a '*Rename-Item*' pedig átnevezéshez jó. A fenti képernyőnézeten a *List* alias-t töröltük vele. Előfordulhat létrehozás, áthelyezés esetén is, de jellemzően törlésnél találkozhatunk olyan helyzettel, hogy egy adott elem nem törölhető – mert mondjuk írásvédett. Erre az esetre az ilyen jellegű *cmdlet-ek* rendelkeznek egy *-Force* kapcsolóval, amit a *cmdlet* kiadásakor a végrehajtási szándék megerősítéseképpen fel kell tüntetni.

3.3.7.7 New-Alias, Set-Alias, Get-Alias, Export-Alias, Import-Alias

Ha valami általánosan használható, az általános feladatoknál, gyors egyszerű műveleteknél ideális eszköz lehet. Kifinomultabb műveletekhez viszont már nem biztos hogy elég szolgáltatást tud nyújtani. Így van ez az álnevek kezelésével is. Az előbb tárgyalt *cmdlet*-ek ugyan alkalmasak álnevek kezelésére, viszont nehézkes lehet az, hogy folyton hivatkozni kell a megfelelő meghajtóra – ez kötegelt feladatoknál pedig hatékonyság szempontjából sem elhanyagolható, hiszen ez mindig plusz egy művelet.

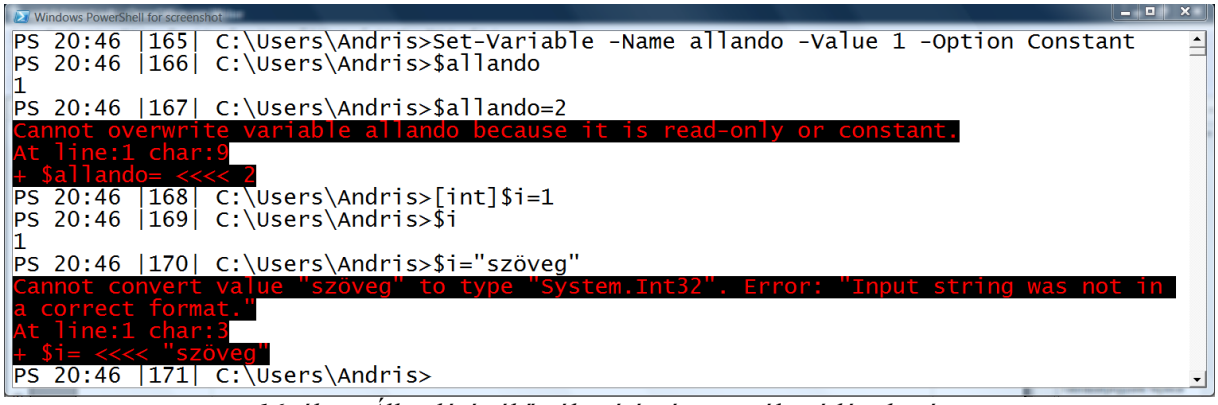


15. ábra: Álnevek exportálása Export-Alias cmdlet-tel

Mindemellett az álnevek rendelkeznek egy '*Description*' tulajdonsággal, amit viszont a *New-Item* már nem tud elérni. Ilyenkor használható helyette – bármely környezetben is legyünk – a *New-Alias* cmdlet. Segítségével ugyanis az *alias* létrehozásakor *-Description* kapcsolóval egy karakterlánc formájában meghatározható az álnév funkciója. Más művelet mellett ugyanezt a feladatot képes a *Set-Alias* cmdlet is végrehajtani annyi különbséggel, hogy ő már egy létező alias-t fog módosítani. Az *Export-Alias*, *Import-Alias cmdlet-ekkel* pedig szkript vagy *csv* formátumban lehet ki illetve beexportálni álneveket.

3.3.7.8 New-Variable, Get-Variable, Clear-Variable, Remove-Variable, Set-Variable

Hasonlóan az álnevekhez, a változók is saját *cmdlet*-készlettel kezelhetők – amellett hogy a „*New-Item* és társai” is tudják őket kezelni. A változók létrehozására használható a *'New-Variable' cmdlet*. Ekkor a változó nevét(dollár nélkül) *'Name'* kapcsoló mögött, az értékét pedig *'-Value'* kapcsolóval tudjuk megadni. Ugyanezzel a *cmdlet-tel* hozhatunk létre „állandó értékű” úgynevezett konstans változót. Ilyenkor az *'-Option'* kapcsoló mögött *'Constant'* felsorolástípusú értékkel jelezzük, hogy a változó értékét nem lehet megváltoztatni.



16. ábra: Állandó-értékű változó és típusos változó létrehozása

A *Set-Variable* egy már létező változó értékét, tulajdonságait képes megváltoztatni. A *Get-Variable* egy már létrehozott változó egészét adja vissza – tehát nem csak az értékét kapjuk meg, hanem egyéb tulajdonságait is[1]. A *Clear-Variable* a változóban tárolt értéket állítja üres *$null* értékre. A *Remove-Variable cmdlettel* a változónk megszüntetésére van módunk.

3.3.7.9 Format-Table, Format-List, Format-Wide

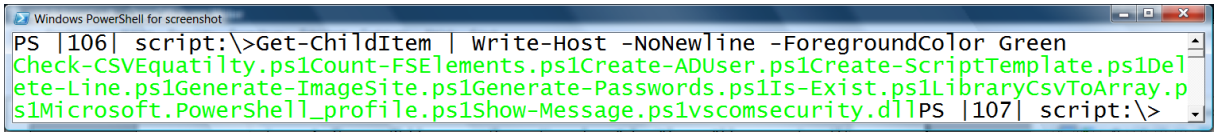
Valahányszor eddig kiadtunk egy parancsot, az valamilyen kimenettel szolgált. Mivel a kimenettel többé kevésbé meg voltunk elégedve – tehát olvasható, jól értelmezhető volt – ezért fel sem merült bennünk az a gondolat, hogy azt vajon lehet-e másképp is „látni”. Természetesen a válasz: igen. Köszönhetően a PowerShell OOP támogatásának, a csövezési technológiának és a formázó *cmdlet-eknek*, a parancsaink kimenete formázható. Az OOP fejezetben említettem, hogy valójában minden cmdlet kimenete vagy egy objektum, vagy objektumoknak egy gyűjteménye. Ugyanígy arról is volt szó, hogy a PowerShell minden parancsom végére odailleszt egy *'| Write-Output'* részt, ami azt csinálja, hogy a parancsom eredmény objektumát – vagy objektumait – egy csövön(|) keresztül a *Write-Output cmdlet*-hez irányítja – ami ebben az esetben a konzolon jeleníti meg a kimenetet. Ugyanezt meg tudjuk tenni a formázó cmdlet-ekkel is, ennek

1 Hiszen objektum orientál a parancssor, vele pedig a cmdlet-ek valamint azok eredményei is

eredményeképpen táblázatos, listaszerű, oszlopos eredményeket jeleníthetek meg. Figyelni kell arra, hogy ezek a formázó cmdlet-ek megváltoztatják a kimenetet, tehát ami a csővezeték egyik oldalán adott típusú objektum volt, az ezután nem biztos hogy az marad. Mindhárom cmdlet esetében a *-Property* kapcsoló mögött megadható, hogy az objektum mely tulajdonságait szeretném látni[1]. A *Format-Wide* cmdlet és a *Format-Table* tartalmaz egy *-AutoSize* kapcsolót, ami a konzolablak méretei alapján a lehető legjobb kihasználtsággal próbálja meg az eredményeket kilistázni. A *Format-Wide* tartalmaz egy *-Column* kapcsolót, amivel megadható az oszlopok száma. A *Format-Table* esetében, a *-Wrap* kapcsolóval új sorba törhetjük a képernyőből „kilógó" tulajdonságokat, *-HideTableHeaders*-szel pedig eltüntethetjük az oszlopfejléceket.

3.3.7.10 Write-Output, Write-Host

A PowerShell író utasításai közül ezt a kettőt szoktuk általánosan használni. Ránézésre két nagyon hasonló de mégis különböző cmdlet-tel van dolgunk. Az első *Write-Output* cmdlet egyedüli szerepe, hogy a kapott objektumokat továbbítsa a csővezetékben következő új parancs számára. Amennyiben nincs soron következő elem – mert nincs továbbirányítva a kimenet –, az objektum egyszerűen kiíródik a kimenetre. Ezzel szemben a *Write-Host* kifejezetten a konzolos megjelenítésre használjuk. Ezt az is jól mutatja hogy van *-ForeGroundColor* és *-BackGroundColor* kapcsolója, amivel az előtér és a háttér színeket tudjuk megadni. Emellett a *-NoNewLine* kapcsoló jó azokban az esetekben, mikor a csővezetéken kapott objektumokat egy sorban szeretnénk látni.



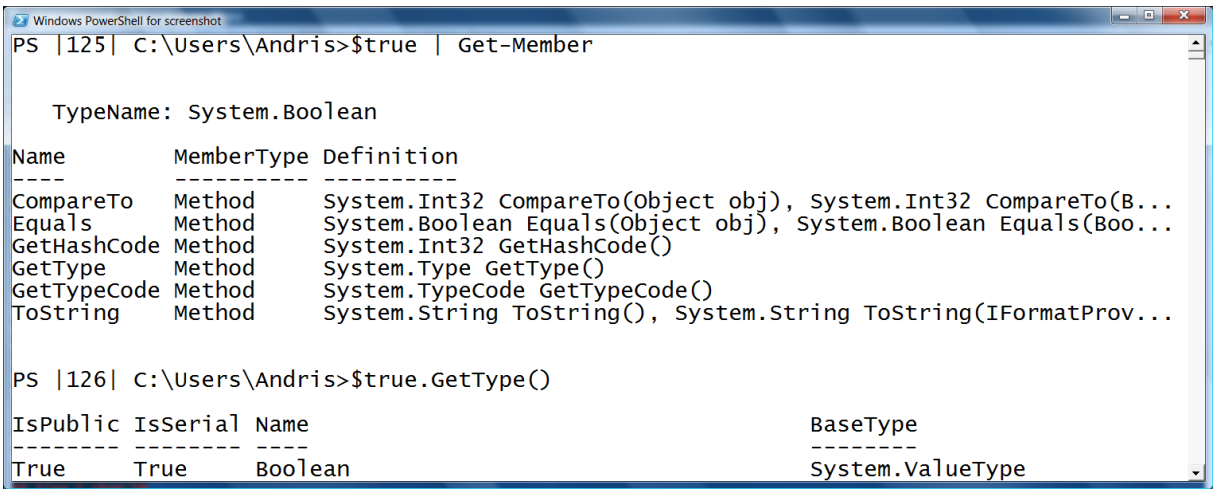
17. ábra: Sorfolytonosan zöld karakterrel Write-Host cmdlet-tel

1 Format-Wide esetén csak egy érték adható meg a kapcsoló mögött, a többinél veszővel elválasztva kell felsorolni a tulajdonságokat

3.3.7.11 Get-Member

Sokat volt már szó arról, hogy folyton mindenhol objektumokkal kommunikálunk, hogy az eredmény is objektum, de igazán valódi jeleit ennek még nem láttuk. A *Get-Member cmdlet* pontosan azért lett kitalálva, hogy mi minél hatékonyabban tudjuk ki aknázni az OOP lehetőségeket. Tudjuk hogyha objektumok vannak a háttérben, akkor őket számos feladatra rá lehet bírni a metódusaik segítségével, számos tulajdonságukat meg lehet fogni a mezőik segítségével. Már csak két kérdés vetődik fel: honnan tudom meg hogy milyen tagjellemzői vannak az adott objektumnak, illetve hogyan tudom őket meghívni, kiolvasni? Az objektumok metódusainak, mezőinek listázására természetesen a *Get-Member cmdlet-et* használjuk. Az objektum metódusait, mezőit és további tagjellemzőit pedig a .NET programozásból is ismert módon, pont(.) operátorral tudjuk megszólítani.

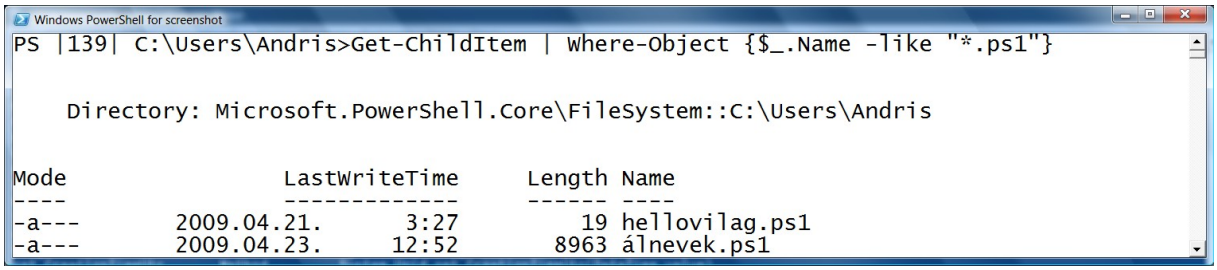
A *Get-Member* kétféle módon képes az objektumokat fogadni, az egyik módszer szerint az *-inputObject* kapcsoló után kell szerepeltetni az objektumot, a másik módszer szerint pedig csővezetéken kapja egy másik *cmdlet*-től. Mi a helyzet akkor, ha nem egy darab objektumot kap a *Get-Member*? Ekkor a *Get-Member* a bemenetre érkező összes lehetséges típus jellemzőit kiírja. Tehát ha egy olyan tömböt kap a *Get-Member* eredményül amiben fájlok és mappák vannak, akkor mind a fájl mind a mappa objektum jellemzőit – tagjait – kiírja.



18. ábra: Get-Member kiírja az IGAZ logikai érték tagjellemzőit

3.3.7.12 Where-Object

Az utolsóként tárgyalt *cmdlet* hasonlóan a *Get-Member-hez* általában csővezeték „belsejében” szokott állni. Szerepe az, hogy a hozzá érkező objektumokat valamilyen ismérv alapján szűrje vagy kiválassza[1]. Használata egyszerű, kapcsos zárójelek között[2] egy logikai kifejezést kell megadni, ha a kifejezés teljesül, az objektum a kimenetre kerül, ha a feltétel hamis a *Where-Object* nem küldi azt a kimenetre.



19. ábra: ps1 kiterjesztésű fájlok szűrése Get-ChildItem cmdlet kimenetéről Where-Object-tel

3.3.7.13 Sort-Object

Legvégül de nem utolsó sorban, a leggyakoribb cmdlet-ek közül a *Sort-Object-et* kell megemlítenem, mellyel a bemenetre érkező objektumok sokaságát valamely tulajdonság vagy tulajdonságok alapján rendezhetjük. Használata roppant egyszerű, a *-Property* kapcsoló után kell megadnunk a rendezés ismérvét – vagy ismérveit –, melyből azután a a *Sort-Object* alapértelmezés szerint egy növekvő sorozatot fog képezni a kimenetre. Az alapértelmezett növekvő rendezést *-Descending* kapcsolóval tudjuk csökkenőre változtatni. A *-Unique* kapcsolóval pedig az elemek ismétlődését kerülhetjük el. Példát erről a függelék 'Sort-Object egyedi és nem egyedi elemekkel'feliratú képernyőnézetén láthatunk.

3.3.7.14 A jéghegy csúcsa

Az eddig „részletesebben” említett *cmdlet*-ek a mindennapi munkához kellenek. Ezeken kívül számos igen hasznos *cmdlet* létezik még. Azért hogy szakdolgozatomban témánál maradhassak, ezen cmdlet-ek előzőekhez hasonló tárgyalását viszont

1 A kiválasztás egy olyan speciális szűrés melynél tudjuk hogy a kritériumnak adott feltétel csak egy elemre lesz igaz.

2 Valójában ez egy úgynevezett ScriptBlock, egy olyan objektum mely utasításokat tartalmaz(hasonlóan mint a függvények) de nincs nevesítve

nem teszem most meg. Némileg viszont pótlom a hiányt az elkövetkező fejezetek szkriptjeihez fűzött magyarázatokban.

3.3.8 Vezérlő Utasítások

A PowerShell egy szkriptek futtatására alkalmas környezet, azaz a parancssorba beírható utasításokat rögzíthetjük egy szöveges fájlban, melyet később automatizálási célokra, rutin adminisztrációs műveletekre felhasználunk. A szkriptekbe rögzített algoritmusok sokszor nem csak *cmdlet*-ek, függvények szekvenciális végrehajtásából állnak, hanem tartalmaznak szelekciót és iterációt is, nézzük meg milyen vezérlő utasítások találhatóak a PowerShell-ben.

3.3.8.1 Szelekció

Szelekciós vezérlési szerkezetet jellemzően akkor alkalmazunk, ha egy feltétel teljesülése esetén szeretném, hogy egy bizonyos kódblokk lefusson vagy esetleg egy másik hajtódjon végre.

Szelekció	

1. ...
2. If (*logikaikif.*){
3. *utasítások*
4. } else {
5. *utasítások*
6. }
7. ...
	A 2. sor zárójelei között található logikai kifejezés IGAZ értéke esetén a 3. sorra kerül a vezérlés, egyébként a második blokkban folytatódik a program. Bármely blokk végrehajtása után a vezérlés a 7. sorban folytatódik.

1. ...
2. If (*logikaikif1*){
3. *utasítások*
4. } elseif(*logikaikif2*) {
5. *utasítások*
6. }
7. ...
	Az előzőekhez hasonlóan a vezérlés a 3. sorban folytatódik, ha az 1. logikai kifejezés igaz, amennyiben az érték hamis, a vezérlés csak akkor kerül az 5. sorra, ha a 2. logikai kifejezés igaz, egyébként a blokk után folytatódik a program

1. ... 2. Switch(*változó*){ 3. {*feltétel1*} 4. {*utasítások*} 5. {*feltétel2*} 6. {*utasítások*} 7. ... 8. default {*utasítások*} 9. } 10. ...	Összetettebb feltételek esetén switch szerkezetet használunk. A 2. sorban zárójelek között szerepeltetett változó értékét vizsgáljuk a 3-5 sorban található blokkok feltételei szerint. Fentről lefelé haladva ellenőrizzük hogy teljesül-e valamelyik feltétel és amennyiben igen, akkor a vezérlés a neki megfelelő kódblokkot hajtja végre. A blokk végrehajtása után a 10. sorra kerül a vezérlés.

3.3.8.2 Iteráció

Utasításaink többszöri végrehajtását ciklusokkal oldhatjuk meg. PowerShell-ben négyféle módon tudunk utasításokat iteráltatni, két feltételes ciklussal, egy léptetős ciklussal, és egy összetett adatszerkezet elemeit végigléptető ciklussal.

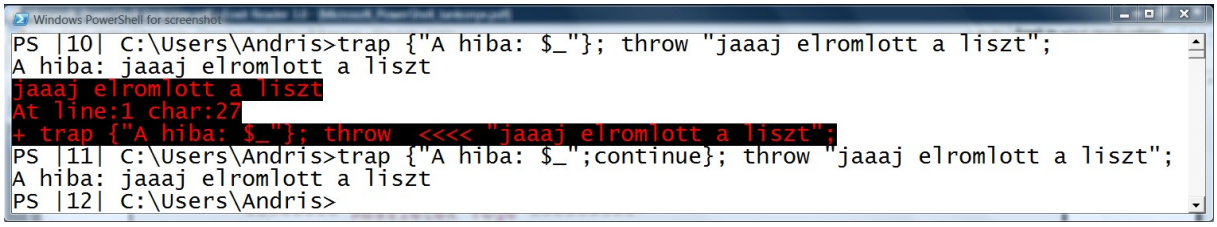
Ciklus	
1. ... 2. do{ 3. *utasítások* 4. }while(*logikaikif*) 5. ...	A do-while ciklus belső blokkja mindenesetben legalább egyszer lefut. A belső blokk végrehajtása után a vezérlés a 4 sorban található logikai kifejezést vizsgálja, hogy teljesül-e. Mindaddig amíg a feltétel igaz lesz, a vezérlés a 2. sorban folytatódik ismételve a kódblokkot. A feltétel hamissá válása esetén a vezérlés az 5. sorban folytatódik
1. ... 2. while(*logikaikif*){ 3. *utasítások* 4. } 5. ...	A while elöltesztelős ciklus legelőször is vizsgálja, hogy teljesül-e a logikai kifejezés, azaz hogy IGAZ értéket ad-e. Amennyiben ez teljesül, akkor a ciklus belső blokkjában a 3. sortól folytatódik tovább a vezérlés. A blokk utolsó utasítása után a vezérlés újra a 2. sorra ugrik és ott megint az előző lépések lesznek érvényesek. A ciklus a logikai kifejezés HAMIS értéke esetén az 5. sornak adja a vezérlést.
1. … 2. for(*értekadas;logikaikif; léptetés*){ 3. *utasítások* 4. } 5. ...	A for-ciklus, egy ciklusváltozót használ, melynek ad egy kezdőértéket az értékadás résznél – ez csak akkor fut le, mikor a vezérlés először kerül a ciklushoz. Következő lépésben vizsgáljuk, hogy teljesül-e a logikai kifejezés. Amennyiben igen, a vezérlés a 3. sorban folytatódik, majd a blokk utolsó utasítása után végrehajtjuk a léptető utasítást. Ez után újravizsgáljuk a feltételt és ismétlődik az előző mondat lépéssorozata. Ha nem teljesül a feltétel, a programfutás az 5. sorban folytatódik.
1. ... 2. foreach(*elem* in *gyűjtemény*){ 3. utasítás 4. } 5. ...	A foreach-ciklussal egy több elemből álló adatszerkezet elemein iterálhatunk végig. Mindaddig amíg az adatszerkezetnek létezik következő eleme, a ciklus-blokk lefut. A ciklus belsejében az adatszerkezet soron következő elemére az *elem* azonosítóval hivatkozhatok.

3.3.8.3 Continue, Break, Return

A *continue* utasítást ciklusok belsejében célszerű alkalmazni. Az utasítással átugorhatjuk a blokk *continue* utáni utasításait és a blokk végére ugorhatunk. A *break* utasítással egy adott kódblokk belsejéből ugorhatunk ki a külső kódblokkba, a *return* a hívó programkódhoz tér vissza. Amennyiben szerepeltetünk utána valamilyen értéket, visszatéréskor a hívókódnak átadja azt.

3.3.8.4 A kivételkezelés, Throw és Trap

A vezérlő utasításokhoz közvetetten kapcsolódik a PowerShell héjprogramok hibakezelése. Szkriptjeink futtatásakor előfordul, hogy emberi vagy egyszerűen valamilyen más végrehajtás során képződő hiba[1] megszakítja a programfutást. A PowerShell kivételkezelése nyújt segítséget ilyen esetekben. Alapvetően két hibafajta létezik, megszakító és nem megszakító. Míg az első esetében a teljes utasítás sorozat végrehajtása megszakad, a második esetében visszakapjuk a hibát, de a hibától független részek végrehajtódnak. Azt, hogy hiba esetén mit szeretnék tenni, a kód legelején *trap* kulcsszóval tudom definiálni a lenti képernyőnézet szerint[2]. Ha a trap kulcsszó után és a hibakezelő blokk előtt kapcsos zárójelek között megadom a kivétel osztályt, külön hibakezelő rutinokat készíthetek minden lehetséges hibához.



20. ábra: Hiba dobása és elkapása

1 nem tudunk hivatkozni valamire, típusinkompatibilitás lép fel
2 Idézett szöveg Besenyő Pista bácsi egyik híres értekezéséből

3.3.9 Gyakorlati példák

3.3.9.1 Script sablon generálása

A nyelv rugalmassága miatt a szkriptjeink kódjának kialakításakor viszonylag nagy szabadsággal rendelkezünk. Mégis érdemes odafigyelni arra, hogy a szöveges fájl jól tagolt legyen, és a benne található utasításokat könnyen tudjuk elkülöníteni egymástól. Ez egyrészt programozási stílus kérdése másrészt kódszervezési feladat. A következő Create-ScriptTemplate.ps1 szkriptben megjegyzés karakterek segítségével INFO, PARAMTERS, FUNCTIONS, VARIABLES, MAINCODE részekre tagolom a létrehozandó szkriptem, illetve bejegyzem a szkript nevét, a fejlesztő nevét, a felhasználási célt, és a létrehozás dátumát.

A paraméterben karakterlánc típusú objektumokat várok, amennyiben nem érkezik bármelyikre érték, a PowerShell az alapértelmezés szerinti értéket adja bármely paraméternek. Ez azt eredményezi, hogy lefut a *$()* zárójele közötti kód. Jelen esetben ez egy kivételt vált ki, ami a megfelelő paraméter használatára és az érték pótlására hívja fel a felhasználó figyelmét.

A 27. sortól meg lehet figyelni, hogy az *`n* escape-pel fűzök sortörést az *$s* karakterláncba, kialakítva ezzel az létrehozandó szkript törzsét. A paramétereket nem az aposztrófok közé helyezve fogom a karakterláncba beleágyazni, hanem a '+' művelettel hozzáfűzöm a megfelelő részhez. A szkript utolsó utasításaként pedig az $s változó tartalmát eltárolom a szkriptfájlban.

3.3.9.2 Jelszavak előállítása

A következő példaszkript paraméterben meghatározott számú és hosszúságú jelszót állít elő. Első paraméternek várja a jelszavak számát, második paraméternek pedig a jelszavak hosszát. Amennyiben a paramétereket nem határozzuk meg, alapértelmezés szerint egy darab 5 karakter hosszúságú jelszót fog létrehozni a

szkript. A jelszavak generálását valójában a szkripten belül egy külön erre a célra írt függvény végzi, mely átveszi a jelszó hosszának megadott paraméter értékét, ellenőrzi, hogy megvan-e a minimális jelszóhossz, majd három tömb segítségével véletlenszerűen kis és nagy betűket, számokat és speciális karaktereket generál egy karakterláncba. A visszaadott karakterláncokat egy *Arraylist* típusú változóban tárolom, majd miután előállítottam a megfelelő számú jelszót és eltároltam az *ArrayList-ben,* ezt a listát visszaadom a hívó kódnak. A szkriptben megfigyelhető hogy a tömböm elemeire szögletes zárójelek között egy egész értékű számmal hivatkozok, valamint az is látszik, hogy a véletlen számokat a *random* osztály egy példányának segítségével generáltam

3.3.9.3 Címtár felhasználó létrehozása

Az előzőekben generált jelszavakat csak akkor tudom igazán felhasználni, mikor nagy mennyiségű felhasználót kell generálni. Erre való a Create-ADUser.ps1 szkript. A szkriptben újdonság, hogy a címtárunk kezeléséhez egy úgynevezett *ADSI provider*-t használunk – ez konkrétan a *DirectoryServices.DirectoryEntry* osztály egy rövidített neve. Ennek az osztálynak a segítségével képesek vagyunk Active Directory domain-hez csatlakozni és felhasználókat, csoportokat, szervezeti egységeket felvinni, manipulálni azokat. A szkriptünk a bekért paraméterek alapján legenerálja a fiók legfontosabb adatait, így a *upn*-t, a *SAMAccount Name*-t, a felhasználó e-mail címét. Ha szolgáltatunk jelszót, akkor a szkript ezt is beállítja majd bekapcsolja a fiókot. A domainhez történő csatlakozás során egy változóba megkapjuk a domain-ünk *ADSI* objektumát, ez után egy másik változóba az előző domain tároló metódusával létrehozunk egy felhasználót, beállítjuk adatait, majd a beállításokat véglegesítjük.

3.3.9.4 Egy Windows PowerShell Profil

A PowerShell profilok szerepe az, hogy inicializálják a parancssoros környezetet. Ez többnyire függvények, változók, konstansok létrehozását, *cmdlet*-ek futtatását, meghajtók létrehozását, objektumok példányosítását jelenti. A profil nem más mint egy PowerShell szkript, tehát működésében sem tér el az előzőektől. A dokumentum végi szkript, létrehoz egy-egy változót, egyet a gyakorló könyvtárra (schome), illetve egyet a kódaláíró tanúsítványomra. Mivel gyakran használjuk a *Get-Help cmdlet*-et, ezért erre is célszerű egy *alias*-t deklarálni. A következő sor meghajtót hoz létre egy olyan könyvtárra, melyben a végleges aláírt szkriptjeink találhatóak. Ez után betöltünk egy *dll*-t, mely majd lehetővé teszi, hogy elérjem a *Microsoft Virtual Server* com-objektum mezőit és metódusait. Az *edit-profile* függvénnyel a szkripteditorom betölti a PowerShell profilt szerkesztésre. A *save-commands* kimenti aktuális munkamenetben kiadott parancsokat egy *xml* fájlba. A *get-synopsis* függvény egy *cmdlet*-et vár paraméterben, és a kapott *cmdlet* használati útmutatóját írja ki, a *get-definition* ugyanezt teszi a *cmdlet* szintaxisával. A *sign-script* függvénnyel aláírhatom a paraméterben kapott fájlokat a saját kódaláíró tanúsítványommal. A függvényeim listázását a függvény meghajtón kiadott *Get-ChildItem* cmdlet-tel érem el a *get-functions* függvény meghívásakor. A *Set-VSObjectSecurity* a paraméterben kapott Virtual Server objektumon egy biztonsági szintet állít be. A *Create-Script* a fenti *Create-ScriptTemplate*.ps1 szkriptet használja szkript létrehozására, melyet aztán szkriptszerkesztőben nyit meg. A parancssorom html alakú mentését az out-*html* függvénnyel tehetem meg. A *get-methods* és *get-properties* függvényekkel egy objektum metódusait és mezőit listázhatjuk. A *prompt* egy speciális rendszer függvény, mellyel a parancssori prompt megjelenítését szabályozhatom. Az utolsó három sorban pedig a korábban feltöltött függvény információkat tartalmazó *hash* tábla tartalmát jelenítem meg formázott alakban.

4. Oktatás

A PowerShell oktatás nagy kihívást jelentő feladat. A hallgatóság az idő nagy részében egy egyhangú karakteres felületet lát, vagy egy szkripteditort – ami talán némileg több élményt nyújthat. A lehetőségeink nagyban korlátozottak, hiszen minden alkalommal erősen szorítkozunk az előző órai ismeretekre, illetve a parancssor korábbról ismert cmdlet-ek hiányában önmagában semmilyen formában nem nyújt segítséget az eligazodáshoz. Egy magas szintű programozási nyelv szabályaihoz képest a szintaktika is megengedőbb, mindezeken túl pedig a hibaüzenetek is kevésbé beszédesek. Éppen ezért az első órákon hangsúlyozottan kell arra törekedni, hogy megtalálják azokat az eszközöket melyekkel azonnal segíthetnek magukon. Ha pedig már kellő rutinnal tudják ezeket a segédeszközöket forgatni, megértették az alap szisztémát, jöhetnek a bonyolultabb parancssoros példák, végül pedig a szkriptek. Az általános cél tehát az, hogy a PowerShell-t önállóan tudják használni, ne riadjanak el az ismeretlen nevű *cmdlet-ektől,* értsék a *shell-szkript* írás lényegét, előnyeit, célszerű módszertanát, biztonsági kockázatait.

4.1 Első lépés: ismerkedés

Legelőször tehát fontos, hogy a hallgatók testre tudják szabni a konzolt, billentyűkombinációkkal kényelmesen és hatékonyan tudják kezelni az ott megjelenített információt. Fontos, hogy megismerjék a konzolban használható billentyűket, melyekkel szavakat törölhetnek, parancsokat hívhatnak vissza, parancstörténetet jeleníthetnek meg, másolhatnak, beilleszthetnek. Miközben ezt gyakorolják célszerű a régről ismert a régi *dir*, *copy, mkdir, rmdir* parancsokat használni, melyek nagy része *alias* formájában már alapértelmezés szerint létezik – amelyek nincsenek azokat hozza létre a tanár PowerShell profilok segítségével.

4.2 Rámutatni arra, hogy ez merőben más

Miután otthonosan mozognak a parancssorban, rá lehet mutatni arra hogy bár a régi parancsok használhatók, a háttérben mégsem az történik ami a *cmd*.exe-nél megszokott volt. Eddig szó sem volt *cmdlet*-ekről, most viszont be lehet vezetni ezek fogalmát. Mutassunk rá, hogy valójában az eddigi parancsok mindegyike egy álnév volt valamelyik *cmdlet*-re. Mutassuk be a *Get-Command* és *Get-Help* és az álnevekkel kapcsolatos parancsok használatát. Ismerkedjenek meg alaposan ezekkel, keressék vissza, hogy az előzőekben használt parancsok mely *cmdlet*-et jelentik. Tudjanak szinte bármiről információt gyűjteni, tudjanak hatékonyan *cmdlet*-et keresni, fontos hogy megértsék a szintaxis leírásokat.

4.3 Providerek és ismerkedés további cmdlet-ekkel

Ahhoz hogy a későbbiekben további cmdlet-ekkel foglalkozhassunk, először is meg kell értsük a *provider*-ek és meghajtók működését. Meg kell figyeljük, hogy a *cmdlet*-ek más és más környezetben különbözően viselkednek(például nem mindenhol lehetséges a szűrés). Mutassunk rá, hogy a meghajtók kezelése PowerShell-ben nem bonyolult feladat, néhány példában használjuk a *New-PSDrive, Remove-PSDrive cmdlet*-eket.

4.4 Változók és OOP

Ha bevezettük a *provider-eket* mint új téma a *variable*: meghajtóval megjelenik a változó fogalma, ekkor tartom szükségesnek a változók és az Objektum Orientált Programozási paradigma tárgyalását. Használjuk a változókkal kapcsolatos cmdlet-eket, hozzunk létre konstansokat. Figyeljük meg a változók viselkedését, például: hogyan törölhetjük a konstans változót, hogyan állíthatjuk azt alaphelyzetbe, figyeljük meg, hogy a konzolablak bezárása után az eddigi változóink megszűnnek,

vizsgáljuk meg a PowerShell „jelenleg hasznos”[1] automatikus változóit, mutassunk rá hogy alaphelyzetben változóink nem szigorúan típusosak, majd pedig használjunk egyszerű típusokat. Mutassuk be az aritmetikai, szöveges és összehasonlító operátorokat valamint a reguláris kifejezések használatát. Mutassunk rá az operátorok jellegzetességeire, hogy képesek automatikusan típuskonverziót végezni. Emellett hívjuk fel a figyelmet az ezzel járó veszélyekre is. Az eddig ismert cmdlet-eink kimeneteit rögzítsük változókba, mutassuk meg *Get-Member* cmdlet-tel – csővezeték nélkül – a tagjellemzőket. Hívjuk az objektumok metódusait, mezőit.

4.5 Ismerkedés a csővezetékkel és az operátorokkal

Miután megértettük, hogy a PowerShell objektumokkal operál, következhet a csővezeték tárgyalása, ehhez használjuk a formázó cmdlet-eket, a Sort-Object, a Select-Object, Write-Host, Foreach-Object, Where-Object, Group-Object cmdlet-et.

4.6 Operációs rendszert közvetlenül érintő cmdlet-ek használata

Gyakorlatias feladatokkal mutassuk be a *Get-Service, Start-Service*, *Stop-Service*, *Get-Process*, *Start-Process*, *Stop-Process*, *Get-Eventlog*, *Get-History*, *Set-Acl, Get-Acl* parancsokat. Figyeljünk arra, hogy a hallgatók rendelkezzenek a helyi gépen megfelelő jogosultságokkal a feladatok elvégzéséhez. Ehhez célszerű egy virtuális gépet használni, hiszen ezek a cmdlet-ek könnyen tudnak helytelen rendszerműködést előidézni. Ezek a foglalkozások jó alkalmat adnak ahhoz is, hogy e tekintetben óvatosságra hívják fel a hallgatók figyelmét.

4.7 Függvények, szűrő, iteráló cmdlet-ek, és vezérlési szerkezetek

Egy idő után eljutunk arra szintre mikor többsoros *cmdlet*-láncokat hozunk létre,

1 ~A tanulás ezen fázisában

a függvények bevezetésére és tárgyalására úgy gondolom ez az ideális idő. Mutassuk meg a paraméterek használatát. Adjunk olyan feladatokat, ahol szűrésre és iterálásra van szükség, ezzel újra gyakoroljuk a *Foreach-Object* és *Where-Object cmdlet-ek* használatát. Ennek kapcsán felmerül a különböző vezérlési szerkezetek használata, vizsgáljuk a szelekciós, iterációs és egyéb vezérlő szerkezeteket és rögzítsük függvényben őket. Az előző *ForEach* és *Where-Object cmdlet*-ek példáit alakítsuk át ilyen formára.

4.8 Szkriptek

Mutassunk rá a függvények tulajdonságaira és ezzel vezessük be a szkripteket. A szkripteknél ugyancsak célszerű tisztázni, milyen esetekben használjuk őket, milyen előnyökkel, hátrányokkal rendelkeznek. Térjünk ki a végrehajtási házirendekre, a szkriptek aláírására. Fontos, hogy szkriptjeink értelmezhetőek legyenek. Emeljük ki a későbbi felhasználás fontosságát, adjunk módszert az ebből eredő problémák elkerülésére. Törekedjünk arra, hogy a futás során kapott kivételeket kezeljük. Minél életszerűbb problémák felvetésével sarkalljuk a hallgatókat kreatív feladatmegoldásra. Egy adott problémára próbáljanak meg több és minél hatékonyabb szkripteket írni.

4.9 Továbbhaladás

Ekkorra a hallgatók eljutnak egy olyan szintre a tanulásban, hogy már önállóan is stabilan tudják használni a megismert eszközöket. A kurzus végén érdemes felhívni a figyelmet a különböző forrásokra, ahol folyamatosan nyomon követhetik a változásokat, fejlesztéseket, esetleg cikkek segítségével olvashatnak utána valaminek. Így elősegíthetjük PowerShell ismereteik későbbi felhasználását továbbkamatoztatását.

5. Összefoglalás

Bár már lassan 3 év telt el a Windows PowerShell kiadásától számítva, e parancssoros környezet terjedése többnyire mégis csak üzleti-termelési szférában mutatkozik. A „mindennapi" felhasználók pedig még mindig a régi *cmd.exe* mellett használják rendszerüket. A Windows 7 operációs rendszer kiadásával viszont egyértelmű a cél PowerShell-t illetően: a Microsoft platform alapértelmezett parancssoros környezetének szánják. A jövőben tehát úgy hiszem szükség lesz arra, hogy a felhasználók képesek legyenek a PowerShell-t mindennapi munkájuk során használni. Ezért tartom fontosnak azt, hogy átgondoljuk és minél jobb módszertant fejlesszünk ki a PowerShell tanításához.

Úgy gondolom hogy szakdolgozatomban sikerült a PowerShell-t tervezett módon bemutatni és tárgyalni. Tudva viszont azt, hogy a gyakorlatban milyen sok helyzetben volt már segítségemre, és mennyi lehetőség rejlik benne, mégis úgy érzem lenne még miről írnom. Ezért minden kedves érdeklődőnek bátran javaslom az irodalomjegyzékben szereplő PowerShell szakkönyveket, weboldalakat. Mindemellett pedig a következő e-mail címen várok minden témát érintő kérdést, problémafelvetést, javaslatot (andris@aries.ektf.hu).

Beláthatjuk tehát hogy az utóbbi néhány évben a PowerShell versenyképessé nőtte ki magát a parancssorok világában. A 2.0 verzió pedig bővülő cmdlet készletével valamint számos újdonságával a jövőben még inkább növelni fogja e parancssor értékét a rendszerfelügyeletek világában.

Irodalomjegyzék

[1] opensource.blogspot.com, 2007, http://useopensource.blogspot.com/2007/03/unscientific-linux-popularity-contest.html

[2] Google Trends, 2009, http://www.google.com/trends?q=fedora%2C+ubuntu%2C+suse%2C+debian%2C+mandriva%2C+red+hat&ctab=0&geo=all&date=ytd&sort=1

[3] Tyson Kopczynski, Windows Powershell korlátok nélkül, 2008

[4] wikipedia.org, 2009, http://en.wikipedia.org/wiki/Windows_PowerShell

[5] Ed Wilson, PowerShell Scripting Guide, 2008

[6] Soós Tibor és Szerényi László, Microsoft Powershell 1.0 rendszergazdáknak - elmélet és gyakorlat, 2008

[7] Powerscripting Podcast, 2009, http://powerscripting.wordpress.com/

[8] Powershell Technet documentation, 2009 http://technet.microsoft.com/en-us/library/bb978526.aspx

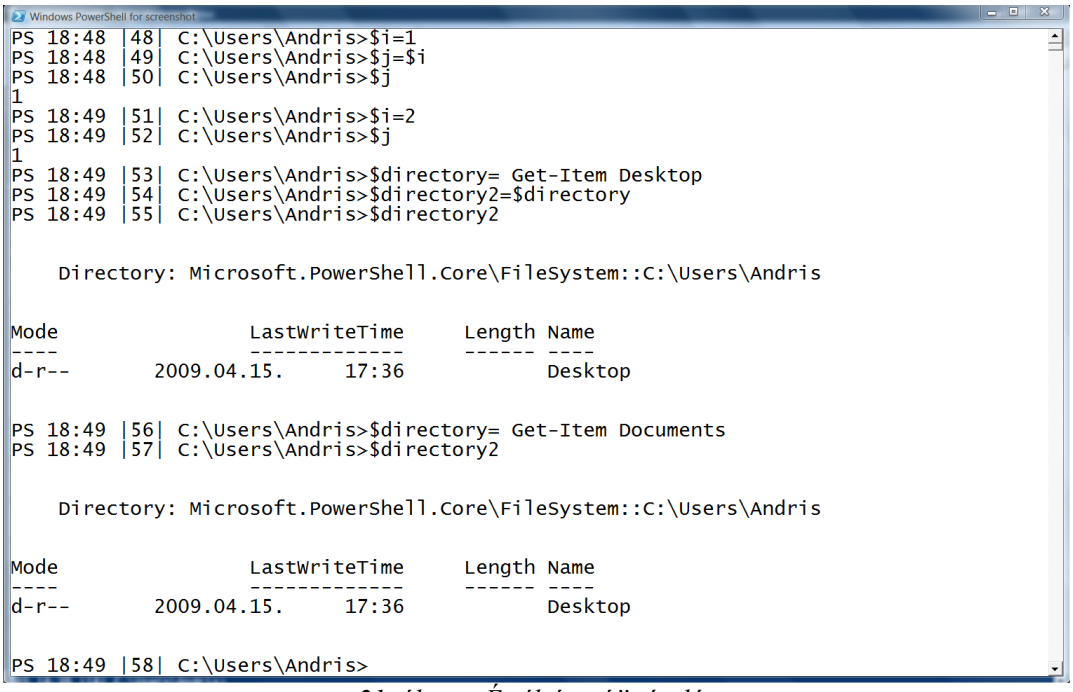
Táblázatjegyzék

Ábrajegyzék

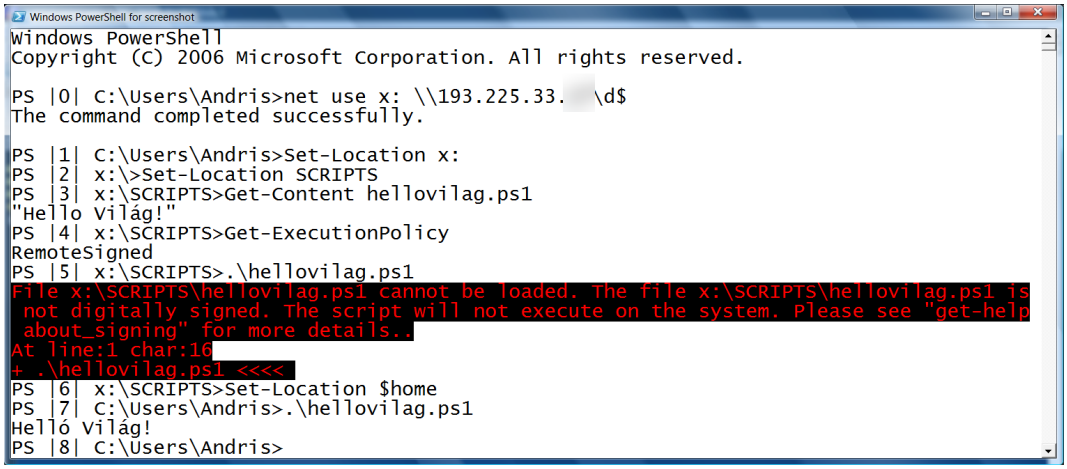
Függelék

Alapértelmezett cmdlet-ek			
Add-Content	Get-EventLog	New-Item	Set-ExecutionPolicy
Add-History	Get-ExecutionPolicy	New-ItemProperty	Set-Item
Add-Member	Get-Help	New-Object	Set-ItemProperty
Add-PSSnapin	Get-History	New-PSDrive	Set-Location
Clear-Content	Get-Host	New-Service	Set-PSDebug
Clear-Item	Get-Item	New-TimeSpan	Set-Service
Clear-ItemProperty	Get-ItemProperty	New-Variable	Set-TraceSource
Clear-Variable	Get-Location	Out-Default	Set-Variable
Compare-Object	Get-Member	Out-File	Sort-Object
ConvertFrom-SecureString	Get-PfxCertificate	Out-Host	Split-Path
Convert-Path	Get-Process	Out-Null	Start-Service
ConvertTo-Html	Get-PSDrive	Out-Printer	Start-Sleep
ConvertTo-SecureString	Get-PSProvider	Out-String	Start-Transcript
Copy-Item	Get-PSSnapin	Pop-Location	Stop-Process
Copy-ItemProperty	Get-Service	Push-Location	Stop-Service
Export-Alias	Get-TraceSource	Read-Host	Stop-Transcript
Export-Clixml	Get-UICulture	Remove-Item	Suspend-Service
Export-Console	Get-Unique	Remove-ItemProperty	Tee-Object
Export-Csv	Get-Variable	Remove-PSDrive	Test-Path
ForEach-Object	Get-WmiObject	Remove-PSSnapin	Trace-Command
Format-Custom	Group-Object	Remove-Variable	Update-FormatData
Format-List	Import-Alias	Rename-Item	Update-TypeData
Format-Table	Import-Clixml	Rename-ItemProperty	Where-Object
Format-Wide	Import-Csv	Resolve-Path	Write-Debug
Get-Acl	Invoke-Expression	Restart-Service	Write-Error
Get-Alias	Invoke-History	Resume-Service	Write-Host
Get-AuthenticodeSignature	Invoke-Item	Select-Object	Write-Output
Get-ChildItem	Join-Path	Select-String	Write-Progress
Get-Command	Measure-Command	Set-Acl	Write-Verbose
Get-Content	Measure-Object	Set-Alias	Write-Warning
Get-Credential	Move-Item	Set-AuthenticodeSignature	
Get-Culture	Move-ItemProperty	Set-Content	
Get-Date	New-Alias	Set-Date	

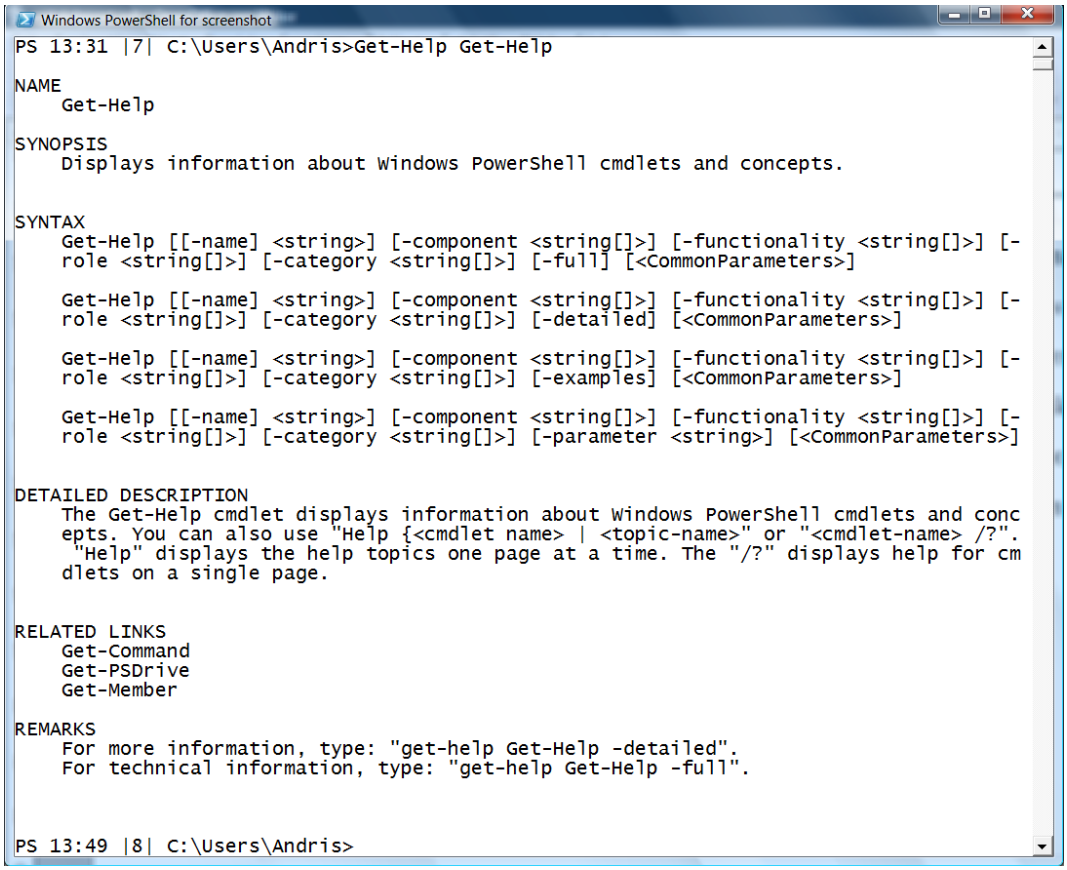
2. táblázat: Alapértelmezett cmdlet-ek listája



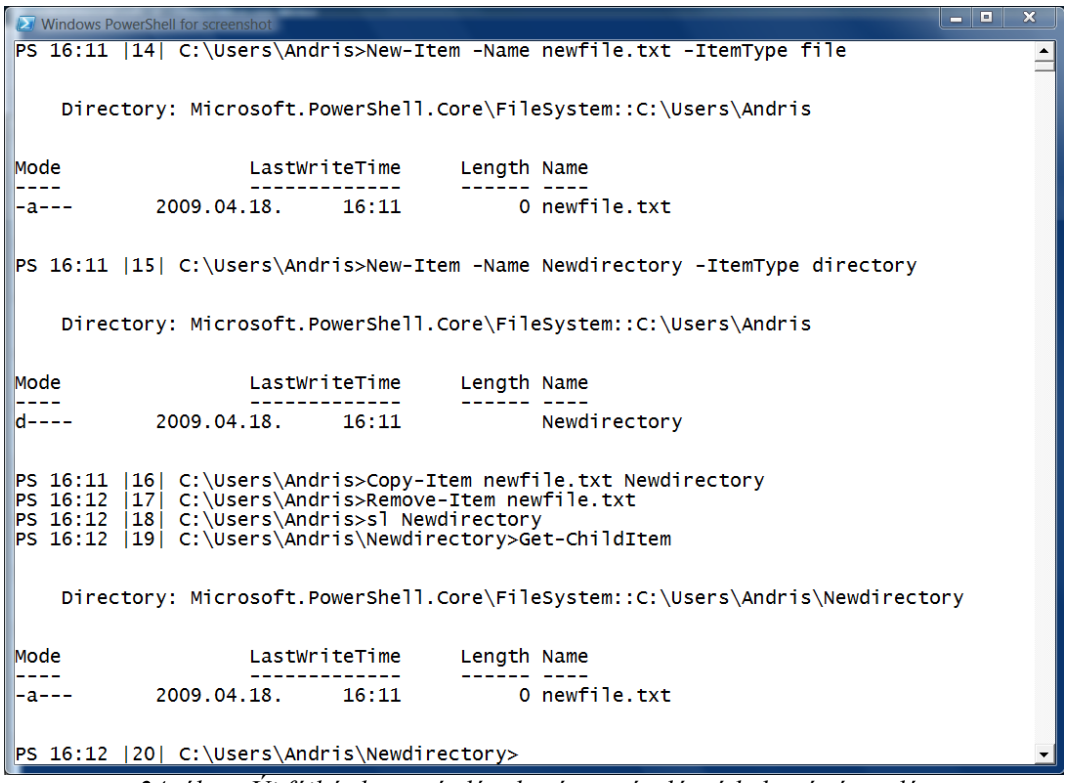
21. ábra: „Értéktípusú” tárolás



22. ábra: RemoteSigned végrehajtási házirend esetén hálózati meghajtón csak aláírt szkripte futtathatók



23. ábra: Get-Help cmdlet



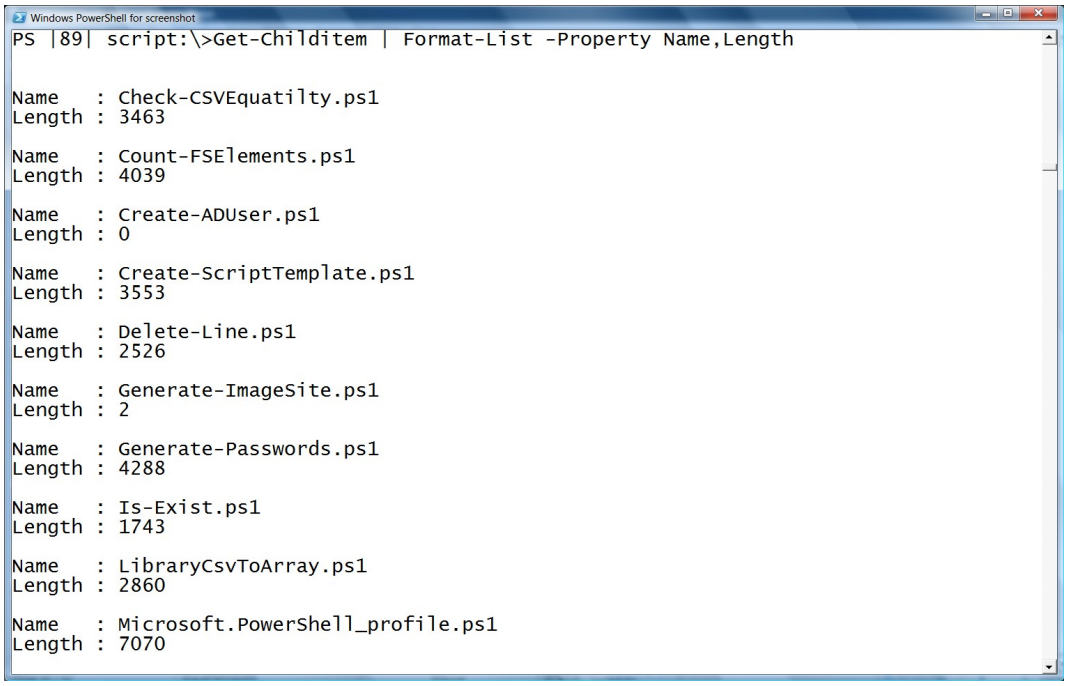
24. ábra: Új fájl és könyvtár létrehozása, másolás, áthelyezés és törlés

Változó neve	Leírása
$_	Az csővezetéken keresztül kapott objektum
$^	A legutoljára kiadott parancs első tagja
$$	A legutoljára kiadott parancs utolsó tagja
$?	A legutolsó kiadott utasítás sikerességét/sikertelenségét rögzíti
$Args	Paraméterátadás során használhatjuk szkriptjeinkben függvényeinkben
$Error	Hiba esetén ide rögzülnek a hibával kapcsolatos információk
$foreach	Foreach ciklus belsejében léptetésre, az aktuális elem kiolvasására használhatjuk
$HOME	Az aktuális felhasználó home könyvtára
$Input	A csővezetékkel függvénybe vagy kódblokkba irányított objektum
$Match	Reguláris kifejezések esetén a találatokat tároló hash tábla
$MyInvocation	Az aktuális parancssorról illetve szkriptről tárol információit
$Host	Az aktuális hosztról tárol információkat
$LastExitCode	Legutoljára végrehajtott külső program kilépési kódja
$true	Logikai igaz érték
$false	Logikai hamis érték
$null	Null érték
$OFS	Tömbök szöveggé alakításokat az elemek közötti elválasztó karakter
$ShellID	Parancskörnyezet azonosító, az érték alapján meghatározható az ExecutinPolicy és hogy milyen profilokat használ indításkor
$StackTrace	A hibakezelés során a nyomonkövetési információkat tároló verem

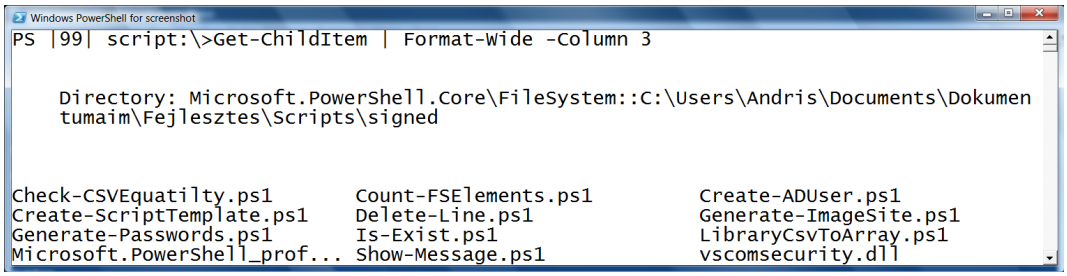
3. táblázat: Powershell automatikus változói[5]

Rövid név	Teljesen minősített név
[int]	System.Int32
[long]	System.Int64
[string]	System.String
[char]	System.Char
[bool]	System.Boolean
[byte]	System.Byte
[double]	System.Double
[decimal]	System.Decimal
[float]	System.Single
[single]	System.Single
[regex]	System.Text.RegularExpressions.Regex
[array]	System.Array
[xml]	System.Xml.XmlDocument
[scriptblock]	System.Management.Automation.ScriptBlock
[switch]	System.Management.Automation.SwitchParameter
[hashtable]	System.Collections.Hashtable
[psobject]	System.Management.Automation.PSObject
[type]	System.Type
[datetime]	System.DateTime
[void]	System.Void

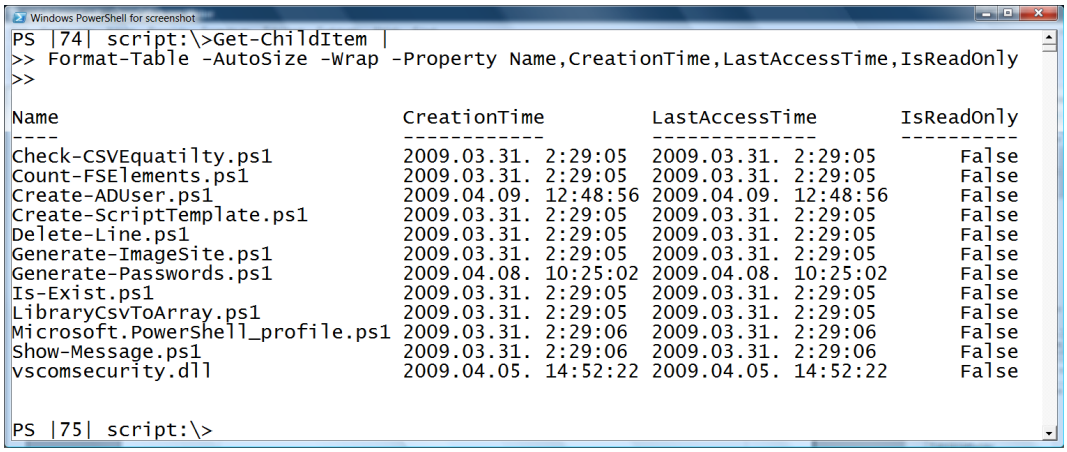
4. táblázat: Típusok rövidnévvel és minősített névvel[6]



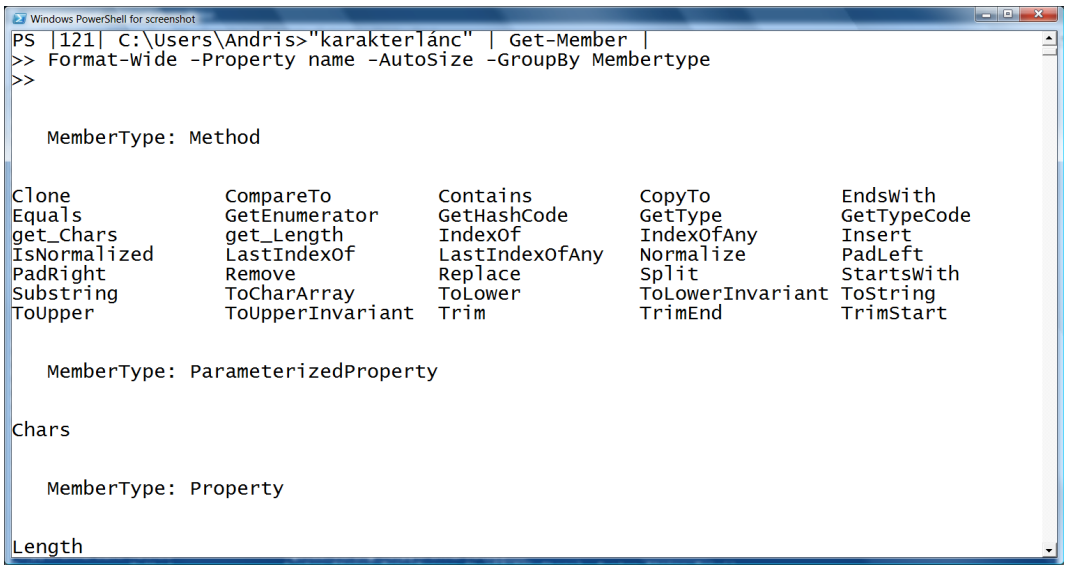
25. ábra: Format-List egy hasznos alkalmazása



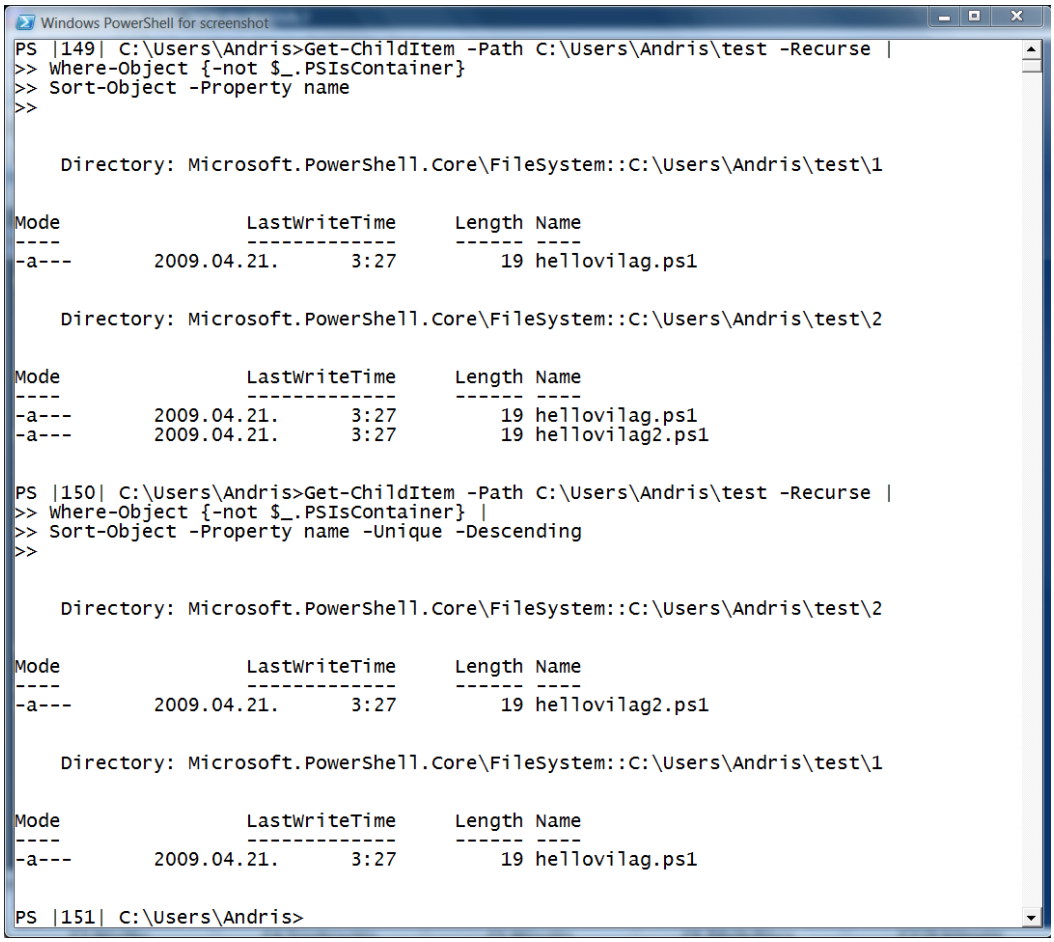
26. ábra: Format-Wide egy alkalmazása



27. ábra: Format-Table egy hasznos alkalmazása



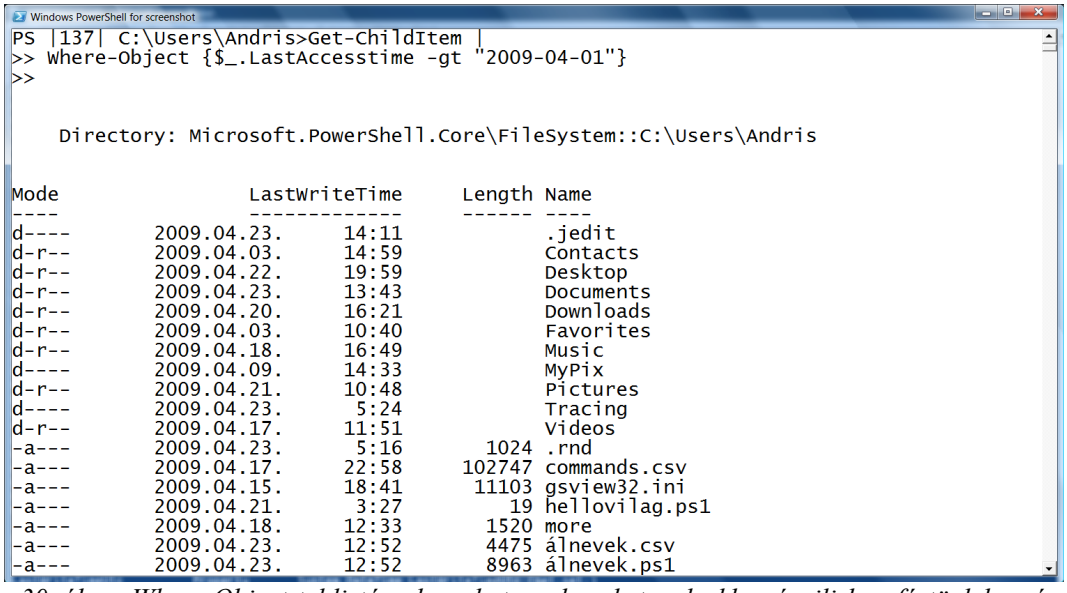
28. ábra: Egy karakterlánc objektum tagjai



29. ábra: Sort-Object egyedi és nem egyedi elemekkel

Rövid név	Teljesen minősített név
TAB	Automatikus kiegészítés, további TAB további találatokat ad
Shift + TAB	Visszalép az előző találatra
F1	1 karakter a történet-tárból
F2	Beírt karakterig másol
F3 vagy FEL	Előző parancs a történet-tárból
F4	Kurzortól az adott karakterig töröl
F5 vagy LE	Lépked vissza a történet-tárból
F7	Történettárat megjeleníti
F8	Parancstörténet, de csak a már begépelt szövegre illeszkedőeket
F9	Bekéri a történettár adott elemének számát és futtatja
Jobbkattintás	Beillesztés
Egérrel kijelöl + Enter	Másolás
JOBBRA	Navigálás jobbra karakterenként
BALRA	Nevigálás balra karakterenként
ESC	Törli az aktuális parancssort

5. táblázat: Konzolon használható billentyűkombinációk[6]



30. ábra: Where-Object-tel listázzuk azokat az elemeket melyekhez áprilisban fértünk hozzá utoljára

PéldaSzriptek

Create-ScriptTemplate.ps1

<vissza a magyarázathoz>

```
#INFO##########################################################################
#       Name:     Create-ScriptTemplate.ps1
# Created by:     Perjési András
#      Usage:     Used for creating main skeleton of a script
#       Date:     2008.11.08 16:24
################################################################################

#PARAMETERS#####################################################################
param(
      [string]$fileName=$(throw "Adja meg a script nevét kiterjesztéssel `" -filename`" kapcsoló után"),
      [string]$developer=$(throw "Adja meg a fejlesztõ nevét `" -developer`" kapcsoló után"),
      [string]$email=$(throw "Adja meg a felhasználás módját `" -usage`" kapcsoló után"),
      [string]$usage=$(throw "Adja meg a felhasználás módját `" -usage`" kapcsoló után")
)

#EXCEPTIONS#####################################################################

#FUNCTIONS######################################################################

#VARIABLES######################################################################

#MAINCODE#######################################################################

New-Item -Name $fileName -ItemType file
[string]$s=""
$s=
"#INFO##########################################################################`n"+
"#       Name:`t"+$fileName+"`n"+
```

```
29. "# Created by:`t"+$developer+"`n"+
30. "#     E-mail:`t"+$email+"`n"+
31. "#      Usage:`t"+$usage+"`n"+
32. "#       Date:`t"+(Get-Date -UFormat '%Y.%m.%d %H:%M')+"`n"+
33. "################################################################################`n"+
34. "`n"+
35. "#PARAMETERS#############################################################################`n"+
36. "`n"+
37. "#EXCEPTIONS#############################################################################`n"+
38. "`n"+
39. "#FUNCTIONS##############################################################################`n"+
40. "`n"+
41. "#VARIABLES##############################################################################`n"+
42. "`n"+
43. "#MAINCODE###############################################################################`n"
44. Out-File -Append -InputObject $s -FilePath $fileName
```

Generate-Passwords.ps1

<vissza a magyarázathoz>

```
#INFO###########################################################################
#       Name:     Generate-Passwords
# Created by:     Perjési András
#      Usage:     Paraméterben meghatározott számú jelszót állít elõ
#       Date:     2009.04.08 10:25
################################################################################

#EXCEPTIONS#####################################################################

#PARAMETERS#####################################################################
param(
      [int]$passLength=4,
      [int]$count=1
)

#FUNCTIONS######################################################################
function RandomPassword ([int]$intPasswordLength)
{
      if ($intPasswordLength -lt 4) {return "password cannot be <4 chars"}

      #számok tömbje, ebbõl választunk véletlenszerûen:
      $numbers = '1','2','3','4','5','6','7','8','9','0'
      #betûk tömbje, ebbõl választunk véletlenszerûen:
      $upcaseLetters =
'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'
      #szimbólumok tömbje ebbõl választunk véletlenszerûen:
      $symbols = "!%^&*()+=/?{}[]~,.<>:"
      #véletlenszerû generáláshoz:
      $rand = new-object random
      $s=""
```

```
30.
31.       for ($i=0; $i -le $passLength; $i++){
32.             sleep -Milliseconds 1
33.             $r = $rand.next(1,5) # [1,5[ intervallum
34.
35.             switch ($r)
36.                   {
37.                         "1" {$s += $numbers[$rand.next(0,$numbers.length-1)];break} #számok generálása
38.                         "2" {$s += $upcaseLetters[$rand.next(0,$upcaseLetters.length-1)];break} #nagybetûk generálása
39.                         "3" {$s += ($upcaseLetters[$rand.next(0,$upcaseLetters.length-1)]).ToLower();break} #kisbetûk
   generálása
40.                         "4" {$s += ($symbols[$rand.next(0,$symbols.length-1)]);break} #szimbólumok generálása
41.                   }
42.       }
43.
44.    return $s
45. }
46.
47. #VARIABLES#############################################################################
48. [System.Collections.ArrayList]$resultList = New-Object System.Collections.ArrayList
49.
50. #MAINCODE###############################################################################
51. for([int]$i=0;$i -lt $count;$i++){
52.       $tPass=RandomPassword($passLength)
53.       $resultList.Add($tPass) | Out-Null
54. }
55.
56. return $resultList
```

Create-ADUser

<vissza a magyarázathoz>

```
#INFO###########################################################################
#       Name:     Create-ADUser
# Created by:     Perjési András
#      Usage:     Felhasználói fiók létrehozása paraméterben meghatározott tartományban
#       Date:     2009.04.08 12:19
################################################################################

#PARAMETERS#####################################################################
param(
      [ADSI]$domain=[ADSI]"", #aktuális domain, ha mást meg nem adnak
      [string]$sn=$(throw "adja meg a felhasználó vezetéknevét -sn kapcsoló mögött"),    #vezetéknév
      [string]$givenName=$(throw "adja meg a felhasználó keresztnevét -givenName kapcsoló mögött"),  #keresztnév
      [string]$email=$(throw "adjon meg e-mail címet -email kapcsoló mögött"),     #a hallgató eléréséhez
      [string]$password,
)
#FUNCTIONS######################################################################
function cnToUpnSuffix{ # DC=example,DC=org formátumot example.org alakra hozza
      param(
            [ADSI]$domain
      )
      $tempString=($domain.distinguishedName.toString()).Split(",") |  # LDAP csomópontok szerint tömbelemekre bontjuk
            foreach {$_.Remove(0,3)}                        # eltávolítjuk a "DC=" karakterláncot az elejérõl
      return ([string]::join(".",$tempString))  # összefûzzük .-okkal ;D
}

#VARIABLES######################################################################
[string]$cn=$givenName+" "+$sn #létrehozandó felhasználó common name tulajdonsága
[string]$distinguishedName="CN="+$cn #relatív megkülönböztetõ neve
[string]$samAccountName=$givenName.ToLower()+"."+$sn.ToLower()
[ADSI]$container=$domain.psbase.Children | Where-Object {$_.name -eq "Users"} #tartomány Users tárolója
```

```
31. [string]$domainSuffix
32. #MAINCODE#########################################################################
33.
34. if(($password -ne "") -and ($password.length -lt 6)){
35.       throw $cn+": túl rövid jelszó"
36. }
37.
38. $domainSuffix=cnToUpnSuffix($domain)      #tetszõleges domain DNS formátumrahozása
39.
40. $user=$container.Create("user",$distinguishedName) # paraméterben karakterláncok user,cn=Orsolya Gork
41. $user.put("sn",$sn)     #Gork
42. $user.put("givenName",$givenName)   #Orsolya
43. $user.put("sAMAccountName",$samAccountName)     #orsolya.gork
44. $user.put("userPrincipalName","$sAMAccountName@$domainSuffix")    #orsolya.gork@example.org
45. $user.put("mail",$email)
46. $user.Setinfo()
47.
48. if ($password -ne ""){  #ha nem üres a jelszó karakterlánc
49.       $user.SetPassword($password)
50.       $user.psbase.InvokeSet("AccountDisabled","FALSE")
51.       $user.SetInfo()
52. }
53.
54. $user.psbase.properties
```

Microsoft.Powershell_profile.ps1

<vissza a magyarázó részhez>

```
1.  ###########################################################################
2.  #saját változók definiálása#############################################################
3.  $functions=New-Object System.Collections.ArrayList
4.  Set-Variable -Name schome -Value "C:\Users\Andris\Documents\WindowsPowerShell"
5.  Set-Variable -Name mycert -Value (Get-Item cert:\CurrentUser\My\0BCEA9FEC1C3B40A2A3EAF39B5379A94A5106E6A) -Option
    ReadOnly
6.
7.  ###########################################################################
8.  #saját alias-ok definiálás#############################################################
9.  Set-Alias gh Get-Help
10. New-PSDrive -Name script -PSProvider filesystem `
11.       -Root C:\Users\Andris\Documents\Dokumentumaim\Fejlesztes\Scripts\signed `
12.       -Description "Folder containing signed scripts" | Out-Null
13.
14.
15. ###########################################################################
16. #Virtual Server comobj kezeléséhez szükséges biztonsági dll betöltése##########
17. write-host -foregroundColor White -Object "
18. `n
19. ==Virtual Server Security dll betöltése=======================================
20. "
21. [System.Reflection.Assembly]::loadfrom("C:\Users\Andris\Documents\Dokumentumaim\Fejlesztes\Scripts\signed\vscomsecurit
    y.dll")
22.
23.
24. ###########################################################################
25. #Saját függvények#########################################################################
26. $functions.add(@{'Edit-Profile'='Windows PowerShell profil szerkesztése'}) `
27.       | Out-Null
28. function Edit-Profile{
29.       &"C:\Program Files\PowerGUI\Quest.PowerGUI.ScriptEditor.exe" $profile
```

```
30. }
31.
32. $functions.add(@{'Save-Commands'="Aktuális munkamenet parancsainak kimentése XML fájlba `nElérése
    `'$home\Documents\WindowsPowerShell\history\`' mappában"}) `
33.       | Out-Null
34. function Save-Commands{
35.       New-Item -Path "C:\users\Andris\Documents\WindowsPowerShell" `
36.             -Name temp.xml -ItemType file | Out-Null
37.       Get-History -Count $MaximumHistoryCount | Export-Clixml `
38.             -Path "C:\users\Andris\Documents\WindowsPowerShell\temp.xml";
39.       $mydate=(Get-Date).ToString("yyMMdd_HHmmss");
40.       Rename-Item -Path "C:\users\Andris\Documents\WindowsPowerShell\temp.xml" `
41.             -NewName $mydate".xml"
42. }
43.
44. $functions.add(@{'Get-Synopsis <commandlet>'='Commandlet használati infó'}) `
45.       | Out-Null
46. function Get-Synopsis{
47.       param(
48.             [string] $cmdletName=$(throw "Adja meg mely cmdlet használati információját szeretné megtudni")
49.       )
50.       get-help -Name $cmdLetName | Format-list -Property name, synopsis
51. }
52.
53. $functions.add(@{'Get-Definition <commandlet>'='Commandlet szintaxis definíciós infó'}) `
54.       | Out-Null
55. function Get-Definition{
56.       param ([string] $pattern)
57.       Get-Alias | Where-Object {$_.Definition.Contains($pattern)}
58. }
59.
60. $functions.add(@{'Sign-Script <skript teljeselérésiúttal> [-move|-copy]'="Skript aláírása sajáttanusítvánnyal illetve
    másolása/áthelyezése `'script:`' meghajtóra"}) `
61.       | Out-Null
```

```
function Sign-Script{
	param (
		[string] $scriptname,
		[switch] $move,
		[switch] $copy
	)

	Set-AuthenticodeSignature $scriptname $mycert

	if ($move){
		Move-Item $scriptname -destination script:
	}elseif ($copy) {
		Copy-Item $scriptname -destination script:
	}
}

$functions.add(@{'Get-Functions'='Összes PS függvény listázása'}) | Out-Null
function Get-Functions{
	Get-ChildItem -path function:\
}

$functions.add(@{'Set-VSObjectSecurity <VirtualServer comobj>'='Virtual Server objektum elérése'}) `
	| Out-Null
function Set-VSObjectSecurity{
	Param($object)
    [Microsoft.VirtualServer.Interop.PowerShell]::SetSecurity($object)
}

$functions.add(@{'Create-Script <szkriptnév> <fejlesztõnév> <felhasználás>'="Szkriptfájl létrehozása sablon alapján"})
`
	| Out-Null
function Create-Script{
	param(
		[string]$scname=$(throw "Adja meg a script nevét"),
```

```
95.             [string]$developer=$(throw "Adja meg a fejlesztõ nevét"),
96.             [string]$usage=$(throw "Adja meg felhasználási célt")
97.       )
98.       $scname+=".ps1"
99.       script:\Create-ScriptTemplate $scname $developer $usage
100.
101.            &"C:\Program Files\PowerGUI\Quest.PowerGUI.ScriptEditor.exe" (Get-Item $scname).FullName
102.      }
103.
104.
105.      $functions.add(@{'Out-Html [-desc <jelleg>] [-format <"print"|"powershell">] [-count <utolsósorok
    száma>]'="Konzol nyomtatás HTML-be"}) `
106.            | Out-Null
107.      function Out-Html{
108.            param(
109.                  [string]$desc="",             #convertált parancsok jellege
110.                  [string]$format="print",      #nyomtatási vagy powershell szerû séma
111.                  [int]$count=0                       #utolsó hány darab parancs legyen kimentve
112.            )
113.
114.            $utime= Get-Date -UFormat %m%d_%H%M
115.
116.            if($count -eq 0){
117.                  if($format -eq ""){
118.                        &"c:\Users\Andris\Documents\Dokumentumaim\Fejlesztes\Scripts\Web\get-bufferhtml.ps1" -palette
    $format |
119.                              Out-File -FilePath $schome\"buffer"$utime$desc".html"
120.                  }
121.            }else{
122.                  if($format -eq ""){
123.                        &"c:\Users\Andris\Documents\Dokumentumaim\Fejlesztes\Scripts\Web\get-bufferhtml.ps1" -palette
    $format -last $count |
124.                              Out-File -FilePath $schome\"buffer"$utime$desc".html"
125.                  }
```

```
126.            }
127.
128.      }
129.
130.      $functions.add(@{'Get-Methods <object>'="Bemeneti objektum metódus listája"}) `
131.            | Out-Null
132.      function Get-Methods{
133.            param(
134.                  $object
135.            )
136.            Get-Member -InputObject $object -MemberType Methods
137.      }
138.
139.      $functions.add(@{'Get-Methods <object>'="Bemeneti objektum tulajdonság listája"}) `
140.            | Out-Null
141.      function Get-Properties{
142.            param(
143.                  $object
144.            )
145.            Get-Member -InputObject $object -MemberType Properties
146.      }
147.
148.      $functions.add(@{'home'="Home könyvtárba ugrás"}) `
149.            | Out-Null
150.      function home{
151.            Set-Location $HOME
152.      }
153.
154.      function prompt {
155.            #'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
156.
157.            $private:id=(Get-History -Count 1).id
158.            $private:dateS=get-date -uFormat %H":"%M
159.            $private:locationS=Get-Location
```

```
160.            $nested=$(if ($nestedpromptlevel -ge 1) { '>>' })
161.
162.            if ($id -eq $null) {
163.                  $id=0
164.            }
165.
166.            "PS |{0}| {1}>{2}" -f  $id, $locationS,$nested
167.
168.            #'PS ' + $(get-date -uFormat %H":"%M) + ' |' + $(if ($id -eq $null) {0} else {$id}) + '| ' + $(Get-
    Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
169.      }
```