SKRIPTNI JEZICI

⇒ služe za jednostavnu automatizaciju repetitivnih operacija (između ostalih funkcionalnosti)

⇒ mi ćemo se baviti sa:

a) BASH LJUSKOM
b) programskim jezikom PERL
c) programskim jezikom PYTHON

⇒ ŠTO JE SKRIPTA?

↳ skripta je program vrlo visoke razine koj je najčešće vrlo kratak
↳ najčešće se bavi sa stringovima te datotečnim sustavom

⇒ za što se koriste skriptni jezici:

↳ za povezivanje više programa u cjelinu
↳ za intenzivnu manipulaciju teksta
↳ skripte su obično veoma portabilne, pa se mogu koristiti na Windowsu, Linuxu i Macu

⇒ prednost skriptnih jezika u odnosu na jezike opće namjene:

↳ kraći programi (brži razvoj)
↳ ne prevode se niti se povezuju (INTERPRETIRAJU SE)
↳ nema deklaracija (al puno provjera tijekom izvođenja)

⇒ KLASIFIKACIJA PROGRAMSKIH JEZIKA:

a) dodjela tipova

- statički (npr. C)

```
double c;
c = 5.2;
c = "abc"; // error
```

- dinamički (npr. Python)

```
c = 2;          # integer
c = [2, 3, 4] # lista integera
```

b) promjena tipova

- slabo tipizirani (npr. Perl, C++)

```
int a=2;
float b = 1.2F * a; // int->float
```

- jako tipizirani (npr. Python)

```
b = '1.2'
c = 5 *b # nema promjene tipa
         # rezultat je ponavljanje
         # niza pet puta
```

LJUSKA OPERACIJSKOG SUSTAVA

⇒ fleksibilno sučelje prema operacijskom sustavu
⇒ to je jezik koj se interpretira
⇒ služi za izvođenje programa (najčešće manjih programa - skripti)

⇒ ljuska nije dio jezgre!
↳ ona interpretira ulaz i prevodi ga u pozive jezgre OS-a

⇒ ljuske su standardizirane kako bi se osigurala kompatibilnost
⇒ ljuske obično podržavaju velik broj naredbi i omogućavaju brzo izvođenje tih naredbi

⇒ danas je najpopularnija ljuska BASH i time se ovaj kolegij bavi
↳ skraćenica: „Bourne Again Shell"
↳ pogledati čemu neke naredbe ljuske bash

⇒ neke naredbe:

- man ls ⇒ prikazuje „manual" ljuske
  ⇒ možemo pretraživat manual:
  / ključna_riječ
- cat ⇒ prikazuje datoteku (koristi za malu datoteku)
- more ⇒ prikaz datoteke ekran po ekran
- less ⇒ prikaz datoteke uz mogućnost „scrollanja"

⇒ redirekcija ispisa:

↳ koristimo poznati operator: >

↳ imamo dodatne opcije:

2> ⇒ preusmjeravanje dijagnostičkog ispisa

>> ⇒ nadodavanje u izlaznu datoteku

⇒ preusmjeravanje upisa:

↳ koristimo poznati operator: <

↳ nadovezivanje naredbi:

- operator: |
- na primjer: ls | more

⇒ uvjetno izvođenje naredbi:

↳ svaka naredba vraća status koji govori o uspješnosti izvođenja naredbi (tip mu je int)

↳ uspješna naredba vraća nulu!

↳ operatori: && i ||

⇒ varijable ljuske:

↳ nema tipova jer je ljuska namijenjena za lijepljenje, pa su gotovo sve varijable stringovi

↳ postavljanje vrijednosti varijable:

X="abcd"

↳ dereferenciranje (korištenje) varijabli:

echo $X ⇒ ispisuje sadržaj varijable X

↳ prikaz svih varijabli ljuske možemo dobit pomoću naredbe set

⇒ varijable okoline:

↳ poseban vrsta (podskup) varijabli ljuske

↳ neku varijablu možemo proglasit varijablom okoline naredbom export

↳ VARIJABLA $PATH:

- sadrži listu karala u kojima se traže izvršne datoteke

↳ postoj još nekoliko korisnih varijabli okoline (vidi slajdove!)

⇒ POKRETANJE LJUSKE:

- interaktivni rad ⇒ korisnik zadaje naredbe i dobiva rezultate

⇒ mjesto unosa je označeno odzivnim znakom (prompt)

- neinteraktivni rad ⇒ ljuska izvodi naredbe iz liste naredbi, tj. izvodi skriptu

⇒ KONCEPT ŠIRENJA (EXPANSION):

a) širenje vitičastih zagrada:

↳ mehanizam kojim se mogu generirati proizvoljni znakovni nizovi

↳ dolazi se prije ostalih širenja

a{b,c}d ⇒ 'abd acd'

b) širenje oznake matičnog imenika:

↳ ako riječ počinje sa ~, svi znakovi do / ili do kraja su tilda prefiks

c) širenje varijabli i parametara:

↳ varijable se evaluiraju u njihove vrijednosti

↳ unutar dvostrukih navodnika će se ovo širenje izvršiti

d) širenje aritmetičkih izraza:

↳ šire se nakon varijabli, pa u računu možemo koristiti varijable

e) širenje imena datoteka:

↳ koristi se jednostavan oblik regularnih izraza sa znakovima ?, *, [

⇒ SLOŽENIJI UNIX ALATI 8

1) wc ⇒ „word count"
⇒ broj riječi, redaka i znakova u datoteci il na standardnom ulazu

2) grep ⇒ služi za pretraživanje teksta po danom uzorku (regularnom izrazu)
⇒ redak u kojem se nađe dan uzorak ispisuje se na standardni izlaz
⇒ ovu naredbu ćemo puno koristiti

3) sed ⇒ „stream editor"
⇒ namijenjen filtriranju i sortiranju teksta

4) find ⇒ služi za traženje datoteka
⇒ napredna naredba sa mnoštvom opcija

5) locate ⇒ traži datoteke na temelju vlastitog indeksiranja
⇒ brža od naredbe „find"

REGULARNI IZRAZI

⇒ razlikuju se od uroraka lješka
↳ NE TO MIJEŠATI!

^ ⇒ lista se tumači kao negacija (unutar uglatih zagrada)
[„znakovi"] ⇒ ulazni niz se podudara sa bilo kojim znakom iz niza „znakovi"
⇒ raspon se radaje crticom „-" (npr. [0-9])

[:alnum:], [:alpha:], [:lower:], ... ⇒ neke od ugrađenih klasa znakova

. ⇒ podudara se sa bilo kojim pojedinačnim znakom
\w ⇒ sinonim za klasu [:word:]
$ ⇒ oznaka za kraj retka
\< ⇒ oznaka za početak riječi
\> ⇒ oznaka za kraj riječi
\b ⇒ odgovara granici riječi
\B ⇒ odgovara negaciji granice riječi (unutrašnjost)

⇒ nakon regularnog izraza slijede operatori ponavljanja:
? ⇒ najviše jednom
* ⇒ proizvoljno puta
+ ⇒ jednom ili više puta
{n, m} ⇒ barem „n", a najviše „m" puta

POKRETANJE I PISANJE SKRIPTI

⇒ primjer izrade skripte:

```
... $ cat hello.sh
echo "Hello World"
```

⇒ primjer pokretanja skripte:

```
... $ source hello.sh
```

↳ drugi način:

```
... $ . hello.sh
```

⇒ možemo koristiti i izravno pozv, al prij toga trebamo ljusc dat do znanja da se radi o tekstualnim datotekama, a ne o binarnim datotekama:

```
... $ chmod u+x
... $ ./hello.sh
```

⇒ PRENOŠENJE PARAMETARA U SKRIPTU:

↳ prenosimo ih preko pozicijskih argumenata

↳ ima i drugih načina, al je ovaj najsigurniji

⇒ IZLAZNI STATUS:

↳ povratna vrijednost funkcije main

↳ govori o uspješnosti izvođenja skripte

NAREDBE LJUSKE ZA UPRAVLJANJE PROGRAMSKIM TOKOM

⇒ uvjetno izvođenje:

```
if naredba_t
then
    blok_naredbi
fi
```

blok_naredbi će se izvršiti ako je naredba_t uspješno završila (vratila je 0)

⇒ o naredbi "test":

- koristi se za ispitivanje uvjeta u naredbi if
- npr. test "$name" = Vedrana
- podržava mnoštvo operatora koje je korisno znati
- specijalna sintaksa naredbe test:

  ["$dan" = Tuesday]

  ↳ ova sintaksa daje čitljiviji kod

⇒ postoje naredbe za kontrolu slične ostalim programskim jezicima:

- case
- for ; do ; done
- while ; do ; done
- until ; do ; done
- break n ⇒ izlaz iz „n" unutrašnjih petlji
- continue n ⇒ preskače „n" petlji i provjerava uvjet

UVOD U PROGRAMSKI JEZIK PERL

⇒ autor jezika: Larry Wall
  ↳ jezik osmišljen 1987. godine
  ↳ PERL → „Practical Extraction and Report Language"
    → „Pathologically Eclectic Rubbish Lister"

⇒ ideja je da se pisanje skripti olakša pomoću uvođenja dobrih stvari C-olike sintakse
  ↳ C, C++, assembler ⇒ niska razina
  ↳ ljuska OS-a ⇒ visoka razina

⇒ PERL je jedno vrijeme bio dominantan u domeni programiranja WEB stranica
  ↳ to je radio jer su WEB stranice samo HTML tekst, a Perl je odličan za manipuliranje tekstom

⇒ danas, Perl stagnira u popularnosti jer se Python uspeo u popularnosti

⇒ Perl podržava objektno-orijentirano programiranje
⇒ jezik podržava regularne izraze
⇒ podržava i mrežno programiranje

⇒ Perl je optimiran za rad s tekstom

⇒ PISANJE PERL PROGRAMA:

```
#!/bin/perl
print "Hello World!"
```

↳ Perl je jezik slobodne forme !!

⇒ TIPOVI PODATAKA:

a) SKALARI ⇒ brojčane vrijednosti i nizovi znakova

↳ znakovni nizovi u Perlu nemaju nul-terminator, pa znakovne nizove možemo koristiti kao niz bajtova (npr. učitavanje, izmjena i zapis binarne datoteke)

↳ interpolacija varijabli ⇒ analogno ekspanziji varijabli u ljusci

↳ nadovezivanje nizova: "."

↳ ponavljanje nizova: "x" (5 x 4 = 5555)

↳ pazi na implicitnu pretvorbu između brojeva i nizova (vidi slajd 18 !!)

↳ zbog implicitnih pretvorbi, postoje upozorenja (uključujemo opcijom -w)

↳ imena skalarnih varijabli započinju znakom „$", a zatim Perl identifikator

↳ skalare se mogu ispisati pomoću operatora / naredbe **print()**

nekad ne moramo navoditi ove zagrade

b) POLJA i LISTE ⇒ nalaze se u raselnom prostoru imena

⇒ pristup elementu liste radi se isto kao i pristup varijabl skalara (operator $)

⇒ indeks radnjeg elementa polja možemo dobit na način:

$#polje

⇒ prokrata za pristupanje radnjem elementu polja:

$polje[-1]

↳ negativni indeks, služe za indeksiranje od kraja polja

⇒ liste kao literal se navode unutar oblih zagrada:

(1, 2, 3), (1..5)

↳ npr. lista svih indeksa polja:

(0.. $#polje)

↳ prokrata za pisanje liste riječi:

("janko", "dana", "rud", "banana")

qw/ janko barney radi banana/

↖ „slash" nije jedini mogući graničnik!

qw ⇒ „quoted words"

⇒ referenciranje polja:

@polje

↑

pristup „namespace"-u za polja

⇒ NAPOMENA: element polja mogu biti samo skalari, pa ako u polju pišemo ime drugog polja, to drugo polje se „raspakira" / „ekspandira"

```
@polje = qw/ jen dva tri /
@drugo = ( @polje, "četiri" )
```

↑ ovo polje „drugo" sada ima elemente ("jen", "dva", "tri", "četiri"), a nema referencu na @polje

↳ promjena nad @polje neće utjecati na @drugo (ne pamti se referenca ☹)

⇒ NAPOMENA: u Perl-u postoji način da se referencira (pamti se referenca na nešto) neki objekt, no to nećemo obrađivati na ovom predmetu

⇒ NAPOMENA:

- print @polje ⇒ ispisuje sve elemente polja slijepljene, jedan za drugim
- print "@polje" ⇒ ispisuje elemente polja razdvojene podrazumijevanim separatorom

⇒ <u>podrazumijevani separator</u> je ugrađena varijabla: $"

⇒ **foreach**: iterator za petlju foreach je privatan za doseg petlje te će prilikom izlaska vrijediti stara vrijednost varijable koja ima isto ime kao iterator (ako takva varijabla postoji)

⇒ ukoliko nije naveden iterator, koristi se podrazumijevana varijabla $_

↳ ovo ćemo često koristiti!

⇒ <u>operatori za rad s listama:</u>

- **reverse** - vraća novu listu, obrnutu
- **sort** - sortira po ASCII redoslijedu po defaultu
  - ako želimo drugačije, trebamo definirat potprogram za usporedbu
  - vraća novu listu, sortiranu

```
sort { $a <=> $b } @polje
```

sortiranje po brojčanoj vrijednosti elemenata polja

⇒ POTPROGRAMI:

↳ odvojen prostor imena (predznak &)

↳ definicija započinje ključnom riječi „sub"

↳ povratna vrijednost jest rezultat zadnjeg izraza u potprogramu (može biti problem ako želimo raditi još nešta nakon zadnjeg izraza)

```
sub suma {
    $a + $b;
}
```

↳ zbog mogućih problema oko povratne vrijednosti, uvedena je ključna riječ „return"

↳ ARGUMENTI POTPROGRAMA:

- ne navode se kod definicije
- stoga, svaki potprogram može imati argumente
- lista argumenata se pohranjuje u posebnu varijablu (polje):

@_

- pristup argumentima: $_[0], $_[1], ...
- postoj i mehanizam imenovanja argumenata - kreiranje lokalnih varijabli:

```
sub max {
    my ($m, $n);
    ($m, $n) = @_;
    ...
}
```

⇒ „diamond" operatori:

↳ operatori <>

↳ sa standardnog ulaza možemo čitati na sljedeći način:

- <STDIN>
- <>

↳ općenito, unutar diamond operatora se navod ime datoteke iz koje se čita (ukoliko datoteka nije navedena („<>"), čita se sa standardnog ulaza - ideja je imitirati ponašanje UNIX alata)

⇒ argument naredbenog retka:

↳ predaju se preko posebnog polja @ARGV

↳ ime programa je pohranjeno u poseban varijabl $0

⇒ DATOTEKE:

↳ za pristup datoteci koristi se njen identifikator

↳ 6 posebnih identifikatora:

STDIN, STDOUT, STDERR, DATA, ARGV, ARGVOUT

↳ operator „open" s tri argumenta je dosta sigurniji od onog s dva (jer ovog s dva ima mogućnost da netko napravi napad sličan SQL injection napadu)

```
open PROBA, "<", "$ime" or die
```

↳ „die" je funkcija koja ispisuje poruku na stderr i prekida izvođenje

↳ posebna Perl varijabla: $!

- sadrži poruku o pogreški operacijskog sustava (npr. „File not found!")

↳ konvencija je da se identifikatori datoteka pišu velikim slovima

↳ „print" i „printf" imaju podrazumijevani identifikator datoteke STDOUT

- taj podrazumijevani identifikator datoteke mijenjamo naredbom **select**

⇒ ASOCIJATIVNA POLJA:

↳ mape, rječnici (u drugim jezicima)

↳ elementi se pohranjuju tehnikom raspršenog adresiranja

↳ ključ ⇒ bilo koj skalar koj se pretvara u string

```
$hash{$some_key} = $some_value
```

↳ element asocijativnog polja može bit samo skalar (il referenca)

↳ čitanje s ključa koji se dosada nije javio vraća vrijednost **undef**

↳ **asocijativno polje kao cjelina** označuje se:

```
%hash
```

↳ asocijativno polje se može pretvoriti u listu i obrnuto:

- lista mora biti u obliku parova ključ - vrijednost
- prilikom takve pretvorbe redoslijed nije očuvan

↳ asocijativno polje može se kopirati:

```
%new_hash = %old_hash
```

↑ pretvara se u listu, pa se %new_hash inicijalizira tom listom

→ NAPOMENA: u Perlu postoji sinonim za zarez: operator =>

↳ ovaj operator možemo koristiti svugdje umjesto zareza, ali ima najviše smisla za asocijativna polja:

```
%last_name = (
        "fred" => "flintstone",
        "dino" => undef,
        "barney" => "rubble",
        "betty" => "rubble"
);
```

↳ funkcija „each" svakim pozivom vraća sljedeći element asocijativnog polja, a kada dođe do kraja vraća se prazna lista

- ovo je korisno za iteriranje po cijelom asocijativnom polju

↳ funkcija „delete" briše zapis iz asocijativnog polja

⇒ REGULARNI IZRAZI U JEZIKU PERL:

↳ vrijede jako slična pravila kao i kod regularnih izraza ljuske

↳ Perl kvantifikatori su pohlepni

• to znači da žele podudariti najduljj mogući niz

• ako želimo nepohlepni kvantifikator stavimo „?" nakon kvantifikatora

↳ npr. /.+?:/

nepohlepni kvantifikator „+"

(cijeli izraz daje niz znakova do prve dvotočke)

↳ operacije s regularnim izrazima:

a) PODUDARANJE (pretraživanje)

b) ZAMJENA (substitucija)

↳ sintaksa je vrlo slična sintaksi naredbe „sed" iz Unix alata

↳ KORISNO: ako želimo podudariti . sa znakom "\n" uključimo opciju /s („single-line")

↳ još jedna slična operacija koju obrađujemo s regularnim izrazima iako nije regularan izraz je TRANSLITERACIJA, a obavlja zamjenu znak po znak

↳ ako želimo raditi regularni izraz nad nekom drugom varijablom osim $_ koristimo BINDING OPERATOR:

$varijabla =~ /...../

⇒ napomena:

↳ pogledajmo primjer sa globalnim podudaranjem:

```
$_ = "ovo je jedna proba";
@rez = /(\w+)/g;
print "@rez", '\n';
print $1
```

↳ ispisat će se:

↳

proba

↳ razlog tome je što imamo samo jednu zagradu, pa se samo u jednu varijablu podudaranja spremaju pronađeni nizovi (ostaje samo radnji)

PROGRAMSKI JEZIK PYTHON

⇒ postoje dvije verzije međusobno nekompatibilne
- Python 2
- Python 3

↳ verzija 3 postoji od 2009. godine i nije kompatibilna sa verzijom 2 jer se je odlučila da se „ne želi biti opterećen lošim stvarima iz prošlosti"

↳ mi ćemo raditi sa Pythonom 3 !

⇒ IZVOĐENJE PROGRAMA:
a) interaktivno
b) kao modul (pohranjen u tekstualne datoteke)
c) iz ljuske (kao skripta)

⇒ konvencija je stavljati nastavak ".py" na programe pisane u Pythonu

↳ ovo je potrebno ako želimo taj program koristiti negdje drugdje naredbom „import"

↳ to se naziva MODUL

↳ prilikom učitavanja modula naredbom „import" pokreće se izvođenje tog modula (obavlja se samo prilikom prvog učitavanja !)

⇒ modul nam služi za organizaciju biblioteka i alata

↳ modul općenito predstavlja prostor imena („namespace")

```
>>> import myfile
>>> print(myfile.title)
```

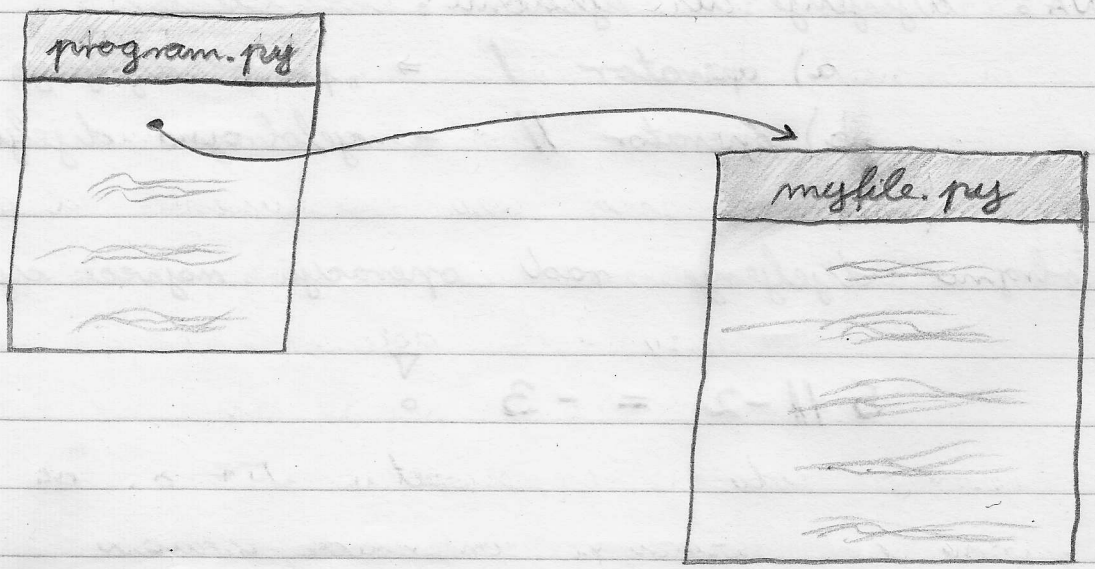
} „myfile" je ime poznato našem programu

```
>>> from myfile import title
>>> print(title)
```

} sada „myfile" nije poznat našem programu

⇒ znači, u prilikom importa, stvara se referenca u našem programu na objekt koji želimo uključiti:

```
>>> import myfile
```



⇒ UGRAĐENI TIPOVI PODATAKA:

↳ valja ih koristiti jer su brze zbog toga što je interpreter implementiran u C-u

↳ tipovi: broj, znakovni niz, lista, rječnik, n-torka, datoteka

⇒ NAPOMENA: u Pythonu nema overflowa cijelih brojeva! ▽

(omogućeno jer se svaka znamenka pamti za sebe)

⇒ IMENA („varijable"):

- ime nema veze sa tipom, tj. ono je **referenca na neki objekt**
- ime se stvara u trenutku prvog dodjeljivanja vrijednosti
- imena se ne deklariraju unaprijed - uvijek ih prilikom stvaranja moramo vezati uz neki objekt

⇒ NAPOMENA: dijeljenje u Pythonu:

a) operator / ⇒ „pravo" dijeljenje

b) operator // ⇒ cjelobrojno dijeljenje

PR: Cjelobrojno dijeljenje radi operaciju najveće cijelo: ▽

$$5 // -2 = -3$$

⇒ heksadekadska, oktalna i binarna notacija:

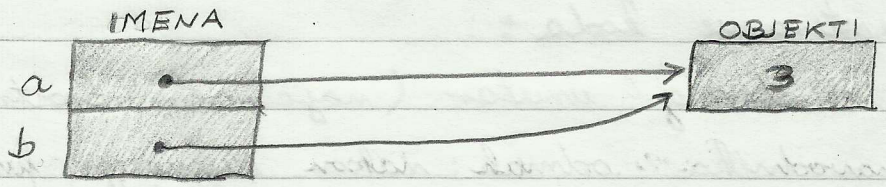
0xA1 0o62 0b1101

⇒ NAPOMENA: imena su reference!

BITNO!

↳ ako imamo sljedeći kod:

```
a = 3
b = a
```



↳ tip je svojstvo objekta, a ne imena

⇒ NEPROMJENJIVOST (engl. „immutability"):

↳ brojevi i znakovni nizovi su primjeri nepromjenjivih objekata

↳ liste su primjer promjenjivih objekata

⇒ korisno za kasnije:

- element liste je ime / referenca koj se ista ponaša kao i „stand-alone" ime

⇒ GARBAGE COLLECTION:

↳ ponaša se isto kao u drugim jezicima

↳ radi se brojanjem referenci (kad je brojač nula ⇒ RECIKLIRAJ!)

⇒ ZNAKOVNI NIZOVI:

↳ uređena, neizmjenjiva kolekcija znakova

↳ ugrađena je podrška za obradu znakovnih nizova

↳ svrstavaju se u SEKVENCE (nepromjenjive)

• to znači da se može pristupiti nekom znaku niza na isti način kao što pristupamo elementu polja

↳ dokumentiranje koda:

• string unutar (najčešće) trostrukih navodnika odmah nakon definicije funkcija

```
def mojaFunkcija:
    """ Ovo je moja funkcija.
        Radi ovo i ono!
        """
    ...
```

} zapis stringa kroz više redaka

↳ stringovi mogu biti uokvireni jednostrukim i dvostrukim znakovima (ekvivalentno!)

↳ interno, string nije niz byte-ova jer (od Pythona 3) je podržana internacionalizacija

↳ nizovi nisu zaključeni nul-terminatorom već se interno vodi informacija o duljini

↳ duljina se dobiva funkcijom len(s)

⇒ kodiranje izvornog koda je po default-u UTF-8, pa je i za izvorni kod podržana internacionalizacija

⇒ nizovi bajtova:

↳ pisemo binarne znakove u nizovima

```
>>> bajtovi = b'abc\x00\x41\x42\x43'
```

ASCII vrijednosti

brojčana ASCII vrijednost (heksadekadska)

↳ moramo paziti s kojom kodnom stranicom odlučimo kodirati znakovni niz jer isti niz nema istu bajtnu reprezentaciju u različitim kodnim stranicama (npr. UTF-8, windows-1250)

⇒ LISTE:

↳ sadržavaju reference na objekte

↳ podržavaju umetanje, gniježćenje, ponavljanje, ...

↳ može se proširit bilo kojem ITERABILNIM (POBROJIVIM) tipom

↳ napomena: matrica najčešće prikazujemo ugniježćenim listama

```
matrica = [[2, 9, 3],
          [8, 4, 7],
          [2, 5, 1]]
```

↳ liste spadaju u izmjenjive objekte, pa treba biti oprezan s pristupom nakon neke izmjene preko svih referenci

⇒ RJEČNICI:

↳ neuređena kolekcija objekata

↳ elementi se dohvaćaju i pohranjuju pomoću ključa elementa

↳ efikasno ostvaren

↳ promjenjivi objekt

↳ analogni mapama u Javi

$$D1 = \{'spam' : 2 , 'eggs' : 3\}$$

↳ kroz rječnik se može iterirati, al se iterira po ključevima po defaultu

↳ dva rječnika „spajamo" metodom **update**:

d2.update(d3)

⇓

d3 se dodaje u d2 i

pregazuje vrijednosti s istim ključem

↳ ključ rječnika može biti bilo koj nepromjenjivi tip koj ima metodu **__hash__**

↳ korištenje rječnika kao liste je „rogobno" jer možemo pohraniti rijetke indekse bez zauzimanja puno memorije

$$A[33] = 12$$

lista A mora imati barem 100 elemenata

rječnik A može imati samo jedan element

⇒ N-TORKE :

↳ „tuple" je čest naziv

↳ uređena kolekcija proizvoljnih objekata

↳ n-torke nisu izmjenjive !

↳ korisno jer daje integritet

↳ navodi se u dobrim zagradama :

$$t2 = (0, \text{'Ni'}, 1.2, 3)$$

↳ NAPOMENA : nepromjenjivost vrijedi samo na prvu razinu n-torke što znači da ako imamo npr. listu u n-torki mi imamo referencu na tu listu preko n-torke i tu listu možemo mijenjat

BITNO !

⇒ SKUPOVI:

↳ neuređena kolekcija međusobno različitih objekata koji su **jedinstveni**

↳ elementi su uređeno i ključevi

$$skup = \{'kruška', 'šljiva', 2\}$$

↳ neke operacije:

- dodavanje (add)
- ažuriranje (update)
- uspoređivanje - podskup: "<="
- brisanje (remove)
- ispitivanje pripadnosti (in)
- operacije algebre skupova:

unija, presjek, razlika

⇒ DATOTEKE:

↳ otvaramo streame pomoću funkcije:

```
dat = open('fileName', 'mode')
```

↳ funkciji open možemo dati i kodnu stranicu po kojoj da dekodira stranice:

- podrazumijevanu kodnu stranicu možemo vidjeti iz otvorenog toka:

```
dat.encoding
```

↳ ako ne navedemo 'mode', datoteka se otvara samo za čitanje

↳ pomicanje pokazivača datoteke:

```
dat.seek(0)
```

↑ pozicioniranje na početak

↳ **tok je iterabilan!**

↳ iteriramo kroz datoteku redak po redak

↳ automatsko zatvaranje i oslobađanje resursa radimo ključnom riječi WITH:

```
with open('file.txt') as dat:
    for line in dat:
        print(line)
```

↳ binarne datoteke:

- ne radaju se kodna stranica
- predviđena za ne-tekstualne datoteke
- 'mode' mora imati 'b' na kraju
- pazi!

↳ pri čitanju i pisanju se ne obavljaju automatske prilagodbe kraja retka

- datoteka otvorena kao binarna prilikom čitanja vraća niz bajtova, a ne string.

↳ kako pohraniti složenije objekte u datoteku?

a) u obliku znakovnih nizova:

- prilikom učitavanja, nizove znakova pretvaramo nazad u objekte

b) serijalizacija:

- koristimo modul **pickle**
- datoteka mora biti otvorena kao binarna
- metode:

pickle.dump(...)
pickle.load(...)

→ generalne napomene u vezi jezika Python:

1) USPOREDBE:

L1 == L2 → ispitivanje jednakosti objekata (po sadržaju)

L1 is L2 → ispitivanje jednakosti referenci na objekte

↳ svi objekti koji su neprazni su True

↳ svi brojevi različiti od nule su True

2) PISANJE NAREDBI:

↳ pazi na indentaciju!

↳ dvotočka u zaglavlju složene naredbe je obavezna

↳ ulančavanje jednostavnih naredbi u istom retku:

```
a = 2 ; b = 3 ; print(a+b)
```

↳ složene naredbe s kratkim blokom možemo pisati u jednom retku:

```
if a == b : break
```

↳ u Pythonu ne postoj switch-case

- radimo pomoću **dict** il pomoću rječnika s funkcijama

↳ „while" i „for" imaju blok **else** koj se izvršava samo ako je petlja završila do kraja!

3) FUNKCIJA range:

↳ generira pobrojiv objekt s cjelim brojevima (u Pythonu 2 to je bila lista ☹)

range (begin , end , step)

↑ INCLUSIVE ↑ EXCLUSIVE

↳ broj se generira tek kad se zatraži pozivom funkcije .next() nad range objektom

↳ to rad for petlja ☺

↳ samo „end" je obvezan

- default begin: 0
- default step: 1

↳ korisna funkcija je zip koja lijepi dvije kolekcije (VIDI SLAJDOVE ☺)

4) FUNKCIJE:

↳ funkcije su objekt (čak su i definicije razreda objekti)

↳ korisno jer npr. funkcije možemo dinamički stvarat ili dodati u rječnik, pa ovisno o nekom uvjetu pozvati funkciju koju u tom trenutku trebamo

↳ argumenti funkcije postaju imena referenci na objekte koji su poslani u funkciju

⇒ RAZRJEŠAVANJE IMENA:

↳ imena koja nastaju unutar funkcije su lokalna - vidljiva samo unutar funkcije

↳ globalni doseg je doseg modula

↳ redoslijed razrješavanja:

1. lokalni doseg
2. obuhvaćajući doseg
3. globalni doseg
4. ugrađeni doseg

↳ varijable iz obuhvaćajućeg dosega i globalnog dosega se ne mogu mijenjati osim ako ta imena nisu deklarirana kao globalna

↳ u Pythonu, argumenti se razmjenjuju pridruživanjem (CALL BY VALUE) referenci

↳ dvije ključne riječi za mijenjanje dosega:

a) global
b) nonlocal

⇒ ipak, prenošenje argumenata u Pythonu nije strikno BY VALUE ili BY REFERENCE:

- izmjenjivi tipovi su proslijeđeni BY REFERENCE
- neizmjenjivi tipovi su proslijeđeni BY VALUE

↳ sve se zapravo prosljeđuje preko referenci, ali je korisno tako razmišljati jer se objekti tako ponašaju u Pythonu

PR: Izbjegavanje izmjene argumenta!

↳ kad bi htjeli da nam funkcija ne izmijeni listu koju joj šaljemo, možemo joj:

a) poslati kopiju liste:

```
lista = [1, 2]
changer(broj, lista[:])
```

b) u funkciji raditi kopiju:

```
def changer(num, list):
    list = list[:]
    :
```

⇒ Python nema „call by reference", pa ga oponašamo složenim povratnim tipovima

- npr. n-torka - izmijenjenih vrijednosti argumenata funkcije

⇒ proizvoljna lista argumenata funkcije:

```
def f(*args):
    :
```

} argumenti se pakiraju u n-torku

* ⇒ operator raspakiravanja u drugom kontekstu

⇒ drugi oblik proizvoljnog broja argumenata je imenovanje argumenata prilikom poziva te se oni pakiraju u rječnik:

{'name1' : 1 , 'name2' : 2 , ...}

↳ definicija takve funkcije:

```
def f(**args):
    :
```

⇒ LAMBDA IZRAZI:

↳ može se pisati svugdje gdje sintaksa dozvoljava izraz

↳ koristi se najčešće za beznamene funkcije

lambda arg1, arg2, ..., argN : izraz

⇒ DODATNO O MODULIMA:

↳ „import" cijel modul pridružuje jednom imenu

↳ „from" uvozi (kopira) jedno il više imena (ta imena su reference na dodijeljene objekte)

```
# modul small.py
X = 1
Y = [2, 3]
```

→ sada u nekom drugom modulu pišemo:

stvaramo lokalno ime →

```
from small import X, Y
X = 42
```

mijenjamo izvorni objekt →

```
Y[0] = 13
```

↳ NAPOMENA: vidi slajdove za pojašnjenje funkcije reload()

↳ NAPOMENA: vidi slajdove za odnos i doseg pogleda modula i prostora imena

⇒ skrivanje podataka u modulu:

- imena koja započinju sa '_' se ne kopiraju prilikom učitavanja sa „from *"
- standardno učitavanje sa „import" će dat pristup tim imenima preko imena modula

⇒ OBJEKTNO-ORIJENTIRANO PROGRAMIRANJE U PYTHONU:

↳ Python je potpuno objektno-orijentiran jezik kao i Java - sve su objekti!

↳ stvaranje razreda radi se ključnom riječi **class**

↳ razred je (kao i funkcija) objekt prvog reda

↳ stvaranje primjerka:

```
primjerak = Razred()
```

↳ primjerku možemo dati specifične atribute i to samo za taj primjerak nakon što je on već stvoren:

```
primjerak.X = 2
```

↳ metode za objekte moraju biti definirane unutar razreda sa riječi **self**

↳ „self" je analogan „this" u Javi

↳ pri inicijalizaciji novog objekta poziva se metoda **__init__()**

↳ nasljeđivanje je jednostavno:

```
class NoviRazred(Mama, Tata):
    :
```

↓

podržano je višestruko nasljeđivanje, ali je redoslijed navođenja razreda prilikom nasljeđivanja bitan!

VIDI SLAJDOVE!