Skriptiranje u naredbenom retku pomoću jezika Python

Varga, Petar

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Petar Varga

Skriptiranje u naredbenom retku pomoću jezika Python

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Petar Varga

Matični broj: 45873/17–R

Studij: Informacijski sustavi

Skriptiranje u naredbenom retku pomoću jezika Python

ZAVRŠNI RAD

Mentor:

Dr. sc. Zlatović Miran

Varaždin, kolovoz 2021.

Petar Varga

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Skriptiranje se naspram klasičnog korištenja grafičkog sučelja razlikuje po tome da je isključivo tekstualno bazirano. Iako su grafička sučelja prikladna za prosječne korisnike računala, nije mnogo pažnje pridodano za automatizaciju zadataka. Automatizacijom korisnici postižu obavljanje jedne ili više naredbi bez nužne kontrole ili aktivne interakcije. Za automatizaciju klasično su korišteni skriptni jezici, odnosno ljuske koje operacijski sustav pruža. Kod Microsoft Windows operacijskih sustava najkorištenija ljuska je *PowerShell*, dok je kod *Unix* baziranih operacijskih sustava to ljuska *sh* ili *bash*. Iako ugrađene ljuske imaju mnogo mogućnosti, alternativa korištenju uključenih ljuski je korištenje skriptnog programskog jezika kao što je Python. Python-a karakterizira potpuna implementacija programskog jezika visoke razine te između ostaloga ima odlike visoke razine proširljivosti kroz dodatne module te ga odlikuje prilagodljivost raznim jednostavnim i kompleksnim zadacima. Pomoću njega možemo implementirati jednako ili čak bolje sva rješenja koja se mogu kreirati i u skriptama izvršenim putem ljuski operacijskih sustava.

Ključne riječi: Python; ljuska; *PowerShell*; *bash*; skriptiranje; naredbeni redak; Windows; Linux

Sadržaj

1. Uvod

Tema završnog rada nastoji prikazati Python kao alternativni jezik za izradu automatizacijskih i općenitih skripata za korištenje putem naredbenog retka. Iako su ljuske *sh* i *PowerShell* uključene kod standardnih instalacija Linux i Windows operacijskih sustava, kroz ovaj rad će se nastojati približiti korištenje Pythona za skriptiranje i obavljanje radnji koje se najčešće obavljaju u ljuskama operacijskih sustava. Python je izrazito moćan alat sposoban ispuniti sve mogućnosti koje ljuske pružaju te još dodatne mogućnosti koje ljuske teško obavljaju. Motivacija uzimanja ove teme proizlazi iz kontinuiranog rada s Python programskim jezikom kroz trajanje studija, ali i privatno. Često sam se susretao s potrebom za automatizacijom nekih jednostavnih zadataka, te imao ponekih problema sa standardnim načinom rješavanja istih kroz ljuske operacijskih sustava. Smatram da bi Python uvelike olakšao posao skriptiranja radi korištenja standardnih programskih konstrukata i čitljive i razumljive sintakse. Također iza Pythona stoji puno veća zajednica ljudi spremnih pomoći, dok je teže naći rješenja implementirana u *Shell*-u ili *PowerShell*-u.

2. Metode i tehnike rada

Za razradu teme primarno će se koristiti metode istraživanja dokumentacije programskih jezika, ljuski i opća dokumentacija vezana uz područja koja će se proučavati u ovom radu. Od izvora će se također koristiti internetski servisi s provjerljivim izvorom informacija te izdane knjige. Primjeri programskog koda bit će implementirani tako da su lako provjerljivi, odnosno da je moguća uspješno pokretanje cijelih isječaka koda. Prilikom izrade skriptnih rješenja koristit ću Ubuntu 18.04.5 LTS kao i Windows 10 operacijski sustav. Na Ubuntu je instaliran *PowerShell* 7.1.4 koristeći službenu Microsoft verziju. Na Windows 10 instaliran je *PowerShell* verzije 7.1.4. Verzija Python interpretera je 3.6.9. na oba sustava. Za potrebe *Unix* ljuski bit će korištena ljuska *sh* i *bash*. Kao editor koda bit će korišten *Visual Studio Code.*

3. Sučelje naredbenog retka

Današnja sučelja za upravljanjem raznim aplikacijama ili složenijim sustavima kreirana za masovnu upotrebu većinom su grafičkog oblika. Grafički oblik interakcije korisnika i računalnog sustava, odnosno programa, podrazumijeva korištenje miša za navigaciju kroz izbornike (eng. *menu*). Vrlo česta je upotreba grafičkih navigacijskih uzoraka kao što su horizontalna ili vertikalna navigacijska traka (eng. navigation bar) gdje se linkovi s putanjama mogu grupirati u više razina ovisno o njihovoj kontekstualnoj povezanosti [22]. Kod složenih sustava, a pretežito kod elektroničkih sustava za trgovinu, gdje je hijerarhija vrlo kompleksna s mnogo razina i putanja ovisno o željenim akcijama dodatno se može koristiti uzorak dizajna „trag mrvica kruha" (eng. *breadcrumb*). Takav dizajn podrazumijeva linearnu strukturu od korijenskog elementa do trenutnog čvora na kojem se nalazimo [22]. Neovisno koji grafički sustav interakcije, odnosno uzorak dizajna, arhitekti sustava odaberu, glavna misao vodilja je olakšati navigaciju i korisniku pružati tzv. „point-and-click" iskustvo. Takva rješenja najčešće ispadnu vrlo komplicirana te iako se krajnji korisnici uspiju uspješno snalaziti u njima, ona su gotovo uvijek nemoguća za automatizirati. Pod pojmom automatizacije podrazumijevamo korištenje jedne operacije ili određene sekvence više operacija koju program, sustav ili općenito neki dio software-a nudi korisniku, bez ljudske interakcije u samom procesu izvođenja. U automatizaciji očekujemo da će krajnji skup naredbi biti uvijek izvršen točno onako kako je i definirano u uputama automatizacije. Grafički sustavi se najčešće ne mogu automatizirati jer je njihov dizajn od početka dinamičan i sklon konstantnim promjenama. Suprotnost sustavima s grafičkim upravljanjem su sustavi koji koriste naredbeni redak, odnosno naredbeno-linijsko sučelje (eng. *command-line interface* - CLI). Korištenjem naredbenog retka, korisnik je u interakciji s računalom isključivo putem tekstualnog oblika, svaka komanda je ručno napisana i svaka komanda mora biti podržana od strane sustava koji prihvaća tekstualne naredbe. Naredbe koje se mogu zadati nekim tekstualnim zapisom, koje uvijek obavljaju istu radnju bez promjene u načinu zadavanja i koje ne trebaju čekati korisnika na daljnje akcije upravo su ono što omogućava i olakšava automatizaciju zadataka koja nije lako izvediva putem grafičkih sučelja. U suštini, naredbeno-linijsko sučelje procesira tekstualne naredbe korisnika prema računalnu odnosno određenom programu.

3.1. Interpreteri naredbenog retka

Uneseni korisnički tekst naredbi računalo mora pretvoriti u razumljiv set naredbi koje potom mora izvršiti. Takvi programi skupno su nazvani interpreterima naredbenog retka (eng. *command-line interpreter* ili *command-line processor*). Ovisno u kojem kontekstu koristimo naredbeni redak, možemo koristiti različite vrste interpretera – ljuske operacijskih sustava ili ljusku nekog od programskih jezika, odnosno skupa alata vezanih uz programski jezik. Ljuske operacijskih sustava razdvajamo ovisno o tome na kojem operacijskom sustavu su dostupne. Dok se ljuske programskih jezika dijele na vrstu programskog jezika odnosno razvojnog okvira.

3.2. Korištenje sučelja naredbenog retka

Kako bi korištenje naredbenog retka bilo intuitivno i lako shvatljivo, ljuske koriste sljedeći generalni uzorak izdavanja naredbi i prikaz rezultata [1, str. 212]:

```
<odzivnik ljuske> <naredba> <argument>
<rezultat>
```

Gdje su pojedinačni elementi obilježeni sljedećim svojstvima:

- Odzivnik ljuske - prikazana od strane ljuske, odnosno interpretera da korisniku daje kontekst u kojem je direktoriju ili okruženju.
- Naredba – upisuje korisnik, opisuje željenu komandu. Komande su klasificirane u sljedeće tri grupe:
    - Interne – naredbe procesirane direktno od strane interpretera, bez ovisnosti o vanjskim izvršnim datotekama.
        - Primjeri kod *bash* ljuske: `read`, `getopts` [2]
    - Uključene – naredbe koje su dio operacijskog sustava.
        - Primjeri kod *bash* ljuske: `date`, `grep`, ili `wc` [3]
    - Vanjske – naredbe koje su dodane od korisnika.
- Argument – N parametara koji su opcionalno pružani od strane korisnika. Parametri moraju biti podržani od strane naredbe kojoj ih prosljeđujemo. Parametri mogu biti:
    - Argumenti – predstavljaju informaciju koja je proslijeđena naredbi za izvršavanje.
    - Opcije – predstavljaju modifikatore operacije komande.
        - Kod naredbe `ls` možemo dodati opciju `-l` za ispis sadržaja direktorija, dok opcija `-la` ispisuje sadržaj direktorija dodatno sa sakrivenim datotekama.

Korištenje naredbeno-linijskog sučelja dano je sljedećim primjerom:

```
homeserver@homeserver:~$ python3 skripta.py 'Test'
HelloWorld Test
```

U ovom primjeru, u prvoj liniji `homeserver@homeserver:~$` predstavlja odzivnik ljuske, `python3` je vanjska naredba, `skripta.py` je prvi argument, `'Test'` je drugi argument i konačno u drugoj liniji rezultat izvršavanja ove vanjske naredbe jest `HelloWorld Test`. Datoteka `skripta.py` je skripta koja uzima argument te tekstualnu vrijednost tog argumenta dodaje tekstualnoj vrijednosti „HelloWorld“.

3.3. Interpreteri naredbenog retka operacijskih sustava

Istraživač Stephen Bourne 1978. godine razvio je *Bourne Shell* za Version 7 *Unix* te je toj ljuski dan naziv *sh*. *Bourne Shell* postavio je standard značajki koje sve buduće pa tako i današnje ljuske imaju [1, str. 211]. Neke od ostalih najčešće korištenih ljuski u *Unix* obitelji operacijskih sustava su *Bourne-Again shell* (*bash*), *C shell* (*csh*), *Z shell* (*zsh*), itd. Najpopularnije izvorne ljuske operacijskog sustava Windows su *cmd* i *PowerShell*. *PowerShell* je najnovija inačica Windows-ove ljuske te zahvaljujući integraciji sa .NET Core razvojnog okvira pruža više mogućnosti nego *cmd*. Zanimljivo je spomenuti kako je za razliku od *Unix* ljuske *PowerShell* objektno orijentiran skriptni jezik [4]. Za razliku od izvornih Windows ljuski, kroz lagani (eng. *lightweight*) sloj kompatibilnosti moguće je imati Linux razvojno okruženje, te samim time i pristup *Unix* ljuskama [5].

Primjer *bash* naredbe za izlist svih datoteka koje imaju ekstenziju txt u trenutnom radnom direktoriju:

```
ls *.txt
```

Primjer *PowerShell* naredbe za izlist svih datoteka koje imaju ekstenziju txt u trenutnom radnom direktoriju:

```
ls *.txt
```

Primjer *cmd* naredbe za izlist svih datoteka koje imaju ekstenziju txt u trenutnom radnom direktoriju:

```
dir *.txt
```

3.4. Interpreteri naredbenog retka programskih jezika

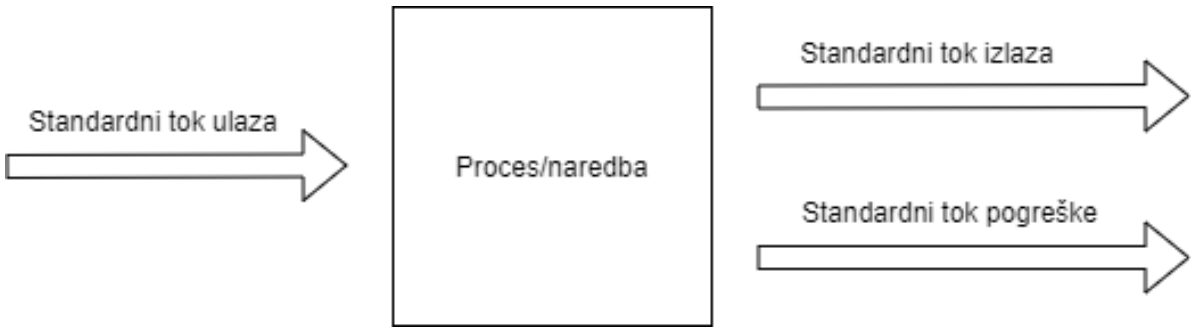
Druga već spomenuta osnovna vrsta ljuske jest ljuska programskog jezika. Kako postoje puno više programskih jezika nego postoji operacijskih sustava, tako postoje različite ljuske, neke od kojih su Python, JavaScript, Perl, PHP, Ruby, Julia, Matlab, itd. Da bi neki programski jezik bio interpreter mora podržavati REPL (kratica za eng. *Read Eval Print Loop*) okruženje, odnosno unutar interaktivnog okruženja mora primiti korisnički ulaz, izvršiti naredbu koju taj ulaz predstavlja u tom jeziku, ispisati rezultat naredbe i biti spreman tu istu sekvencu događaja ponoviti [10]. Kada je neki programski jezik u načinu rada kao interpreter naredbenog retka, tada možemo izvršavati sve mogućnosti koje taj programski jezik nudi liniju po liniju. Izvršava se neovisno o implementaciji naredbenog retka operacijskog sustava. Primjer Python naredbe za izlist svih datoteka koje imaju ekstenziju txt u trenutnom radnom direktoriju, pod uvjetom da smo deklarirali uvoz `glob` biblioteke:

```
glob.glob('./*.txt')
```

3.5. Međuprocesna komunikacija u naredbenom retku

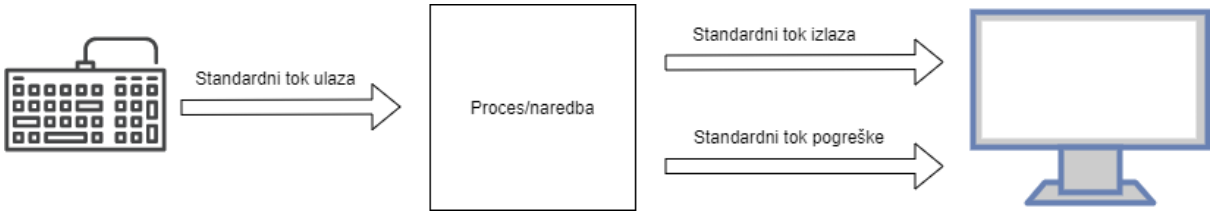
Interakcija računalnog programa, odnosno računalnog sustava i korisnika razvija se na principu da jedan drugome može davati informacije. U počecima ta razmjena je bila striktno u fizičkom obliku, da je korisnik unosio naredbu putem tipkovnice, a računalo je ispisalo rezultat na ekran. Polazeći od prve implementacije razmjene podataka, kasnije su razvijeni standardizirani tokovi (eng. *streams).* Tok je obostrana istovremena (eng. full-duplex) veza između korisničkog procesa i uređaja ili pseudo-uređaja. Kod standardnih tokova definirana su tri toka: standardni tok ulaza (eng. *Standard input, stdin*), standardni tok izlaza (eng. *Standard output, stdout*) i standardni tok pogreške (eng. *Standard error, stderr*) [6, str.1]. Standardni tok ulaza predstavlja tok kojeg program koristi za ulazne podatke, primjer uključenog programa koji koristi standardni ulazni tok jest `cat`. Naredba `cat` koristi ulazni tok te ga ispisuje na ekran. Neki programi ne trebaju ulazni tok, već je sama naredba dovoljna da se izvrši, primjer takvog programa jest `cd`. Nakon što se pokrene, ona će se izvršiti bez ikakvog standardnog ulaznog toka od strane korisnika. Standardni tok izlaza predstavlja tok kojim program ispisuje rezultate svojeg izvođenja, primjer uključenog programa koji vraća rezultat putem standardnog izlaznog toka jest već spomenuta naredba `cat`. Ona nakon što je tekst unesen, putem toka izlaza vraća isti tekst korisniku. Postoje i naredbe koje ne vraćaju nikakav rezultat nakon uspješnog izvođenja, primjer takve naredbe jest `rm`. Standardni tok pogreške predstavlja vrstu izlaznog toka koji upozorava na pogreške koje su se dogodile tijekom njegovog izvršavanja.

Iako on je vrsta izlaznog toka, on je potpuno odvojen tok te je odvojiv od standardnog izlaznog toka. Općenito interakcija standardnih tokova i procesa neovisno o uređajima koji su na ulaznoj ili izlaznoj strani dana je sljedećim dijagramom.



Slika 1. Dijagram interakcije standardnih tokova nad procesom

Uzevši u obzir da je najčešći uređaj ulaznog toga tipkovnica, a najčešći uređaj izlaznog toka monitor, prethodni dijagram možemo izmijeniti i dodati iste.



Slika 2. Dijagram interakcije standardnih tokova s prikazom najčešćih ulaznih i izlaznih uređaja

Kod ljuski u *Unix* obitelji operacijskih sustava standardne tokove moguće je preusmjeriti (eng. *redirect*) na proizvoljne lokacije. Preusmjeravanje toka obavlja se korištenjem < i > znakova između naredbi ili naziva procesa. Kod preusmjeravanja ulaznog toka koristimo < znak, dok kod preusmjeravanja izlaznog toka koristimo > [7]. Prilikom preusmjeravanja izlaza, budući da je izlaz „iskorišten“, tada ne vidimo izlaz na ekranu kao što je to do sada bio slučaj. Primjer naredbe preusmjeravanja standardnog izlaza:

```
homeserver@homeserver:~$ python3 skripta.py > program_rezultat.txt
```

Nakon izvođenja ove naredbe, datoteka `program_rezultat.txt` ima sadržaj `HelloWorld`. Prilikom izvođenja ove naredbe, bitno je napomenuti da na ekranu nije bio

prikazan nikakav tekst, za razliku od prijašnjeg primjera kada izlazni tok nije bio preusmjeren. Primjer preusmjeravanja ulaza dan je sljedećom naredbom:

```
homeserver@homeserver:~$ cat < program_rezultat.txt
HelloWorld
```

Nakon izvođenja ove naredbe, možemo uočiti da je sadržaj datoteke `program_rezultat.txt`, koju smo kreirali u prošloj naredbi, korišten kao standardni ulaz za naredbu `cat`. Ovdje je bitno napomenuti da nije bio korišten nikakav unos nakon pokretanja naredbe već je sve odrađeno pomoću preusmjeravanja. Korištenim primjerima preusmjeravanjem ulaznog i izlaznog toga, zapravo smo ostvarili međuprocesnu komunikaciju.

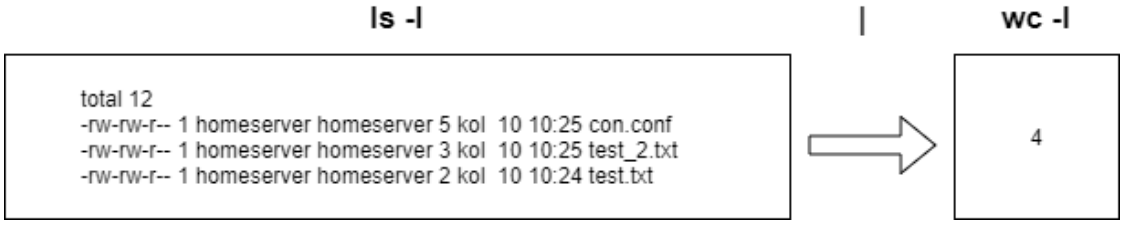
3.5.1. Cjevovodi

Cijevi se koriste za spajanje rezultata jedne naredbe s ulazom neke druge naredbe. Poopćeno, cjevovodi se koriste kako bi standardni tok izlaza prve naredbe postao standardni tok ulaza druge naredbe [1, str. 216]. Kod cjevovoda razlikujemo dvije vrste cijevi, anonimne i imenovane cijevi (eng. *named pipes*) [8].

Anonimne cijevi omogućavaju međuprocesnu komunikaciju te je njihova glavna specifičnost jednostavnost korištenja. Budući da su jednostavne za korištenje, njihov opseg funkcionalnosti je ograničen, pa tako one predstavljaju samo jednosmjerni tok podataka što bi značilo da ne postoji način povratne informacije. Kod *Unix* baziranih i Windows operacijskih sustava anonimne cijevi kreiraju se korištenjem simbola uspravne crte, odnosno „`|`“. Anonimne cijevi koriste se za komunikaciju između dretvi, odnosno procesa roditelja i djece. Kod *Unix* baziranih operacijskih sustava primjer korištenja anonimne cijevi dan je sljedećom naredbom:

```
ls -l | wc -l
```

Ova naredba se sastoji od tri dijela, naredbe `ls -l` koja ispisuje sadržaj trenutnog radnog direktorija u duljem formatu, anonimne cijevi sa simbolom `|` te naredbe `wc -l` koja ispisuje broj linija iz danog ulaza. Korištenjem cijevi, rezultat ispisa trenutnog radnog direktorija proslijeđen je kao ulaz naredbi `wc -l` koja tada ispisuje ukupan broj linija teksta dobivenog kao ulaz, što u ovom slučaju predstavlja izlaz naredbe `ls -l`. Bitno je napomenuti da se rezultat naredbe `ls -l` ne ispisuje na korisnički izlazni tok, jedini ispis je rezultat naredbe `wc -l` nakon što joj je proslijeđen rezultat naredbe `ls -l`. Grafički prikaz toka dan je sljedećim primjerom vlastite izrade.



Slika 3. Grafički prikaz toka rezultata i ulaza naredbi korištenjem cijevi

Imenovane cijevi omogućuju međuprocesnu komunikaciju između više procesa koji koriste istu cijev. Za razliku od anonimnih cijevi, imenovane cijevi traju dulje od vremena postojanja procesa. Nadalje, glavna razlika između anonimnih i imenovanih jest u tome da imenovane cijevi kreiraju datoteku koja predstavlja tu cijev [9]. U *Unix* operacijskim sustavima imenovana cijev kreira se korištenjem sljedeće naredbe:

```
mkfifo <naziv imenovane cijevi>
```

Kasnije, ta ista cijev se referencira putem naziva danog kao parametar naredbe mkfifo. Bitno je napomenuti da se cijev mora referencirati putem pune putanje na disku, odnosno svojim nazivom ako se naredba izvršava u direktoriju u kojem je cijev kreirana. Primjer korištenja imenovane cijevi može biti zadan sljedećim primjerom:

1) `mkfifo imenovana_cijev`
   a) Naredba izvršena u bilo kojem prozoru naredbenog retka: 1, 2 ili 3
2) `echo „123" > imenovana_cijev`
   a) Naredba izvršena u prozoru naredbenog retka 1.
3) `echo „ 456" > imenovana_cijev`
   a) Naredba izvršena u prozoru naredbenog retka 2.
4) `cat < imenovana_cijev`
   a) Naredba izvršena u prozoru naredbenog retka 3.

Kao rezultat, u prozoru naredbenog retka 3 pojavit će se sljedeći rezultat:

```
456
123
```

Kod ovog slijeda naredbi, prva naredba kreira imenovanu cijev naziva `imenovana_cijev` u trenutnom radnom direktoriju. Druga i treća naredba pišu podatke u imenovanu cijev naziva `imenovana_cijev`, dok četvrta naredba čita podatke iz iste imenovane cijevi.

Bitno je istaknuti da izvršavanjem naredbi pod 2. i 3. možemo uočiti da su naredbe blokirane prije nego što se izvrši naredba pod 4. Vaught navodi da jezgra operacijskog sustava obustavlja procese tako dugo dok se cijev ne zatvori [9]. Drugim riječima, programi koji pišu podatke u cijev obustavljeni su tako dugo dok se isti ne pročitaju iz cijevi.

4. Skriptiranje

Skriptiranje predstavlja korištenje skriptnog programskog jezika kako bi se izvršili razni zadaci koje osoba mora ručno obavljati. Odlika skripti je da one nisu ogromni kompleksni programi, već male zasebne datoteke koje obavljaju samo svoju ciljanu radnju. Za razliku od kompajlerskih jezika, skriptni jezici najčešće su interpretirani tijekom izvođenja. Skriptni jezici predstavljaju jezike visoke razine apstrakcije koji aktivno apstrahiraju koncepte kao što su upravljanje memorijom i rad s pokazivačima. Iako nije pravilo, skriptni jezici su najčešće dinamički tipovi programskih jezika dok su kompajlerski najčešće statički tipovi programskih jezika. Kod skripti nije naglasak na najbržoj mogućoj brzini izvođenja niza instrukcija već na olakšanom i ubrzanom razvoju automatizacijskih rješenja. Za skriptne jezike specifično je da se oni osim u razvoju novih samostalnih skripti, koriste za interakciju s programima i kompleksnim sustavima u posebnim skriptnim jezicima kreiranim od strane razvojnog tima koji je izgradio neki kompleksniji sustav. Takvi skriptni jezici i poopćeno skriptiranje u tim okruženjima osmišljeno je za proširivost originalnog sustava s novim funkcionalnostima koje su specifične ovisno o potrebama osoba koje razvijaju proširenja. Već spomenuta prednost ubrzanog razvoja najbolje se može dočarati usporedbom implementacije jednostavnog programa u C++ i Python programskim jezicima gdje je jedina funkcionalnost množenje dva korisnički unesena broja. C++ primjer dan je isječkom koda 1, Python primjer je dan isječkom koda 2.

```
#include <iostream>
using namespace std;
int main()
{
    int prvi_broj, drugi_broj;
    cout << "Unesite prvi broj: ";
    cin >> prvi_broj;
    cout << "Unesite drugi broj: ";
    cin >> drugi_broj;
    int rezultat = prvi_broj * drugi_broj;
    cout << "Rezultat množenja: " << rezultat;
    return 0;
}
```

Isječak koda 1. C++ implementacija jednostavnog programa množenja

```
prvi_broj = int(input("Unesite prvi broj: "))
```

```
drugi_broj = int(input("Unesite drugi broj: "))
rezultat = prvi_broj * drugi_broj
print("Rezultat množenja:", rezultat)
```

Isječak koda 2. Python implementacija jednostavnog programa množenja

Odmah je uočljiva prednost brzine izrade rješenja u Python primjeru, sa samo 4 linije koda naspram 13 potrebnih za C++ rješenje. Iako kod primjera danog u isječcima iznad nije primjetno, performanse kompajlerskih jezika, a specifično jezika baziranih na C programskom jeziku, neusporedivo su bolje od skriptnih jezika.

Iako se specifično navodi za Clojure, možemo poopćiti da je za skriptne jezike, karakteristična mogućnost upotrebe REPL okruženja, odnosno *read-eval-print loop* okruženja [10]. Takva okruženja prate sljedeće korake pri realizaciji svoje funkcionalnosti, gdje je dani primjer specifičan za Python interaktivna sučelja:

1) *Read* funkcionalnost
    a) Prihvaćanje korisničkog unosa te pretvorba istog u odgovarajuću strukturu podataka
        1. Primjer: korisnik unosi tekst „32 + 42“
2) *Eval* funkcionalnost
    a) Izvršavanje ili kalkulacija dane naredbe
        1. Primjer: naredba 32 + 42 se izvršava te se izračunava vrijednost 74
3) *Print* funkcionalnost
    a) Ispisivanje rezultata naredbe
        1. Primjer: ispisuje se vrijednost 74, koja je izračunata u Eval dijelu
4) *Loop*
    a) Vraćanje na *Read* funkcionalnost

REPL okruženje omogućava brzo prototipiranje i brzu provjeru manjih, nekompleksnih isječaka programskog koda. Očita je prednost u samoj brzini jer se ne treba kreirati nova datoteka, pokretati nova instanca interpretera te izvršiti kompletnu skriptu u jednom potezu. Daljnja prednost je u brzom otkrivanju potencijalnih grešaka radi mogućnosti provjere pojedinačnih komponenta složenijeg skriptnog rješenja.

4.1. Skriptiranje u Windows operacijskom sustavu

Skriptiranje u Windows operacijskom sustavu realizirano je putem tri osnovna interpretera naredbenog retka: *COMMAND.COM*, *cmd* i *PowerShell*. *COMMAND.COM* je prvi interpreter kojeg je Microsoft koristio u svojim operacijskim sustavima, s prvim izdanjem u 80-im godinama prošlog stoljeća u MS-DOS operacijskom sustavu. On se koristio sve do Windows Millenium Edition verzije operacijskog sustava [11]. Nudio je dva načina rada, interaktivni koji vraća rezultat naredbe nakon što je izvršena i *batch* način rada koji izvršava skriptu sa `.BAT` ekstenzijom gdje je skripta sadržavala jednu ili više slijednih naredbi. Neke od naredbi koje su se mogle izvršavati na *COMMAND.COM* interpreteru dane su sljedećom tablicom [12]:

Tablica 1. Naredbe *COMMAND.COM* interpretera (prema: [12])

Komanda	Opis
chdir	Prikazuje trenutni radni direktorij ili mijenja isti na novo specificirani direktorij koji je zadan u argumentu poziva naredbe
ctty	Mijenja ulazno/izlazni uređaj na pomoćni uređaj
date	Prikazuje trenutni datum ili mijenja isti na novo specificirani datum koji je zadan u argumentu poziva naredbe
dir	Prikazuje sadržaj trenutnog radnog direktorija ili sadržaj nekog drugog koji je zadan u argumentu poziva naredbe
echo	Prikazuje tekstualnu poruku
exit	Prekida rad interpretera naredbenog retka
rmdir	Briše poddirektorij

Sljedeći interpreter kojega je Microsoft razvio kao glavno sučelje za rad s Microsoft Windows NT operacijskim sustavima je *cmd* (eng. *Command Prompt*). Inicijalno je predstavljen u Windows NT liniji operacijskih sustava [11]. Iako vrlo sličan *COMMAND.COM* interpreteru, *cmd* proširuje postojeću funkcionalnost i dodaje novine, kao što su automatska dopuna teksta kod pisanja naziva datoteka i direktorija, povijest upisanih komandi po kojoj se može kretati pomoću strelica na tipkovnici te bolji opis poruka pogrešaka uslijed loše upisane naredbe. Kao što je već spomenuto, *cmd* nadograđuje mogućnosti *COMMAND.COM*, pa tako podržava naredbe iz prethodne implementacije te dodaje nove vezane uz novu funkcionalnost koja se razvijala kako se Windows NT razvijao.

Najnoviji interpreter koji je Microsoft razvio jest *PowerShell*. Pojavio se prvi puta unutar Windows XP operacijskog sustava 2006 godine pod nazivom *PowerShell* 1.0 [13]. Trenutna zadnja verzija je *PowerShell* 7.1 izdan 2020 godine. Isto kao i kod *cmd*-a, u *PowerShell*-u je moguće izvršavanje naredbi direktno u naredbenom retku kao i nizanjem jedne ili više naredbi u .ps1 datoteku.

4.1.1. *Batch* skripte

Batch skriptiranje predstavlja pokretanje datoteka koje u sebi imaju niz naredbi koje interpreter naredbenog retka mora izvršiti. Interpreteri za *batch* skripte su *COMMAND.COM* ili *cmd*.exe. Unutar skripti mogu se nalaziti sve komande koje interpreter može izvršiti, no mogu se pojaviti i programski konstrukti kao što su selekcija i iteracija. Kao i svim ostalim skriptama, glavna zadaća *batch* skripta je bila omogućiti i olakšati automatizaciju zadataka. *Batch* datoteke imaju ekstenzije `.bat` i `.cmd` [14]. Primjer *batch* skripte sa `.cmd` ekstenzijom pokrenute unutar Windows NT dan je isječkom koda 3.

```
@ECHO OFF
SET /p unesen_tekst=Unesite neki tekst:
ECHO Uneseni tekst je: %unesen_tekst%
```

Isječak koda 3. Primjer jednostavne *batch* skripte

U ovoj skripti, kao i kod svih ostalih, naredbe se izvršavaju redak po redak, pa tako u prvom retku `@ECHO OFF` služi da se ne ispiše tekst cijele neizvršene skripte, u drugom retku koristeći `SET` i argument `\p` za pomoćni tekst varijabli `unesen_tekst` dodjeljujemo vrijednost koju korisnik upiše u naredbeni redak. Kao zadnji korak ispisujemo predefinirani tekst zajedno s vrijednosti varijable `unesen_tekst`. U *batch* skriptama možemo koristiti i argumente prilikom pokretanja kao i već prije spomenuti programski konstrukt iterator. Skripta koja koristi argumente i for iterator dana je sljedećim isječkom koda:

```
@ECHO OFF
ECHO Vrijednost prvog argumenta: %1
FOR /l %%x in (1, 1, %1) DO (
    ECHO Neka poruka
)
```

Isječak koda 4. Korištenje argumenata i for petlje

U ovom isječku skriptu započinjemo sa `@ECHO OFF` za ignoriranje ispisa cijele skripte. Na drugoj liniji ispisujemo vrijednost prvog argumenta kojeg smo naveli prilikom pozivanja

same *batch* skripte. Na daljnjim linijama pomoću for petlje ispisujemo poruku u novom redu onoliko puta koliko je vrijednost prvog argumenta poziva skripte. Primjer pokretanja ove skripte je dan sljedećom naredbom `argument_iteracija.cmd 3` dok je rezultat izvođenje skripte s takvim pozivom dan sljedećim izlazom:

```
Vrijednost prvog argumenta: 3
Neka poruka
Neka poruka
Neka poruka
```

4.1.2. *PowerShell* skriptiranje

Kao što je već navedeno, treća i zadnja generacija interpretera naredbenog retka u Microsoft operacijskim sustavima jest *PowerShell*. Microsoft na svojoj stranici navodi da je *PowerShell* više-platformski alat za automatizaciju zadataka sačinjen od ljuske naredbenog retka, skriptnog jezika i okvira za upravljanje konfiguracijama [15]. Također navodi da se *PowerShell* može pokrenuti na Windows, Linux i MacOS operacijskim sustavima. Za novije verzije specifično je da koristi .NET Core razvojni okvir. *PowerShell* je specifičan u odnosu na sve ostale ljuske operacijskih sustava u tome da prihvaća i radi sa .NET objektima. Drugim riječima, *PowerShell* je objektno orijentirani skriptni jezik. Novina koja je uvedena u *PowerShell*-u naspram *cmd.exe* jest *cmdlets* (eng. izgovor *command-lets*), odnosno *PowerShell* komande. Microsoft na svojim stranicama dokumentacije ističe da su *cmdlet*-ovi izvorne *PowerShell* komande, a ne zasebne izvršne datoteke [16]. One predstavljaju prethodno definirane komande koje se mogu učitati pomoću *PowerShell* modula. Specifično je da *cmdlet*-ovi mogu biti napisani u bilo kojem .NET jeziku kao i *PowerShell* skriptnom jeziku. Predefinirani *cmdlet*-ovi generirani od strane *PowerShell* razvojnog tima prate Glagol-Imenica nazivlje, te je primjer takve naredbe `Get-Command`. Primjer jednostavne skripte napisane u *PowerShell* dan je sljedećim isječkom koda:

```
$unesen_tekst = Read-Host -Prompt "Unesite neki tekst"
Write-Output "Uneseni tekst je: $unesen_tekst"
```

Isječak koda 5. Jednostavna *PowerShell* skripta

U ovoj skripti, korisniku se korištenjem parametra `-Prompt` prikazuje tekst koji opisuje traženu radnju unosa teksta, te isti zatim sprema u varijablu. U sljedećoj liniji koda ispisuje se uneseni tekst zajedno sa predefiniranim tekstom. Uočljivo je da je sintaksa i sama čitljivost koda drastično poboljšana u *PowerShell*-u naspram *cmd.exe* koda za istu funkcionalnost.

Programski kod koji čita argumente zadane prilikom pokretanja *PowerShell* skripte i koristi iteraciju ovisno o argumentu dan je sljedećim isječkom koda:

```
$prvi_argument = $args[0]
Write-Output "Vrijednost prvog argumenta: $prvi_argument"
for ($i = 0; $i -lt $prvi_argument; $i++) {
    Write-Output "Neka poruka";
}
```

Isječak koda 6. *PowerShell* skripta koja čita ulazni argument i koristi for petlju

Skripta započinje dodjeljivanjem vrijednosti prvog argumenta zadanog pri pozivanju skripte varijabli `prvi_argument`. U drugoj liniji tu novu varijablu koristimo kako bismo je ispisali zajedno s predefiniranim tekstom. U sljedećim linijama imamo relativno standardni oblik for petlje kojim ispisujemo neki tekst onoliko puta koliko smo zadali vrijednost prvog argumenta. Primjer pokretanja ove skripte je dan sljedećom naredbom `./argument_iteracija.ps1 3` dok je rezultat izvođenje skripte s takvim pozivom dan sljedećim izlazom:

```
Vrijednost prvog argumenta: 3
Neka poruka
Neka poruka
Neka poruka
```

Isto kao i kod prethodne skripte, vrlo je očita veća razina čitljivosti koda nego kod *cmd.exe* skripte koja obavlja isti zadatak. Bitno je naglasiti da prilikom pokretanja *PowerShell* skripti, moramo koristi `./` ispred naziva skripte inače dobivamo sljedeću poruku:

```
The command argument_iteracija.ps1 was not found, but does exist in the current location. Windows PowerShell does not load commands from the current location by default
```

4.2. Skriptiranje u Linux operacijskom sustavu

Skripte kod *Unix* baziranih operacijskih sustava imaju iste karakteristika kao i skripte općenito, odlikuje ih jednostavnost korištenja za automatizaciju raznih zadataka. Skripte se u *Unix* operacijskim sustavima pokreću putem raznih implementacija *Unix* ljuske. Kao što je već bilo spomenuto, standard značajki koje ljuske imaju zadan je prema *Bourne Shell*, odnosno *sh* ljuski. *sh* ljuska, kao što je već bilo spomenuto, razvijena je krajem 70-ih godina prošlog stoljeća. Neke od danas često korištenih ljuski koje počivaju na standardu značajki *sh* ljuske su *Bourne-Again shell* (*bash*), *C shell* (*csh*), *Z shell* (*zsh*), *Secure Shell* (*ssh*), te mnogo drugih open-source, ali i komercijalnih poput *KornShell* (*ksh*) [17]. Neke od standardnih mogućnosti koje ljuske moraju implementirati za izvođenje skripti dane su tablicom.

Tablica 2. Osnovne mogućnosti koje ljuske implementiraju

Mogućnost	Opis	Primjer
Komentari	Komentari započinju sa „hash“ znakom na početku reda i nastavljaju se do kraja istog reda. Komentari su ignorirani od strane ljuske	`# Naziv autora: XYZ`
Izbor interpretera	Eng. shebang, predstavlja posebnu vrstu komentara koji daje direktivu sustavu koju ljusku mora koristiti kako bi izvršio skriptu. Shebang mora biti prva linija skripte, te mora započeti sa „#!“.	`#!/bin/bash` `# ostatak programa...`

Konstrukti u programiranju	Interpreteri bi trebali podržavati konstrukte kao što su selekcije, iteracije, varijable, nizovi podataka koji mogu biti indeksirani i asocijativni i slično. Omogućava kreiranje skripti koji su proširenijeg skupa mogućnosti nego samo izvršavanje naredbe.	`#!/bin/sh`
		`n=1`
		`while [ $n -le 10 ]`
		`do`
		`echo "Neki tekst"`
		`n=$(( $n + 1 ))`
		`done`

Izvršavanje naredbi dostupnih operacijskom sustavu	Skripte moraju dozvoljavati pokretanje naredbi koje operacijski sustav prepoznaje.	`#!/bin/sh`
		`clear`
		`date`

Prihvaćanje argumenata pri pokretanju	Prilikom poziva skripte, ljuske moraju prihvatiti, odnosno spremiti vrijednosti koji su zadane kao parametri prilikom pokretanja skripte.	Skripta
		`#!/bin/sh`
		`echo $1`
		Poziv skripte:
		`./skripta 123`
		Rezultat izvršavanja:
		`123`

Kako bismo mogli skripte pokretati bez potrebe specificiranja ljuske, a samim time i zapravo koristili funkcionalnost *shebang* linije, potrebno je novo kreiranoj tekstualnoj datoteci

pridružiti prava izvršavanja. Uzmimo za primjer da smo kreirali tekstualnu datoteku naziva `moja_skripta`, koja ima unesen tekst skripte. Kako bismo izbjegli potrebu pokretanja skripte s izričitim navođenjem interpretera, kao npr. `bash moja_skripta`, možemo izvršiti naredbu `chmod +x moja_skripta`, te sada istu skriptu možemo pozivati klasično kao izvršnu datoteku sa `./moja_skripta`. Kao što je definirano u tablici iznad, skripte moraju podržavati limitirani skup konstrukata programiranja da bi proširile mogućnosti koje se mogu ostvariti skriptom. Jednostavni primjer *sh* skripte dan je isječkom koda 7.

```
#!/bin/sh
echo "Unesite neki tekst: "
read unesen_tekst
echo "Uneseni tekst je: $unesen_tekst"
```

Isječak koda 7. Primjer jednostavne *Shell* skripte

Skriptu započinjemo *shebang* linijom u kojoj specificiramo da želimo da se skripta izvršava pomoću *sh* ljuske. U narednoj liniji pomoću naredbe `echo` ispisujemo tekst na ekran korisnika. Pomoću naredbe `read` tekst, koji se unosi sve do prelaska u novi red tipkom `Enter`, spremamo u varijablu naziva `unesen_tekst`. U zadnjoj liniji ponovno koristimo naredbu `echo` kako bismo ispisali predefiniran, zajedno sa dinamički unesenim tekstom na ekran korisnika. Skripta koja koristi argumente i for iterator dana je isječkom koda 8.

```
#!/bin/bash
echo "Vrijednost prvog argumenta: $1"
for (( n=0; n<$1; ++n ))
do
 echo "Neka poruka"
done
```

Isječak koda 8. Korištenje argumenata i for petlje u *bash* ljusci

Skriptu započinjemo *shebang* linijom u kojoj iskazujemo da želimo da se skripta izvrši pomoću *bash* ljuske. Koristimo *bash* umjesto *sh* radi nepostojanja podrške za sintaksu for petlje u *sh*. U sljedećoj liniji ispisujemo vrijednost prvog argumenta koristeći `$1`. U daljnjim linijama koristimo for petlju kako bi ponovili ispis vrijednosti „Neka poruka“ na ekran onoliko puta koliko smo zadali numeričku vrijednost prvog argumenta pri pokretanju skripte. Primjer

pokretanja ove skripte je dan sljedećom naredbom „`./argument_iteracija 3`“ dok je rezultat izvođenje skripte s takvim pozivom dan sljedećim izlazom:

```
Vrijednost prvog argumenta: 3

Neka poruka

Neka poruka

Neka poruka
```

4.3. Usporedba Windows i Linux skriptiranja

Već je ranije napomenuto da je trenutno zadnja verzija Microsoftovog interpretera naredbenog retka za operacijski sustav *PowerShell* 7.1, dok je najpopularniji i najčešće korišteni interpreter za operacijski sustav kod *Unix* operacijskih sustava *bash*. Stoga, ima smisla uspoređivati *PowerShell* sa *bash*-om kada govorimo o usporedbi Windows i Linux skriptiranja. Fundamentalna razlika između njih jest u tome da *PowerShell* koristi objektno orijentirani pristup, dok je *bash* potpuno baziran na tekstualnom prijenosu podataka. Uzevši tu razliku u obzir, sljedeći primjer prikazuje razliku između naredbe `dir` u *PowerShell*-u i `ls -l` u *bash*-u. Prilikom izvršavanja obiju naredbi, možemo vidjeti vrlo sličan format rezultata s nekim osnovnim informacijama, no tu sličnosti završavaju. Obzirom da je *PowerShell* objektno orijentiran, svaki redak informacije o sadržaju direktorija je objekt koji ima brojna svojstva i metode. U *bash*-u sve što dobijemo je tekst, jedino što dodatno možemo na temelju izlaza jest drugačije ga formatirati manipulirajući tekst. Kod *PowerShell*-a, možemo izvršiti isječak koda 9.

```
$direktorij=dir

$je_direktorij = $direktorij[0] -is [System.IO.DirectoryInfo]

if ($je_direktorij) {

    New-Item -Path $direktorij[0].Name -Name "testfile.txt" -
ItemType "file" -Value "Neki tekst"

}
```

Isječak koda 9. Primjer OOP kod *PowerShell*-a

U prvom retku ove skripte rezultat naredbe `dir`, koji je niz objekata koji predstavljaju datoteke i direktorije, dodjeljujemo varijabli direktorij. Uzevši u obzir da je rezultat niz objekata, tada u sljedećoj liniji provjeravamo ako je tip prvog objekta iz niza jednak tipu direktorija. Ako

jest, tada kreiramo novu datoteku kojoj postavljamo vrijednost putanje na naziv direktorija iz prethodne linije pristupanjem svojstvu `Name` na instanci objekta tipa `System.IO.DirectoryInfo`.

4.3.1. Usporedba naredbi Windows i Unix ljuski

Budući da je *Unix* ljuska bila izvor inspiracija mnogim ljuskama iz drugih operacijskih sustava, tada možemo vidjeti slične nazive komandi između *Unix*-ovih i Microsoftovih *cmd* i *PowerShell* ljuski. Neke od komandi, njihovo različito nazivlje i kratak opis dan je tablicom 3.

Tablica 3. Usporedba naredbi *Unix* ljuske, *cmd* i *PowerShell*-a

Unix ljuska	cmd	PowerShell	Opis
ls	dir	Get-ChildItem	Izlist svih datoteka i direktorija u zadanom direktoriju odnosno trenutnom
cp	copy	Copy-Item	Kopiranje datoteka i direktorija u drugi direktorij
mv	move	Move-Item	Premještanje datoteka i direktorija u drugi direktorij
rm	del	Remove-Item	Brisanje datoteka ili direktorija
curl	curl	Invoke-WebRequest	Slanje zahtjeva za dohvaćanjem podataka s udaljene ili lokalne web adrese
pwd	cd	Get-Location	Ispis trenutnog direktorija
ps	tasklist	Get-Process	Ispis svih trenutno pokrenutih procesa
ping	ping	Test-Connection	Testiranje dostupnosti prema udaljenom ili lokalnom poslužitelju
cd	cd	Set-Location	Mijenjanje trenutnog radnog direktorija

Možemo upisati `Get-Command` u *PowerShell* i tada dobivamo punu listu svih *cmdlet*-ova koje *PowerShell* može izvršiti.

4.3.2. Podsustav Linuxa na Windows operacijskom sustavu

Već je ranije spomenuto da je *PowerShell* dostupan i na *Unix* operacijskim sustavima, no nativno nije moguće pokretati *Unix* ljuske na Windows operacijskim sustavima. Microsoft je 2016. godine predstavio "Windows Subsystem for Linux (WSL)" [18]. Prema Microsoftovim riječima, WSL pruža GNU/Linux okruženje, zajedno sa svim alatima koji proizlaze iz istog, direktno unutar Windows operacijskog sustava, bez potrebe za virtualizacijom ili konfiguriranjem novog operacijskog sustava. Nadalje, Microsoft na svojim stranicama definira dostupnost sljedećih mogućnosti [19]:

- Odabir distribucija GNU/Linux
- Mogućnost pokretanja standardnih alata naredbenog retka kao što su grep, sed, awk, itd.
- Pokretanje skripti u ljusci *bash* kao i aplikacije GNU/Linux naredbenog retka, kao što su:
  - Alati: `vim`, `emacs`, `tmux`
  - Programski jezici: NodeJS, Javascript, Python, Ruby, C/C ++, C#, F#, Rust, Go itd.
  - Usluge: SSHD, MySQL, Apache, lighttpd, MongoDB, PostgreSQL.
- Instaliranje dodatnog softvera pomoću vlastitog upravitelja distribucijskih paketa GNU/Linux.
- Pokretanje Windows aplikacija pomoću *Unix* baziranih ljuski naredbenog retka.
- Pokretanje GNU/Linux aplikacija u sustavu Windows.

Instalacija WSL-a je vrlo jednostavna, dovoljno je u *cmd* ili *PowerShell* ljusku upisati `wsl --install` i Windows će automatski instalirati WSL. Microsoft je predstavio nadogradnju u obliku WSL2, s glavnim razlikama kao što su drastično poboljšane performanse i korištenje potpune sistemske kompatibilnosti s Linux-om [5]. Takve promjene omogućene su radi korištenja potpune Linux jezgre s blagim izmjenama za potrebe WSL2.

4.4. Skriptiranje pomoću Python programskog jezika

Prema uvodnim riječima u Python dokumentaciji, Python je interpretiran, dinamičkog tipa, interaktivan i objektno orijentiran programski jezik [19]. Dinamičkog je tipa što znači da tipovi nisu statički definirani prilikom kreiranja skripti ili zadavanja naredbi. Prema intervjuu s Guidom van Rossumom, glavnim programerom i dizajnerom, razvoj Python započeo je 80-ih godina prošlog stoljeća [20]. Dvije glavne verzije bile su Python 2 i Python 3. Python 3 je zamijenio Python 2 verziju i Python 2 je proglašen zastarjelim te više ne prima nadogradnje od

Python razvojnog tima. Specifičnost Pythona je sveobuhvatna standardna biblioteka koja se isporučuje sa standardnom instalacijom Pythona. Glavna filozofija prilikom kreiranja Python programskog koda jest proširljivost funkcionalnosti istog [20]. Za Python je specifično da je strogo ovisan o uvučenosti koda, gdje s postavljenom krivom razinom uvučenosti skripta javlja grešku nemogućnosti pokretanja. Python se čvrsto drži svojih principa dizajna, do te razine da je u sam Python interpreter uključena skrivena poruka. Ako u Python interpreteru izvršimo naredbu import this, prikazuje se sljedeća poruka prenesena u cijelosti na engleskom jeziku:

„The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!“

Iz ove poruke, između ostalog, možemo izdvojiti da se prilikom dizajniranja, kreiranja odnosno pisanja Python koda mora voditi misao o urednosti koda, izričitosti naredbi, preferiranja jednostavnosti, izbjegavanja ugnježđivanja logike, prednosti oskudnog naspram gustog raspoređenog koda i bitnosti čitljivosti. Python može izvršavati naredbe u dva načina rada, kao interaktivni interpreter, gdje se prati REPL uzorak kao i izvršavanje niza naredbi spremljenih u jednu datoteku. Koristeći sveobuhvatnu Python dokumentaciju, u tablici 4. ćemo izdvojiti neke od najčešćih ključnih riječi odnosno simbola kojima se implementiraju osnovni konstrukti programskih jezika i dati njihov kratki opis [21].

Tablica 4. Ključne riječi i simboli korišteni u Python-u (prema: [21])

Ključna riječ ili simbol	Opis funkcionalnosti
`for`	Obavlja iteraciju nad objektom koji se može iterirati
`if`	Obavlja funkcionalnost selekcije na temelju logičkog rezultata
`while`	Iteracija s ispitivanjem logičke vrijednosti izraza nakon svake iteracije
`try, except, finally`	Kontrola pojavljivanja pogrešaka i hvatanje i manipulacija istima
`raise`	Uzrokuje pojavljivanje pogreške, odnosno ručno okidanje pogreške
`with`	Blok koda koji ima specifičan kontekst, ovisno o naredbi. Na primjer, često je korišten kod manipulacije datotekama, datoteka je otvorena unutar with bloka koda, dok se blok koda završi, datoteka se automatski zatvara
`break`	Prekida izvođenje iteracije (for ili while petlja)
`continue`	Nastavlja na sljedeći korak iteracije, ne izvršava naredbe koje se nalaze poslije naredbe continue
`del`	Briše vrijednost varijable, često korišteno kod asocijativnih nizova
`pass`	Predstavlja praznu naredbu, potrebno za stvaranje praznog bloka naredbi
`import`	Uključivanje modula iz drugih skripti u trenutni program. Mogu se uključivati varijable, funkcije i klase
`class`	Ključna riječ korištena pri kreiranju, odnosno definiranju nove klase
`=`	Dodjeljivanje vrijednosti varijabli
`==, !=, <, <=, >, >=`	Operatori usporedbe
`and, or, not`	Logički operatori
`None`	Predstavlja vrijednost tipa NoneType, analogno null vrijednosti u drugim programskim jezicima

Jednostavna skripta koja prikazuje osnove sintakse i semantike u Pythonu dana je isječkom koda 10.

```
unesen_tekst = input("Unesite neki tekst: ")
print(f"Uneseni tekst je: {unesen_tekst}")
```

Isječak koda 10. Jednostavna Python skripta

U ovoj skripti u prvoj liniji korisnika tražimo unos teksta prikazanom instrukcijskom porukom te istu unesenu vrijednost spremamo u varijablu unesen_tekst. U sljedećoj liniji koristimo naredbu print kako bismo ispisali poruku tipa string. U primjeru je specifično

korištenje „f-strings“, što u Pythonu predstavlja posebno formatirane vrijednosti tipa string. U Pythonu također možemo koristiti argumente prilikom pokretanja skripte, te u isječku koda 11. vidimo takav primjer.

```
import sys
prvi_argument = int(sys.argv[1])
print(f"Vrijednost prvog argumenta: {prvi_argument}")
for i in range(0, prvi_argument):
    print("Neka poruka")
```

Isječak koda 11. Korištenje argumenata i for petlje

Kako bismo mogli dohvatiti argumente zadane prilikom pokretanja, moramo uključiti biblioteku sys putem naredbe import. U sljedećoj naredbi drugi argument poziva spremamo u varijablu prvi_argument. Specifično je u odnosu na prijašnje primjere s Windows i *Unix* ljuskama da u Pythonu prvi element, na mjestu 0 u indeksiranom nizu, predstavlja naziv skripte koja je pokrenuta, dok je prvi argument dan indeksom 1. Također, dobiveni argument je tekstualnog tipa, pa isti moramo pretvoriti u cjelobrojni kako bismo ga mogli koristiti. U sljedećoj liniji koristimo f-strings kako bismo formatirano ispisali željeni tekst. Nadalje, koristimo ključnu riječ for kako bismo iterirali preko novo kreiranog objekta koji je rezultat naredbe range(0, prvi_argument). Range funkcija vraća sekvencu brojeva od 0 do N, gdje je N jednak vrijednosti koju unesemo. Primjer pokretanja ove skripte je dan sljedećom naredbom „`python argument_iteracija.py 3`“ dok je rezultat izvođenje skripte s takvim pozivom dan sljedećim izlazom:

```
Vrijednost prvog argumenta: 3
Neka poruka
Neka poruka
Neka poruka
```

4.4.1. Prednosti Pythona u odnosu na druge programske jezike

Već smo bili spomenuli kako je jedna od glavnih značajki Python programskog jezika to da je dinamički pisan jezik. Većina jezika koji su kompilirani statičkog su pisanja. Dinamički pisani programski jezici provjeru tipova podataka obavljaju tijekom samog izvođenja programa, dok statički pisani programski jezici istu provjeru obavljaju tijekom kompiliranja. Ako se napravila greška pridruživanja vrijednosti unutar programa, dinamički jezici će se bez problema inicijalno pokrenuti, no kada naiđu na tu grešku, tada će se program srušiti, odnosno pojavit će se greška koju treba pravilno obraditi. Za razliku od dinamičkih, kod statičkih jezika, kompajler će prilikom kompiliranja prepoznati da je dodjela vrijednosti kriva, odnosno da se

tipovi razlikuju te se program neće uspješno kompilirati. Budući da nije potrebno specificirati tipove podataka prilikom pisanja programa, programi u dinamičkim jezicima u pravilu se brže mogu napisati u odnosu na statičke. Sljedeća prednost Pythona naspram drugih jezika je vrlo kratka minimalno potrebna struktura za pokretanje programa. Ovdje istovremeno možemo govoriti o strukturi datoteka potrebnih da se program pokrene, kao i čistom broju linija potrebnim za izvođenje vrlo jednostavne naredbe. Mnogi statički pisani jezici prilikom samog kreiranja, odnosno tijekom razvoja programa, generiraju mnogo datoteka i pomoćnih direktorija, dok kod Pythona postoji samo jedna datoteka. Uzmimo za primjer jednostavan problem dobivanja korijena iz broja kojega korisnik unosi u konzolu prilikom pokretanja aplikacije. C# implementacija dana je isječkom koda 12, dok je Python implementacija dana isječkom koda 13.

```
using System;
namespace PrimjerCSharp
{
    class Program
    {
        static void Main()
        {
            string vrijednostUnosa;
            Console.WriteLine("Molim upišite broj za koji želite saznati njegov korijen: ");
            vrijednostUnosa = Console.ReadLine();
            double pretvorenUnos = double.Parse(vrijednostUnosa);
            double korijenUnosa = Math.Sqrt(pretvorenUnos);
            Console.WriteLine($"Korijen broja {pretvorenUnos} je {korijenUnosa}");
        }
    }
}
```

Isječak koda 12. C# kod

```
import math
vrijednost_unosa = input("Molim upišite broj za koji želite saznati njegov korijen: ")
vrijednost_unosa = float(vrijednost_unosa)
korijen_unosa = math.sqrt(vrijednost_unosa)
print(f"Korijen broja {vrijednost_unosa} je {korijen_unosa}")
```

Isječak koda 13. Python kod

Iako je većina striktno potrebnog koda u C# rješenju automatski generirana prilikom kreiranja novog projekta, i dalje Python vodi s daleko manje broja linija koda. Nadalje, iako možemo prije kompiliranja vidjeti ako smo pogriješili tip prilikom pisanja koda i dalje je moguća greška ako korisnik umjesto brojčanog zapisa unese neku vrijednost koja nije brojčana. U oba slučaja će se desiti pogreška, odnosno programi će se srušiti jer nije obavljena adekvatna provjera korisničkog unosa. Sljedeća stavka koja nije zanemariva jest sama kompleksnost korištenja C# za jednostavne zadatke, u primjeru iznad za trivijalno izračunavanje korijena iz broja, za razvoj programa ukupno je generirano 29 datoteka i 12 direktorija. Za Python rješenje, generirana je samo jedna datoteka.

4.4.2. Nedostaci Pythona u odnosu na druge programske jezike

Dinamički tipovi, što smo prije spominjali kao prednost Pythona, ujedno je i njegova mana. Dinamički tipovi su vrlo korisni za brza prototipiranja i vrlo jednostavne aplikacije, no kod većih aplikacija koje imaju više različitih komponenti, koje su u interakciji s vanjskim sustavima dinamičnost tipova može dovesti vrlo brzo do neodrživosti koda. Pri tome se primarno misli na održavanje uslijed promjena zahtjeva, odnosno prilikom promjene timova koji rade na istom programskom rješenju. Otkrivanje pogrešaka također je teže jer jedan rubni slučaj koji nije detektiran i obrađen prilikom testiranja može dovesti do rušenja cijele skripte, odnosno aplikacije u sustavu koji je u upotrebi i o kojem ovise druge komponente. Nastavno na dinamičnost Pythona, nedostatak je i slaba iskoristivost automatskog dovršavanja u alatima koji se koriste za razvoj softvera. Ako instancu objekta klase prosljeđujemo funkciji, tada uređivač koda unutar bloka funkcije ne zna kojeg je tipa ta varijabla koja je parametar. Tada to ponovno dovodi do potencijalnih greški krivog korištenja svojstava ili metoda nad tim objektom. Sljedeća boljka Pythona su performanse naspram kompiliranih jezika. Uzevši u obzir da je Python interpreterski jezik, tada se naredbe interpretiraju tijekom izvedbe programa. To dovodi do lošijih performansi naspram jezika koji su prethodno kompilirani u strojni jezik.

4.4.3. Python na različitim platformama

Prednost Pythona je u tome da isti kod na Windows računalu obavlja istu zadaću i na *Unix* računalu. Općenito Python nije ovisan na kojoj platformi radi, osim u slučajevima u kojima je specifično da neke biblioteke podržavaju samo neke platforme. U skriptama možemo tijekom izvođenja programa saznati na kojoj smo platformi i ovisno o platformi izvršiti različiti kod. Uzmimo za primjer sljedeću skriptu danu u isječku 14 koja ovisno o platformi ispisuje različitu poruku.

```
from sys import platform
if platform.startswith('win'):
    print("Windows")
if platform.startswith('lin'):
    print("Linux")
if platform.startswith('dar'):
    print("MacOS")
```

Isječak koda 14. Primjer dohvaćanja trenutne platforme

Iako u primjeru iznad samo ispisujemo poruku na kojem smo tipu operacijskog sustava, u stvarnim skriptama umjesto ispisa možemo ovisno o platformi učitati različite module s različitih lokacija na sustavu, čitati sadržaje datoteka na različitim mjestima i slično.

5. Primjeri

Prikazat ćemo različite primjere s implementacijama u *Shell*, *PowerShell* i Python programskom kodovima.

5.1. Selekcija i iteracija

Skripte najčešće moraju koristiti složenije konstrukte programiranja od osnovnih kao što je čitanje i pisanje podataka na konzolu. Osnovni konstrukti kao što su iteracija I selekcija najčešće se koriste u skriptama koje zahtijevaju neku logiku ovisno o rezultatima prethodnih radnji, odnosno ulaza. Za primjer korištenja konstrukta selekcije zadat ćemo da skripta mora korisnika upitati za unos brojčane vrijednosti, te ovisno o unesenom broju, mora ispisati da li je on manji od 50, veći od 50 ili jednak 50. Rješenje u Pythonu dano je isječkom koda 15, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 16 i 17.

```
varijabla_a = int(input("Unesite neku brojčanu vrijednost:\n"))
print(f"Unesena je varijabla: {varijabla_a}")
if varijabla_a > 50:
    print("Unesena vrijednost veca je od 50")
elif varijabla_a < 50:
    print("Unesena varijabla manje je od 50")
else:
    print("Unesena varijabla jednaka je 50")
```

Isječak koda 15. Python rješenje primjera selekcije

```
#!/bin/sh
echo "Unesite neku brojčanu vrijednost: "
read varijabla_a
echo "Unesena je vrijednost: $varijabla_a"
if [ $varijabla_a -gt 50 ] ; then
    echo 'Unesena vrijednost je veca od 50'
elif [ $varijabla_a -lt 50 ] ; then
    echo 'Unesena vrijednost je manja od 50'
else
    echo 'Unesena vrijednost je jednaka 50'
fi
```

Isječak koda 16. *Shell* rješenje primjera selekcije

```
$varijabla_a = Read-Host -Prompt 'Unesite neku brojčanu vrijednost'
Write-Output "Unesena je vrijednost $varijabla_a"
if ($varijabla_a -gt 50) {
    Write-Output "Vrijednost je veca od 50"
} elseif ($varijabla_a -lt 50) {
    Write-Output "Vrijednost je manja od 50"
} else {
    Write-Output "Vrijednost je jednaka 50"
}
```

Isječak koda 17. *PowerShell* rješenje primjera selekcije

Između tri isječka, vidljiva je razlika u tome kako Python ima najčitljiviju i smisleniju sintaksu. Dok *PowerShell* i *Shell* koriste `lt` i `gt` koji predstavljaju *less-than* i *greater-than*, Python koristi standardne operatore uspoređivanja koji imaju više smisla.

Za prikaz funkcionalnosti iteracije, zadat ćemo da skripta mora upitati i spremiti podatke od korisnika koliko puta želi ponoviti frazu i tekst same fraze. Ona tada mora tu frazu ispisati onoliko puta koliko je zadano te na kraju ispisati koliko je ukupno znakova fraze ispisano. Rješenje u Pythonu dano je isječkom koda 18, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 19 i 20.

```
broj_ponavljanja = int(input("Unesite broj ponavljanja: "))
fraza_za_ponavljanje = input("Unesite frazu za ponavljanje: ")
broj_znakova_fraze = len(fraza_za_ponavljanje)
brojac_znakova = 0
for _ in range(0, broj_ponavljanja):
    brojac_znakova = brojac_znakova + broj_znakova_fraze
    print(f"{fraza_za_ponavljanje} - Trenutno smo na {brojac_znakova} znakova fraze ispisano")
print(f"Ukupno je ispisano {brojac_znakova} znakova fraze na ekran")
```

Isječak koda 18. Python rješenje primjera iteracije

```
#!/bin/sh
i=0
echo "Unesite broj ponavljanja: "
read max
echo "Unesite frazu za ponavljanje: "
read fraza
broj_znakova_fraze=${#fraza}
```

```
brojac_znakova=0
while [ $i -lt $max ]
do
    brojac_znakova=$((brojac_znakova + broj_znakova_fraze))
    echo "$fraza - Trenutno smo na $brojac_znakova znakova fraze ispisano"
    i=$((i + 1))
done
echo "Ukupno je ispisano $brojac_znakova znakova fraze na ekran"
```

Isječak koda 19. *Shell* rješenje primjera iteracije

```
$broj_ponavljanja = Read-Host -Prompt "Unesite broj ponavljanja"
$fraza_za_ponavljanje = Read-Host -Prompt "Unesite frazu za ponavljanje"
$broj_znakova_fraze = $fraza_za_ponavljanje.Length
$brojac_znakova = 0
for ($i = 0; $i -lt $broj_ponavljanja; $i++) {
    $brojac_znakova = $brojac_znakova + $broj_znakova_fraze
    Write-Output "$fraza_za_ponavljanje - Trenutno smo na $brojac_znakova znakova fraze ispisano"
}
Write-Output "Ukupno je ispisano $brojac_znakova znakova fraze na ekran"
```

Isječak koda 20. *PowerShell* rješenje primjera iteracije

Između ova tri odsječka vidi se kako Python i *PowerShell* imaju najsmisleniju i pregledniju sintaksu, te imaju otprilike isti broj linija potrebnih za implementaciju. *Shell* implementacija je nešto kompleksnija i malo nepreglednija radi nemogućnosti korištenja for petlje u *sh* ljusci.

5.2. Usporedba performansi

Iako performanse često nisu primarna metrika po kojoj se skripte uspoređuju i vrednuju, korisno je znati kakve su one ovisno o implementacijama. Za primjer usporedbe performansi zadat ćemo da skripta mora ponavljati operaciju zbrajanja 10 milijuna puta te na kraju ispisati rezultat zbrajanja. Rješenje u Pythonu dano je isječkom koda 21, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 22 i 23.

```
import datetime
```

```
a = datetime.datetime.now()
count = 0
for i in range(1, 10000000):
    count = count + i
b = datetime.datetime.now()
c = b - a
print(count)
print(c.seconds)
```

Isječak koda 21. Python rješenje primjera usporedbe performansi

```
#!/bin/sh
i=1
max=10000000
count=0
while [ $i -lt $max ]
do
    count=$((count + 1))
    i=$((i + 1))
done
echo "$count"
```

Isječak koda 22. *Shell* rješenje primjera usporedbe performansi

Prilikom mjerenja vremena potrebnog za izvršavanje *Shell* skripte koristimo sustavu naredbu time. Stoga, za datoteku skripte naziva `mjerenje_performansi.sh` koristimo naredbu `time mjerenje_performansi.sh`, te dobivamo sljedeći ispis:

```
real    0m24,420s
user    0m24,380s
sys     0m0,012s
```

Uočljivo je da nam ova komanda daje realno vrijeme izvršavanja koje uključuje vrijeme od pozivanja naredbe pa do njenog završetka, kao i vremena provedenog u korisničkom kao i sustavskom načinu rada [24]. Za očitavanje ukupnog vremena izvršavanja korišteno je stvarno vrijeme ukupnog izvođenja (eng. *real*).

```
Measure-Command {
    $count=0
    for($i = 1; $i -lt 10000000; $i++)
```

```
    {
        $count = $count + $i
    }
    Write-Host $count
}
```

Isječak koda 23. *PowerShell* rješenje primjera usporedbe performansi

Prilikom testiranja, korištena je jednaka konfiguracija te su svi testovi bili pokrenuti na Ubuntu 18.04.5 LTS operacijskom sustavu. *PowerShell* 7.1.4 je bio instaliran koristeći službenu Microsoft verziju. Verzija Python interpretera je 3.6.9. Bitno je uočiti da kod Python i *PowerShell* testiranja ne uključujemo vrijeme početka pripremanja interpretera, već mjerimo isključivo vrijeme trajanja zbrajanja i ispisa rezultata. Kod *Shell* primjera kao što je i napomenuto, koristimo sustavsku naredbu time kako bi dobili ukupno trajanje izvršavanja skripte. Rezultati izvršavanja dani su u tablici 5.

Tablica 5. Rezultat mjerenja performansi

Interpreter	Vrijeme izvršavanja
Python	1 sekunda
Shell	24 sekunde
PowerShell	29 sekunde

Možemo uočiti da je Python daleko brži od *Shell*-a i *PowerShell*-a. Postoji mogućnost da bi *PowerShell* bio brži na pravoj Windows instalaciji, no ne bi trebao biti značajno brži.

5.3. Izlist sadržaja direktorija

Često je potrebno automatizirati radnje oko direktorija, bilo to kopiranje, brisanje ili obično ispisivanje sadržaja. Za primjer izlista sadržaja direktorija zadat ćemo da skripta mora ispisati nazive datoteka I direktorija unutar trenutnog radnog direktorija. Rješenje u Pythonu dano je isječkom koda 24, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 25 i 26.

```
import os
elementi_direktorija = os.listdir(".")
for element in elementi_direktorija:
```

```
    print(element)
```

Isječak koda 24. Python rješenje primjera izlista sadržaja direktorija

```
#!/bin/sh
for element_direktorija in "."/*
do
  echo "$element_direktorija"
done
```

Isječak koda 25. *Shell* rješenje primjera izlista sadržaja direktorija

```
$elementi_direktorija = Get-ChildItem
foreach ($element_direktorija in $elementi_direktorija) {
    Write-Output $element_direktorija.Name
}
```

Isječak koda 26. *PowerShell* rješenje primjera izlista sadržaja direktorija

Sva tri rješenja podjednaka su u broju linija koda potrebnim za implementaciju takvog zadatka, no uočljivo je da Python i *PowerShell* imaju čišći kod i same naredbe su smislenije. Kod *Shell*a koristimo tzv. „*glob wildcard characters*“ kako bi odabrali da želimo sve datoteke i direktorije u trenutnom. Kod *PowerShell*-a je to smislenije izvedeno pomoću *cmdlet*-a `Get-ChildItem` koji dohvaća sve elemente trenutnog radnog direktorija. Također, kod *PowerShell*-a možemo uočiti korištenje objektno orijentiranog programiranja gdje je rezultat `Get-ChildItem` niz objekata kroz koji kasnije iteriramo foreach petljom te unutar petlje pristupamo svojstvu Name svakog pojedinog objekta. Python implementacija za specificiranje trenutnog direktorija u funkciji `listdir` koristi znak „`.`“ što predstavlja trenutni direktorij. Funkcija `listdir` vraća tekstualni niz te u sljedećoj liniji kroz for petlju iteriramo kroz taj isti niz kako bi ispisali svaki element.

5.4. Komprimiranje sadržaja određenog direktorija

Komprimiranje sadržaja direktorija također je česta naredba koju korisnici žele automatizirati, najčešće za potrebe arhiviranja, odnosno stvaranja sigurnosnih kopija. Za primjer komprimiranja zadat ćemo da skripta mora komprimirati kompletan sadržaj direktorija “`zeljeni_direktorij`” u jednu komprimiranu arhivu naziva `moj_backup.zip`. Rješenje u

Pythonu dano je isječkom koda 27, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 28 i 29.

```
import shutil
shutil.make_archive("moj_backup", 'zip', "zeljeni_direktorij/")
```

Isječak koda 27. Python rješenje primjera komprimiranja sadržaja određenog direktorija

```
#!/bin/sh
zip -r moj_zip.zip "zeljeni_direktorij/"
```

Isječak koda 28. *Shell* rješenje primjera komprimiranja sadržaja određenog direktorija

```
Compress-Archive -Path  "zeljeni_direktorij/" -DestinationPath
"moj_zip.zip"
```

Isječak koda 29. *PowerShell* rješenje primjera komprimiranja sadržaja određenog direktorija

Možemo uočiti da su sve tri implementacije za izvršavanje takve funkcionalnosti vrlo jednostavne te su sve jednolinijske, osim Pythona koji mora uključiti modul `shutil`. Python koristi već navedeni modul `shutil` kako bi pozvao funkciju `make_archive` s odgovarajućim argumentima. *Shell* koristi sustavsku naredbu `zip` te također specificira odgovarajuće argumente. *PowerShell* koristi *cmdlet* Compress-Archive te kao i prethodne dvije implementacije *cmdlet*-u pridružuje odgovarajuće argumente. Rješenje u Pythonu dano je isječkom koda 30, dok je rješenje u *PowerShell* ljusci dani isječkom 31.

5.5. Međuprocesna komunikacija

Međuprocesna komunikacija podrazumijeva dijeljenje informacija između više procesa. Za primjer međuprocesne komunikacije zadat ćemo da skripta mora u datoteku `rezultat.txt` spremiti ukupan broj datoteka i direktorija unutar trenutnog radnog direktorija. Rješenje u Pythonu dano je isječkom koda 30, 31 i 32, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 33 i 34.

```
import subprocess
proces_listanja = subprocess.Popen(['python',
'python/listanje_direktorija.py'], stdout=subprocess.PIPE)
rezultat_listanja =
proces_listanja.communicate()[0].decode('utf-8')
```

```
    proces_brojanja = subprocess.Popen(['python',
'python/brojac_linija.py', rezultat_listanja],
stdout=subprocess.PIPE)
    rezultat = proces_brojanja.communicate()[0].decode('utf-
8').strip()
    with open("rezultat.txt", mode="w") as datoteka:
        datoteka.write(rezultat)
```

Isječak koda 30. Python koordinator skripta

```
import os
print("\n".join(os.listdir(".")))
```

Isječak koda 31. Skripta ispisa sadržaja trenutnog radnog direktorija

```
import sys
broj_linija = sys.argv[1].count("\r\n")
print(broj_linija)
```

Isječak koda 32. Skripta brojača linija dobivenog argumenta

```
#!/bin/sh
ls -ld * | wc -l > rezultat.txt
```

Isječak koda 33. Skripta *Shell* rješenja

```
    Get-ChildItem | Measure-Object -Line | Out-File -FilePath
"rezultat.txt"
```

Isječak koda 34. Skripta *PowerShell* rješenja

Rješenja u ljuskama operacijskih sustava značajno su kompaktnija pomoću korištenja anonimnih cijevi i redirekcije. Python implementacija sadrži uz skriptu koordinatora dodatne skripte koje predstavljaju funkcionalnosti analogne naredbama `ls -ld *` i `Get-ChildItem` u *Shell* i *PowerShell* ljuskama. Iako je kompaktnost poželjna kod jednostavnih naredbi, kod kompleksnijih zadataka gdje je potrebna filtracija i obrada rezultata jednog procesa prije nego ulazi u drugi, Python nudi lakše izvršavanje zadataka radi svoje punokrvnosti kao programski jezik te programerima puno prirodnije sintakse.

5.6. Višedretvenost

Višedretvenost podrazumijeva izvršavanje više zadataka u različitim dretvama istovremeno. Za primjer višedretvenosti zadat ćemo da skripta simulirati izvršavanje vremenski dugotrajnog zadatka više puta. Bez korištenja dretvi, ukupno vrijeme izvođenja skripte bilo bi jednako zbroju trajanja svih zadataka, dok pomoću dretvi vrijeme izvođenja skripte jednako je vremenu trajanja zadatka koji se najdulje izvršavao. Rješenje u Pythonu dano je isječkom koda 35, dok je rješenje u *PowerShell* ljusci dani isječkom 36.

```
import threading
import time
pocetak_mjerenja = time.time()

def dretva(index):
    print(f"Dretva {index}: Započeta")
    vrijeme_rada = 3
    print(f"Dretva {index}: Radi zadatak {vrijeme_rada} sekundi")
    time.sleep(vrijeme_rada)
    print(f"Dretva {index}: Gotova")

lista_dretvi = []
for index in range(0, 3):
    x = threading.Thread(target=dretva, args=(index,))
    x.start()
    x.join()
    lista_dretvi.append(x)

for dretva_instanca in lista_dretvi:
    dretva_instanca.join()

kraj_mjerenja = time.time()
delta = kraj_mjerenja - pocetak_mjerenja
print(f"Ukupno vrijeme izvođenja skripte: {delta} sekundi")
```

Isječak koda 35. Rješenje pomoću dretva u Pythonu

```
(Measure-Command {  1..3 | ForEach-Object -Parallel {
    Write-Output "Dretva $_ : Započeta"
    $vrijemeRada = 3
    Write-Output "Dretva $_ : Radi zadatak $vrijemeRada sekundi"
    Start-Sleep $vrijemeRada;
    Write-Output "Dretva $_ : Gotova"
} -AsJob | Wait-Job | Receive-Job
}).TotalSeconds
```

Isječak koda 36. Rješenje koristeći dretve u *PowerShell*-u

Iako je vidljivo da *PowerShell* rješenje ima manje linija koda, rješenje pomoću Pythona implementacija je čitljivija i razumljivija. Python rješenje koristi modul `threading` kako bi se implementirala funkcionalnost dretvi te modul time s funkcijom `sleep` za simuliranje trajanja zadatka. Kod *PowerShell*-a koristimo već ugrađene *cmdlet*-ove za implementiranje funkcionalnosti pozadinskih paralelnih zadataka [23]. Također koristimo `Measure-Command` *cmdlet* kako bismo mjerili vrijeme izvršavanja i putem svojstva `TotalSeconds` dobivamo vrijeme izvršavanja u sekundama. Obje skripte su se izvršile u vremenu oko tri sekunde, s tim da je Python skripta bila brža 100-150 milisekundi od *PowerShell*-a. Očit je izostanak *Shell* implementacije, te je to pripisano nedostatku potpore za dretve u *Shell* ljuskama na *Unix* baziranim operacijskim sustavima. Alternativa korištenju dretvi u *Shell* ljuskama je kreiranje novih procesa za paralelno izvođenje zadataka.

5.7. Proširena skripta

Dosadašnje skripte bile su limitiranog dosega funkcionalnosti i fokusirale su se isključivo na jednokomponentna rješenja. Često se pomoću skriptiranja mora obaviti više funkcionalnosti koje su međusobno ovisne jedna o drugoj. Kao primjer skripte proširenog opsega zadat ćemo da skripta mora dohvatiti sve datoteke koje su modificirane unutar određenog direktorija u zadnjih 24 sata. Dalje, potrebno je kreirati `.zip` arhivu tih datoteka. Tu arhivu je zatim potrebno prenijeti na *Amazon Simple Storage Service*, odnosno Amazon S3 uslugu. Nakon prijenosa `zip` datoteke, potrebno je poslati e-mail poruku obavijesti. Rješenje u Pythonu dano je isječkom koda u prilogu 1, dok su rješenja u *sh* i *PowerShell* ljuskama dani isječcima 37 i 38.

```
#!/bin/sh
find "test_direktorij/" -mtime -1 -printf "%p\n" > zadnjih_24h.txt
zip out.zip -@ < zadnjih_24h.txt
aws s3 cp out.zip s3://S3_BUCKET_NAZIV/
mail -s "Backup informacija" pvarga@foi.hr <<< "Zip datoteka
kreirana"
```

Isječak koda 37. Proširena skripta u Shell ljusci

```
$S3_KEY = "API_KLJUC"

$S3_SECRET = "API_TAJNA"

$S3_BUCKET = "BUCKET_NAZIV"

Set-AWSCredential -AccessKey $S3_KEY -SecretKey $S3_SECRET -StoreAs
NoviProfil

$datoteke = Get-ChildItem -Path "test_direktorij" -Recurse | Where-Object {

    $_.LastWriteTime -gt (Get-Date).AddDays(-1)

}

foreach ($datoteka in $datoteke) {

    Compress-Archive -Update $datoteka .\out.zip

}


Write-S3Object -BucketName $S3_BUCKET -File "out.zip" -Key "out.zip"


Send-MailMessage -From "Petar Varga <pvarga@foi.hr>" -To "Petar Varga
<pvarga@foi.hr>" -Subject "Backup informacija" -Body "Zip datoteka
kreirana"
```

Isječak koda 38. Proširena skripta u PowerShell ljusci

Kod ovih implementacija proširene skripte, primarno za primijetiti je dužina same skripte u broju linija koda. Apsolutni pobjednik je ovdje *Shell* skripta sa daleko manje linija koda nego što to imaju PowerShell i Python rješenja. U Shell verziji skripte, potrebno je prethodno instalirati *AWS Command Line Interface* te odraditi konfiguraciju koja podrazumijeva unošenje korisničkih podataka potrebnih za autentifikaciju [25]. Uz to, potrebno je instalirati i konfigurirati `mailutils` proširenje za Unix, gdje je kod Ubuntu-a najlakši način koristeći naredbu `sudo apt install mailutils` [26]. Pomoću naredbe `find` u direktoriju `test_direktorij/` uz korištenje parametra `-mtime -1` pretražujemo one datoteke koje su bile modificirane

tijekom zadnjih 24 sata. Datoteke ispisujemo pomoću formata `%p\n` gdje se ispisuje puna relativna putanja datoteka. Rezultat se sprema redirekcijom u tekstualnu datoteku iz koje se u sljedećoj liniji putem `zip` naredbe i parametra `-@` čita sadržaj i isti se arhivira u arhivu `out.zip`. Sljedeće dvije linije predstavljaju korištenje *AWS CLI* kao i korištenje naredbe `mail`. Kod *PowerShell* implementacije, potrebno je instalirati *AWS cmdlet*-ove te konfigurirati *AWS* kao i postavke za slanje email-a [27, 28]. U skripti je potrebno odraditi inicijalizaciju AWS podataka i konfiguracije. Nakon toga rekurzivno dohvaćamo sve datoteke unutar traženog direktorija kojima je datum zadnje modifikacije unutar 24 sata. Sve te datoteke dodajemo u arhivu `out.zip`. Kako bismo prenesli zip datoteku, koristimo *AWS cmdlet* `Write-S3Object`. Nakon toga koristimo cmdlet `Send-MailMessage` kako bismo poslali email obavijest o kreiranoj arhivi.

Na prvi pogled mogli bismo zaključiti da je *Shell* apsolutni pobjednik s obzirom na broj linija. No, daljnja modifikacija i održavanje *Shell* skripte bila bi puno teža nego što je to slučaj kod Python skripte. Za primjer možemo uzeti dodavanje zapisa o uspješnosti pojedinih radnji unutar kompletne skripte. Kod Python skripte budući da imamo kompletniju skriptu možemo dodatno uključiti `logging` modul te zapisivati informacije o točnom broju pronađenih datoteka, kada je skripta pokrenuta, koja je veličina zip datoteke, koja je lokacija arhive, itd. To je također moguće kod *Shell* i *PowerShell* skripta, no uz veći uloženi napor i više potrošenog vremena. Također, ako bismo željeli koristiti različite konfiguracije prilikom prijenosa na *AWS* ili prilikom slanja e-maila, kod *Shell* i *PowerShell* implementacije to predstavlja mijenjanje pojedinih sustavskih postavki ili dodatnu kompleksnost s dodanim naredbama. U Pythonu se isto može implementirati korištenjem konfiguracijske datoteke koja se učitava prilikom pokretanja skripte.

6. Prednosti i nedostaci Python-a u odnosu na *shell* skripte

U prethodnom poglavlju osvrnuli smo se na različite primjere skripata implementiranih u ljuskama operacijskih sustava kao i u Pythonu. S obzirom na to, možemo zaključiti da za razne svrhe postoje određene prednosti, kao i nedostaci.

6.1. Prednosti Pythona

Glavna prednost Pythona je kompletnost kao programski jezik. Nudi daleko više mogućnosti nego ljuske operacijskih sustava nativno nude. Kao što je bilo prikazano, na *Unix* ljuski nedostaje mnogo mogućnosti koje su dostupne nativno u Python programskom okruženju. Samo neke od prednosti koje Python ima naspram *Unix* ljuske su višedretvenost, korištenje složenih tipova podataka, vrlo jednostavna sintaksa, laka obrada pogrešaka, korištenje softvera za praćenje izvršenja programa (eng. *debugger*) i mnoge druge mogućnosti. Iako *PowerShell* ima više mogućnosti kao kompletan programski jezik i dalje nije na razini kompletnosti kao što je Python. *PowerShell*-ova sintaksa, iako ju je puno jednostavnije za razumjeti i koristiti nego sintaksu od *Unix*-ovih ljuski, ona i dalje nije na razini čitljivosti i jednostavnosti čitanja kao što je Pythonova. Također, neovisno da li promatramo *Unix* ili *PowerShell* ljusku, Python je mnogo brži u izvršavanju naredbi upravo zbog toga što je kompletan programski jezik. Iako *Unix* i *PowerShell* ljuske imaju aktivnu zajednicu koja koristi i daje upute za korištenje ljuski, Python zajednica je aktivnija i postoji detaljnija dokumentacija i više resursa za učenje nego što je to slučaj kod *Unix* i *PowerShell* ljuski.

6.2. Nedostaci Pythona

Ono što je prednost Pythona naspram ljuski operacijskih sustava, ujedno može biti prikazano i kao nedostatak. Pa tako iako je Python potpuni jezik i nudi pregršt mogućnosti i uključivanja biblioteka, za vrlo jednostavne zadatke nema potrebe uvoditi opće troškove u smislu kompleksnosti. Ono što u Pythonu možemo napisati u par linija koda, kod ljuski operacijskih sustava to često možemo napisati u manje linija. Nastavno na jednostavnije skripte, vrijeme početka izvođenja bolje je u ljuskama naspram Pythona, upravo iz tog razloga što Python interpreter ima veće opće fiksne troškove u smislu pokretanja procesa interpretera. Vrlo bitna prednost *bash* i *PowerShell* skripti je da one nativno mogu koristiti sustavske naredbe kao što su `cd`, `ping`, `zip` kod *Unix* ljuski, `Compress-Archive` kod *PowerShell* i mnoge druge.

7. Zaključak

Kroz završni rad protezala se tema primjenjivosti Pythona kao alternativnog jezika za izradu automatizacijskih skripti. Bilo je prikazano kako Python pruža prednosti u obliku skripti koje su lakše za izraditi, održavati i koristiti. Koristeći sve standardne mogućnosti koje Python nudi, možemo izraditi sve što bismo mogli napravili unutar ljuski operacijskih sustava. Ono što Python izdvaja su široka prilagodljivost raznim zadacima i proširljivost uključivanjem dodatnih modula za obavljanje raznih naredbi.

Dokumentacija i količina materijala dostupnog za učenje daleko je većeg opsega i količine u odnosu na ljuske operacijskih sustava. Također, puno je više članova zajednice koji se koriste Pythonom i međusobno si pomažu prilikom nailaska na probleme i greške u odnosu na *shell* i *PowerShell.*

Smatram da je prikazano da je Python realna alternativa te da će se više koristiti za zadatke koji su se do sada isključivo radili u ljuskama operativnih sustava. Za neke jednostavne zadatke se može primjetiti da nema smisla koristiti Python, no za kompliciranije zadatke, odnosno skup zadataka, Python se postavlja kao nadmoćnije rješenje.

Popis literature

[1] T. Adelstein i B. Lubanovic, *Administriranje linux sustava*, Zagreb: Dobar Plan. 2007

[2] The Linux Documentation Project (bez dat.), *Chapter 15. Internal Commands and Builtins* [Na internetu]. Dostupno: https://tldp.org/LDP/abs/html/internal.html [pristupano 22.08.2021]

[3] The Linux Documentation Project (bez dat.), *External Commands in the Prompt* [Na internetu]. Dostupno: https://tldp.org/HOWTO/Bash-Prompt-HOWTO/x279.html [pristupano 22.08.2021]

[4] Microsoft Docs, *about_Classes*, 2021. [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_classes?view=powershpow-7.1#class-properties [pristupano 22.08.2021]

[5] Microsoft Docs, *What is the Windows Subsystem for Linux?*, 2020. [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/windows/wsl/about [pristupano 22.08.2021]

[6] Dennis M. Ritchie, *A Stream Input-Output System*, 1984., [Na internetu]. Dostupno: https://cseweb.ucsd.edu/classes/fa01/cse221/papers/ritchie-stream-io-belllabs84.pdf [pristupano 22.08.2021]

[7] K–State Polytechnic (bez dat.), *File Redirection*, [Na internetu]. Dostupno: http://faculty.salina.k-state.edu/tim/unix_sg/shell/redirect.html [pristupano 22.08.2021]

[8] Microsoft Docs, *Pipe Operations in .NET*, 2017. [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/dotnet/standard/io/pipe-operations [pristupano 22.08.2021]

[9] Andy Vaught, *Introduction to Named Pipes*, 1997., [Na internetu]. Dostupno: https://www.linuxjournal.com/article/2156 [pristupano 22.08.2021]

[10] Clojure (bez dat.), *Programming at the REPL: Introduction*, [Na internetu]. Dostupno: https://clojure.org/guides/repl/introduction [pristupano 22.08.2021]

[11] Ben Stegner, *A Beginner's Guide to the Windows Command Prompt*, 2021., [Na internetu]. Dostupno: https://www.makeuseof.com/tag/a-beginners-guide-to-the-windows-command-line/ [pristupano 22.08.2021]

[12] Everett Murdock (bez dat.), *DOS Command Index*, [Na internetu]. Dostupno: https://web.csulb.edu/~murdock/dosindex.html [pristupano 22.08.2021]

[13] PowerShell Team, *It's a Wrap! Windows PowerShell 1.0 Released!*, 2006., [Na internetu]. Dostupno: https://devblogs.microsoft.com/powershell/its-a-wrap-windows-powershell-1-0-released/ [pristupano 22.08.2021]

[14] Microsoft Docs, *Using batch files*, 2009., [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb490869(v=technet.10) [pristupano 22.08.2021]

[15] Microsoft Docs, *What is PowerShell?*, 2021., [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.1 [pristupano 22.08.2021]

[16] Microsoft Docs, *What is a PowerShell command (cmdlet)?*, 2021., [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/powershell/scripting/powershell-commands?view=powershell-7.1 [pristupano 22.08.2021]

[17] O'Reilly (bez dat.), *Learning the bash Shell*, [Na internetu]. Dostupno: https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch01s03.html [pristupano 22.08.2021]

[18] Microsoft DevBlogs (bez dat.), *Learn About Windows Console & Windows Subsystem For Linux (WSL)*, [Na internetu]. Dostupno: https://devblogs.microsoft.com/commandline/learn-about-windows-console-and-windows-subsystem-for-linux-wsl/ [pristupano 22.08.2021]

[19] The Python Software Foundation (bez dat.), *General Python FAQ*, [Na internetu]. Dostupno: https://docs.python.org/3/faq/general.html#what-is-python [pristupano 22.08.2021]

[20] Bill Venners, *The Making of Python A Conversation with Guido van Rossum, Part I*, 2003., [Na internetu]. Dostupno: https://www.artima.com/articles/the-making-of-python [pristupano 22.08.2021]

[21] The Python Software Foundation (bez dat.), *More Control Flow Tools*, [Na internetu]. Dostupno: https://docs.python.org/3/tutorial/controlflow.html [pristupano 22.08.2021]

[22] Justinmind, *Navigation design: Almost everything you need to know*, 2020., [Na internetu]. Dostupno: https://www.justinmind.com/blog/navigation-design-almost-everything-you-need-to-know/ [pristupano 22.08.2021]

[23] Microsoft Docs, *about_Thread_Jobs,* 2020., [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_thread_jobs?view=powershell-7.1 [Pristupano 29.8.2021]

[24] Linux manual page (bez dat.), *time(1) — Linux manual page*, [Na internetu]. Dostupno: https://man7.org/linux/man-pages/man1/time.1.html [Pristupano 29.8.2021]

[25] AWS (bez dat.), *Batch upload files to the cloud to Amazon S3 using the AWS CLI*, [Na internetu]. Dostupno: https://aws.amazon.com/getting-started/hands-on/backup-to-s3-cli/

[26] Linux Hint, *Bash script to send email*, 2019., [Na internetu]. Dostupno: https://linuxhint.com/bash_script_send_email/

[27] Tod Hilton, *Upload/Backup your files to Amazon S3 with Powershell*, 2015., [Na internetu]. Dostupno: http://todhilton.com/technicalwriting/upload-backup-your-files-to-amazon-s3-with-powershell/

[28] Microsoft Docs (bez dat.), *Send-MailMessage*, [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/send-mailmessage?view=powershell-7.1

Popis slika

Popis tablica

Popis isječaka programskog koda

Prilog 1

```
import time
import os
import glob
from datetime import datetime
import zipfile
import boto3
import smtplib, ssl

S3_BUCKET = "BUCKET_NAZIV"
S3_KEY = "API_KLJUC"
S3_SECRET = "API_TAJNA"
S3_LOCATION = F'http://{S3_BUCKET}.s3.amazonaws.com/'
SENDER_EMAIL = "EMAIL"
SENDER_PASSWORD = "LOZINKA"

def dohvati_datoteke_modificirane_unutar_24h(direktorij="test_direktorij"):
    lista_nedavno_modificiranih = []
    rezultat = []
    for x in os.walk(direktorij):
        for y in glob.glob(os.path.join(x[0], '*.*')):
            rezultat.append(y)
    for datoteka in rezultat:
        path_datoteke = datoteka
        datum_zadnje_modifikacije =
datetime.fromtimestamp(os.path.getmtime(path_datoteke))
        if (datetime.now() - datum_zadnje_modifikacije).days == 0:
            lista_nedavno_modificiranih.append(path_datoteke)
    return lista_nedavno_modificiranih

def zipaj_datoteke(lista_nedavno_modificiranih):
    zip_file_path = 'out.zip'
    with zipfile.ZipFile(zip_file_path, 'w') as zip_objekt:
        for file in lista_nedavno_modificiranih:
            zip_objekt.write(file, compress_type=zipfile.ZIP_DEFLATED)
    return zip_file_path

def uploadaj_zip_s3(zip_path):
    s3 = boto3.client(
```

```
        "s3",
        aws_access_key_id=S3_KEY,
        aws_secret_access_key=S3_SECRET
    )
    s3.upload_file(
        zip_path,
        S3_BUCKET,
        zip_path,
        ExtraArgs={
            "ACL": "public-read",
            "ContentType": "application/zip"
        }
    )

def posalji_mail(message_text="Zip datoteka kreirana", subject_text="Backup
informacija", receiver_email="pvarga@foi.hr"):
    port = 465
    smtp_server = "smtp.gmail.com"
    sender_email = SENDER_EMAIL
    password = SENDER_PASSWORD
    message = f"""Subject: {subject_text}\n\n{message_text}"""
    context = ssl.create_default_context()
    with smtplib.SMTP_SSL(smtp_server, port, context=context) as server:
        server.login(sender_email, password)
        server.sendmail(
            sender_email, receiver_email, message
        )

lista_nedavno_modificiranih = dohvati_datoteke_modificirane_unutar_24h()
path_zipanog_fajla = zipaj_datoteke(lista_nedavno_modificiranih)
uploadaj_zip_s3(path_zipanog_fajla)
posalji_mail()
```