

# Programska potpora komunikacijskim sustavima

• Izv. prof. dr. sc. Goran Delač

## Uvod u programski jezik Python



# Sadržaj predavanja

- Uvod
- Osnove jezika
- Tipovi podataka

# Uvod



# O programskom jeziku Python

- Programski jezik opće namjene
- Brz i jednostavan razvoj primjenskih programa
  - **Skriptni** jezik
  - **Interpretirani** jezik
  - Visoka razina apstrakcije
  - Automatsko upravljanje memorijom
  - **Dinamički tipovi** podataka

# O programskom jeziku Python

- Podržano više programskih paradigmi
  - Objektno orijentirano programiranje
  - Imperativno programiranje
  - Funkcijsko programiranje
- Bogata programska knjižnica
  - **Standardna** knjižnica
  - Strojno učenje, obrada slika, ugrađeni sustavi, web aplikacije...

# Popularnost Pythona (2022.)

## ■ GitHub

- 1. Javascript, 2. **Python**, 3. Java
- <https://octoverse.github.com/#top-languages-over-the-years>

## ■ Tiobe

- 1. **Python**, 2. C, 3. Java, 4. C++
- <https://www.tiobe.com/tiobe-index/>

## ■ PYPL

- 1. **Python** (29% udjela), 2. Java (18% udjela), 3. JavaScript
- <https://pypl.github.io/PYPL.html>

# Povijest

- Začetak u Nizozemskoj, kasne 1980te
  - Guido van Rossum
  - Izvor imena je serija „Monty Python's Flying Circus”
- Prva verzija – veljača 1991
- Python 2.0 – 2000. godine
  - Python Software Foundation (2001. godine)
- Python 3.0 – 2008. godine
  - Veći broj poboljšanja
  - Nije kompatibilan s verzijama 2.x

# Povijest

“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered.”

- Guido van Rossum





# Prednosti i nedostaci

## ■ Prednosti:

- Jednostavnost korištenja – potrebno je malo prethodnog znanja
  - Dinamički tipovi podataka, interaktivno izvođenje ...
- Bogata programska knjižnica
- Čitljivost programskog koda – sintaksa jezika podređena čitljivosti, de-facto standardi

## ■ Nedostaci:

- Python je spor! (izvođenje ciljnog programskog koda)
- Teže održavanje velikih projekata

Jednostavnost  
korištenja



Brzina

# Dinamički tipovi podataka

- Nije potrebno zadati tip podatka

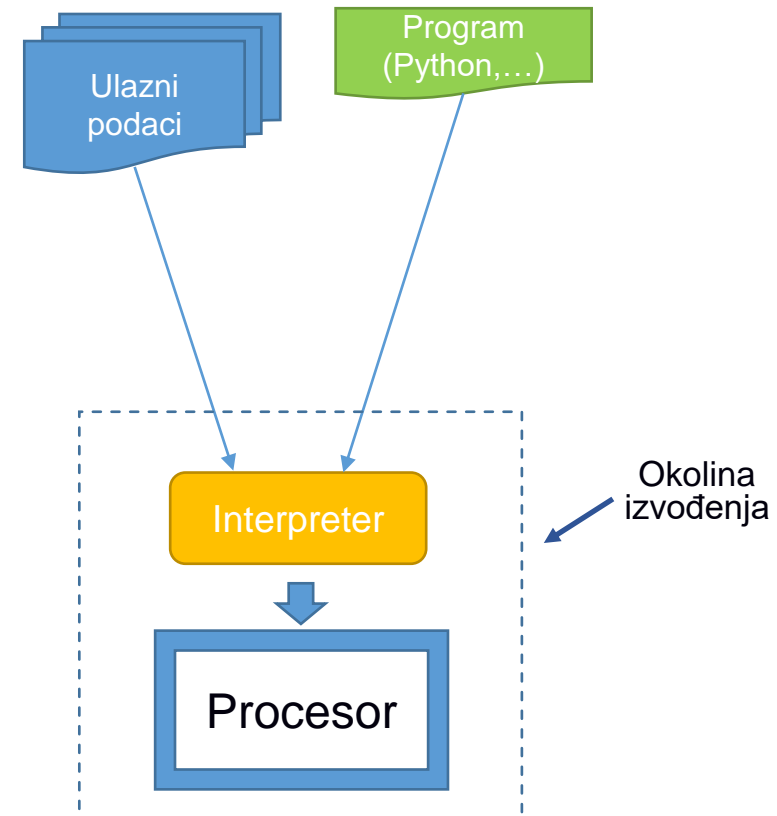
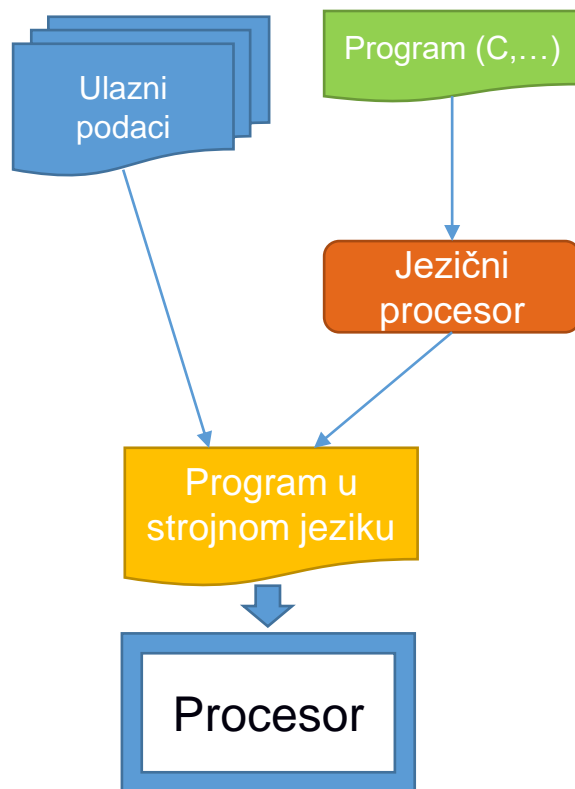
```
a = 3
b = [1, 2, 3, 5]
c = 'Niz znakova'
a = c
```

- Nedostatak:
  - Kod programskih jezika sa statički zadanim tipovima podataka jezični procesor (kompajler) odredi tipove prilikom prevođenja – nije ih potrebno naknadno provjeravati.
  - Programski jezici sa dinamički zadanim tipovima podataka provjere moraju obaviti za vrijeme izvođenja – obično su sporiji.

# Interpreteri

- Poseban tip jezičnog procesora
  - Prevode naredbu po naredbu
    - Jezični procesori opće namjene prevode cijeli izvorni kod u strojni kod (potrebno je na ulazu postaviti kompletan kod koji sačinjava programsko ostvarenje)
  - Python je interpretirani jezik
  - Mogućnost interaktivnog izvođenja programa
    - Unos programa naredbu po naredbu
    - Praćenje izlaza nakon svake naredbe

# Interpreteri



# Interpreteri

- Nedostatak:

- Rezultantni strojni kod je sporiji jer prilikom prevođenja nije moguće provesti neke od postupaka optimizacije
  - Manji broj strojnih naredbi
  - Učinkovitije korištenje naredbenog cjevovoda
  - Učinkovitije korištenje računalne arhitekture (postavljanje podataka u registre)

a = 3

~~a = 3~~

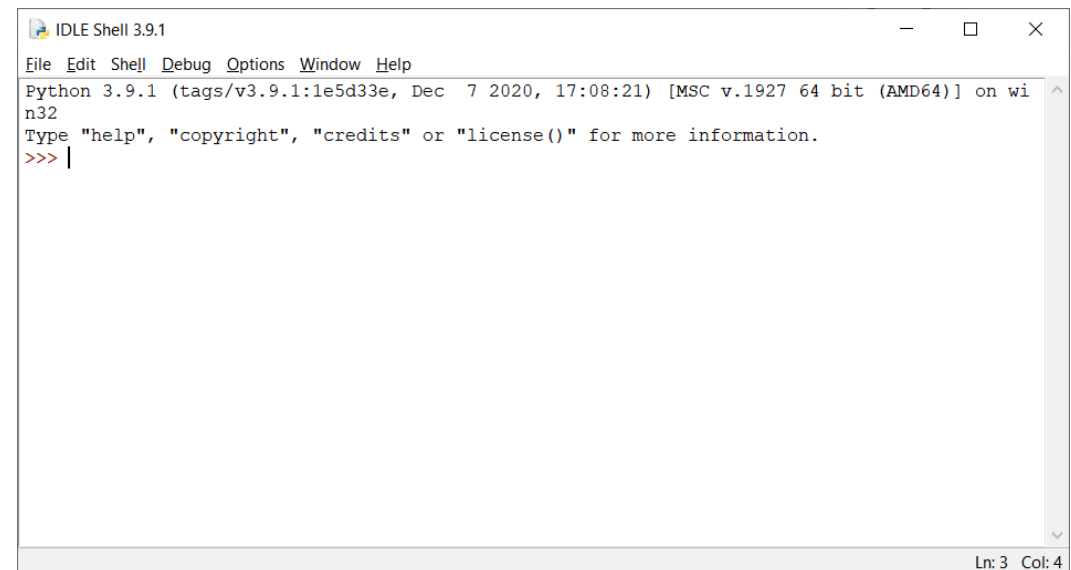
c = 3 + 1 → c = 4

# Skriptni jezik

- Viša razina od uobičajenih programskih jezika
- Komponiranje programa od komponenti
  - Programske knjižnice
  - Primjenski programi
  - Mogu biti napisani u drugom programskom jeziku!
- Primjer: NumPy
  - Rad s višedimenzionalnim poljima
  - Implementacija: C, Python

# Radno okruženje

- Postavljanje Python interpretera
  - <https://www.python.org/downloads/>
  - Linux (Ubuntu): `sudo apt-get install python`
- Pokretanje interpretera
  - `python (.exe)`
  - Zatvaranje interpretera: `quit()`
- IDLE
  - Integrirano razvojno okruženje



```
IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

# Radno okruženje

- Python programi (skripte)
  - Datoteke s ekstenzijom `.py`
- Pokretanje programa
  - `python imePrograma.py [arg1] [arg2] ...`
- Korišćenje prikladnog programa za uređivanje teksta
  - Sublime Text, Notepad++, ...
- Radna okruženja
  - PyCharm, Eclipse, ...

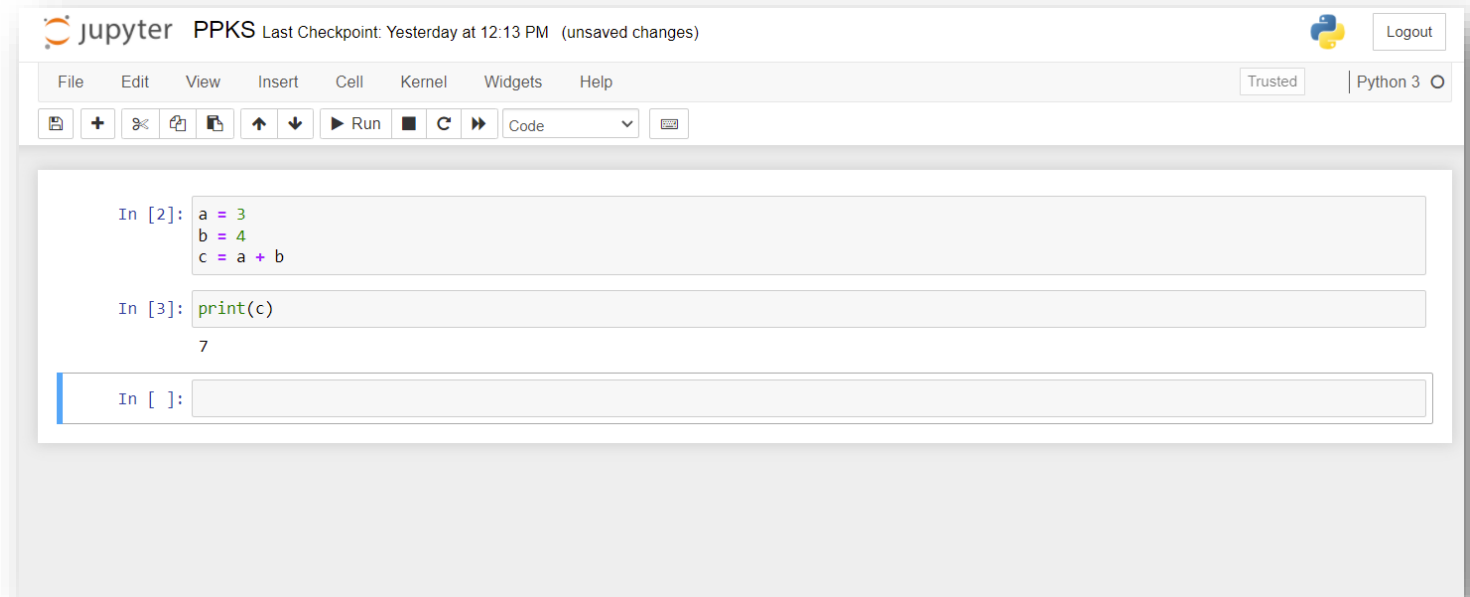


# Radno okruženje

- Python programi (skripte)
  - Datoteke s ekstenzijom `.py`
- Pokretanje programa
  - `python imePrograma.py [arg1] [arg2] ...`
- Korišćenje prikladnog programa za uređivanje teksta
  - Sublime Text, Notepad++, ...
- Radna okruženja
  - PyCharm, Eclipse, ...

# Radno okruženje

- Jupyter Notebook
  - Interaktivna bilježnica za pokretanje isječaka programskog koda
  - Integracija dokumenta i programskog koda – dobar alat za prikazivanje i dijeljenje rezultata
  - Web aplikacija
- Postavljanje
  - `pip install notebook`
- Pokretanje
  - `jupyter notebook`
- JupyterLab (alternativno)
  - `pip install jupyterlab`



# Osnove jezika



# Primjer programa

```
SUFFIXES = { 1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, kb_is_1024_bytes = True):
    '''Convert a file size to human-readable form.
    Description...
    '''

    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if kb_is_1024_bytes else 1000

    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)


    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

# Sintaksa (1)

- Naglasak na čitljivosti koda, jednostavnosti pisanja programa
- Nije nužno koristiti oznaku kraja naredbe ";"
  - Naredba se proteže do kraja retka
  - Prelamanje naredbe kroz više redova se postiže ključne riječi "\"
- Ne ključne riječi za označavanje početka i kraja bloka naredbi (npr. "{" i "}")
  - Blokove određuje broj praznih znakova na početku naredbe
  - Naredba koja započinje blok završava znakom ":"
  - Tabulator (Tab) ili prazan znak
  - 4 prazna znaka (de facto standard)

# Sintaksa (2)

Uvjet:   
\_\_\_\_ **Naredba1**  
\_\_\_\_ **Naredba2**  
\_\_\_\_ Uvjet:  
\_\_\_\_ **Naredba3**  
\_\_\_\_ Naredba4



Ispravno

Uvjet:  
\_\_\_\_ **Naredba1**  
\_\_\_\_ **Naredba2**  
\_\_\_\_ Uvjet:  
\_\_\_\_ **Naredba3**  
\_\_\_\_ Naredba4



Neispravno

# Sintaksa (3)

- Prazan blok naredbi
  - Ključna riječ **pass**

```
Uvjet:  
____Naredba1  
____Naredba2  
____Uvjet:  
____pass  
____Naredba4
```

# Sintaksa (4)

- Deklaracija varijabli:

```
a = 1000  
b = 3 + 1  
c = 'Niz znakova'
```

- Programski tipovi se **ne** deklariraju
  - Varijabloma se pridružuje vrijednost iz koje se određuje tip podatka
  - Dohvaćanje prethodno nedefinirane varijable izaziva grešku



# Sintaksa (5)

- Deklaracija varijabli (primjer):

```
SUFFIXES = { 1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],  
             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

- Osnovne ugrađene strukture podataka

- Riječnik, mapa, asocijativno polje

```
ime_mape = { ključ1 : vrijednost1, ključ2 : vrijednost2 }  
vrijednost = ime_mape[ključ]
```

- Liste

```
ime_liste = [vrijednost1, vrijednost2]  
vrijednost = ime_liste[cijeli_broj]
```

# Sintaksa (6)

- Definiranje funkcije
  - Programski isječci koji o odrađuju određeni zadatak
  - Ulazni podaci (parametri), rezultati

```
def approximate_size(size, kb_is_1024_bytes = True):
```

- Ključna riječ **def**
- Naziv funkcije `approximate_size`
- Argumenti `size`, `kb_is_1024_bytes`
- Dinamički tipovi podataka
  - Nije zadan tip argumenata
  - Nije zadan tip povratne vrijednosti i vraća li funkcija povratnu vrijednost

# Sintaksa (7)

- Vraćanje vrijednosti iz funkcije

```
return neka_vrijednost  
return '{0:.1f} {1}'.format(size, suffix)
```

- Ako funkcija ne vraća povratnu vrijednost rezultat je tipa `NoneType` (**None**)
  - Slično kao `Null` u drugim programskim jezicima
  - Tip podataka koji označava praznu vrijednost

- Pozivanje funkcije

```
povratna_vrijednost = ime_funkcije(argumenti)  
approximate_size(1000000000000)
```

# Sintaksa (8)

- Opcionalni argumenti

pretpostavljena vrijednost

```
def approximate_size(size, kb_is_1024_bytes = True):
```

- Opcionalni argument: kb\_is\_1024\_bytes

- Nije ga potrebno zadati u pozivu funkcije – poprima pretpostavljenu vrijednost
- Nije zadan tip povratne vrijednosti i vraća li funkcija povratnu vrijednost
- Navode se uvijek na kraju liste argumenata

```
print(approximate_size(10000000000000, False))  
print(approximate_size(10000000000000))
```

# Sintaksa (9)

## ■ Komentari

- Unutar trostrukih navodnika `'''`
  - protežu se kroz više linija
- Nakon znaka
  - Protežu se do kraja linije `#`

```
'''Convert a file size to human-readable form.  
    Description...  
'''
```

```
# komentar programskog bloka
```

# Sintaksa (10)

- Uvjetno ganjanje

```
if size < 0:  
    raise ValueError('number must be non-negative')
```

- Uvjetni operator

```
multiple = 1024 if kb_is_1024_bytes else 1000
```

# Sintaksa (11)

- Uvjetno ganjanje - općenito

```
if uvjet1:  
    Naredba1  
    Naredba2  
    ...  
elif uvjet2:  
    Naredba  
    ...  
else:  
    Naredba  
    ...
```

```
rezultat = vrati_ako_tocno if uvjet else vrati_ako_netocno
```

# Sintaksa (12)

## ▪ Iznimke

- Programski konstrukti u višim programskim jezicima koji se koriste za dojavljivanje greške
- Iznimku je moguće izazvati (stvoriti)
- Iznimku je moguće obraditi (`try-except-finally` skup naredbi)

```
raise Objekt_Tipa_Exception
```

```
raise ValueError('number must be non-negative')
```



# Sintaksa (13)

- Programske petlje

```
for x in iteratorski_objekt:  
    # x je i-ti element u iteratorskom objektu - lista, mapa...
```

**Naredba**

**Naredba**

```
for suffix in SUFFIXES[multiple]:
```

```
SUFFIXES = { 1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],  
             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

# Sintaksa (14)

- Programske petlje

```
while uvjet:
```

```
    # iteracija regulirana uvjetom
```

```
    Naredba
```

```
    Naredba
```

# Sintaksa (15)

## ■ Aritmetičke i logičke operacije

- Zbrajanje +, oduzimanje -, množenje \*, dijeljenje /, potenciranje \*\*
- Operacije nad bitovima : &, |, ^
- Logičke operacije: and, or, not
- Operacije usporedbe: >, >=, <, <=, ==, !=, <>

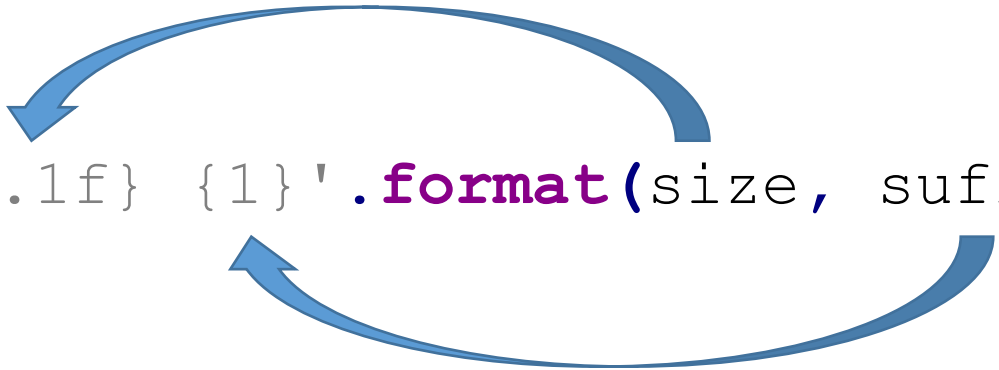
## ■ Konstante

- Znakovni niz: 'Primjer znakovnog niza'
- Cijeli broj: 10000000
- Brojevi s pomičnim zarezom: 425.46

size /= multiple ↔ size = size / multiple

# Sintaksa (16)

- Formatiranje znakovnog niza



The diagram illustrates the mapping between format specifiers in a string and the arguments provided to the `.format()` method. The string `'{0:.1f} {1}'` is shown in a monospaced font. The `.format()` method is highlighted in purple, followed by its arguments `(size, suffix)` in a blue monospaced font. A blue curved arrow originates from the `{0:.1f}` specifier and points to the `size` argument. Another blue curved arrow originates from the `{1}` specifier and points to the `suffix` argument.

```
'{0:.1f} {1}'.format(size, suffix)
```

# Sintaksa (17)

## ■ Glavni program

- Svodi se na ispitivanje trenutnog programskog modula
- Ime programskog modula (knjižnice) koji se izvodi pohranjeno je u varijabli `__name__`
- Modul koji je interpreter pokrenuo izravno naziva se: `'__main__'`

```
if __name__ == '__main__':  
    print(approximate_size(10000000000000, False))  
    print(approximate_size(10000000000000))
```

# Tipovi podataka



# Tipovi podataka - Objekti

- Svaka vrijednost ima tip podatka
  - Varijabloma se određuje tip na temelju pridružene vrijednosti
- Svi tipovi podataka u Pythonu su objekti
  - Objektno orijentirani programski jezik
- Objekti
  - Programski konstrukti koji enkapsuliraju određenu funkcionalnost
  - Svaki objekt je građen od članskih atributa i funkcija (metoda)
  - Korištenje slično struct u programskom jeziku C
  - Operator .
    - `ime_objekta.clanska_var`
    - `ime_objekta.metoda`

```
'{0:.1f} {1}{}'.format(size, suffix)
```

# Tipovi podataka - Objekti

- Razredi
  - Nacrti na temelju kojih se grade objekti
    - Pomoću razreda se programski definira nacrt objekta
    - Moguće je u memoriji stvoriti više objekata istog razreda (više varijabli istog tipa postoji u memoriji)
    - Stvaranje objekta u memoriji na temelju razreda naziva se instanciranje objekta
    - Tip podatka = ime razreda
- Razredi za primitivne tipove podatka (brojeve, nizove znakova)
- Razredi za složene tipove podataka (liste, mape...)
- Sve je objekt!
  - Funkcije su objekti, i sami razredi su objekti



# Određivanje tipa podatka

- Tip podatka je ime razreda iz kojeg je objekt instanciran
- Funkcija `type()` vraća tip objekta
- Funkcija `isinstance()` provjerava je li objekt nekog tipa

```
type(123)
```

```
isinstance(123, int)
```

# Ugrađeni tipovi podataka

- Logički (boolean) – `True` ili `False`
- Brojevi
  - Cijeli brojevi: niz znamenaka `1234`
  - Brojevi s pomičnom točkom `1.124`
  - Razlomci
  - Kompleksni brojevi
- Znakovni nizovi
  - Korištenje jednostrukih ili dvostrukih navodnika
  - Svi nizovi u Pythonu3 su u formatu Unicode
- Liste
- N-torke
- Skupovi
- Mape

# Logički tip podataka

- Tip `bool`
- Vrijednosti: `True` ili `False`
- Rezultat su provođenja logičkih operacija

`1 > 3`

`5 == 4`

`3 != 2`

- Automatska pretvorba u boolean
  - U `False` se pretvaraju: `0`, `' '`, `None`, prazni tipovi (`[]`, `{}`, ...)
  - U `True` sve ostalo

`bool(0)`

# Brojevi

- Cijeli brojevi **int**
  - Proizvoljna veličina, ovisi o veličini radne memorije – bignum (Python 3)
  - Znamenke se pohranjuju zasebno
- Brojevi s pomičnom točkom **float**
  - Preciznost 16 znamenki (double)
- Razlika u definiranju konstante
  - Brojevi s pomičnim zarezom imaju decimalnu točku

# Brojevi

- Automatska pretvorba

- Zbrajanje, oduzimanje, množenje, potenciranje argumenata od kojih je jedan **float**, uvijek rezultira tipom **float**.
- Obavlja se pretvorba **int** u **float** i onda se izvršava operacija
- Dijeljenje dva tipa **int** uvijek rezultira s tipom **float** (Python3)
  - Python 2
    - $1 / 3 = 0$
    - $1 / 2. = 0.5$

# Brojevi

- Cjelobrojno dijeljenje (Python3)
  - Operator `//`
  - Zaokružuje na manji broj ( `3 // 2 = 1` )
  - Ako je jedan operand tipa `float`, rezultat će biti tipa `float`, ali svejedno će se odraditi zaokruživanje na manje
  - Operator ostatka cjelobrojnog dijeljenja `%` ( `2 % 3 = 2` )
- Potenciranje
  - Operator `**`
  - `2 ** 12` – 2 na 12. potenciju

# Brojevi

- Razlomci
  - Potrebno koristiti biblioteku **fractions**
  - Moguće je provoditi standardne operacije
  - Automatsko skraćivanje

```
import fractions
a = fractions.Fraction(2, 4)
b = fractions.Fraction(8, 12)
a + b
```

# Brojevi

- Kompleksni brojevi
  - Programska knjižnica standardno uključena
  - Zadaje se realni i imaginarni dio
  - Moguće je provoditi standardne operacije

```
c = complex(1, 3)
d = complex(5, 8)
c * d
```



# Programska potpora komunikacijskim sustavima

• Izv. prof. dr. sc. Marin Šilić

## Programski jezik Python – Tipovi Podataka



# Sadržaj predavanja

- Liste
- N-torke
- Skupovi
- Mape

# Liste



# Liste

- Lista – apstraktni tip podatka koji podrazumijeva uređenu kolekciju podataka
- Razlika između liste i polja
  - Broj elemenata liste je promjenjiv (nema fiksnu duljinu)
  - Dodavanje i brisanje elemenata liste tijekom izvođenja
- Elementi liste mogu biti različitog tipa
- Liste su slične ArrayList razredu u Javi i C#-u

# Liste

- Stvaranje liste

```
lista = [0, 1, 2, 3, 4]  
lista = list()
```

- Dohvaćanje elemenata liste
  - Dohvaćanje jednog elementa

```
lista[indeks]
```

# Liste

- Negativni indeksi

`lista[indeks]`

- Prvi element liste ima indekse
  - `0`
  - `-(duljina)`
- Zadnji element liste ima indekse
  - `-1`
  - `(duljina-1)`

# Liste

- Dohvaćanje elemenata liste

- Dohvaćanje raspona elementa
- Kao rezultat vraća novu listu

```
raspon = lista[indeks_pocetka: indeks_kraja]
```

- Vraća elemente od `indeks_pocetka` (**uključujući**) do `indeks_kraja` (**isključujući**)

- Ako se `indeks_pocetka` izostavi – `indeks_pocetka` je 0
- Ako se `indeks_kraja` izostavi – `indeks_kraja` je *duljina*
- Oba indeksa se mogu izostaviti – vraća kopiju cijelog polja

# Liste

- **Zadatak**
  - Definirati novu listu sa elementima različitog tipa
  - Dohvatiti elemente liste sa pozitivnim i negativnim indeksima
  - Dohvatiti raspon elemenata



# Liste

- Dodavanje elemenata u listu
  - Konkatenacija više lista
  - Stvara novu listu koja sadrži redom elemente više lista  
`nova_lista = lista1 + lista2 + lista3`
- Metoda `append`
  - Dodaje jedan element na kraj liste (ne stvara novu listu)  
`nova_lista.append(100)`
- Metoda `extend`
  - Dodaje elemente liste na kraj druge liste (ne stvara novu listu)  
`nova_lista.extend([200, 300])`
- Metoda `insert`
  - Ubacuje element na određenu poziciju u listi (ne stvara novu listu)  
`nova_lista.insert(1, 543)`

# Liste

## ▪ Zadatak

- Napraviti praznu listu
- Dodati elemente pomoću metoda `append`, `extend` i `insert`
- Konkatenirati listu sa nekom drugom listom
- Sličnosti/razlike `extend` i `+`

```
l1 = [1, 2, 3]
```

```
l2 = [10, 20, 30]
```

```
l1 + l2
```

```
l1.extend(l2)
```

# Liste

- Broj elemenata liste
  - Ugrađena funkcija `len`, nije metoda objekta!
  - `len([1, 3, 5, 7, 7])`
- Broj pojavljivanja elementa u listi
  - Metoda `count`
  - `[1, 3, 5, 7, 7].count(7)`
- Ispitivanje prisutnosti elementa u listi
  - Operator `in`
  - `4 in [1, 3, 5, 7, 7]`
- Indeks prvog pojavljivanja elementa u listi
  - Metoda `index`
  - `[1, 3, 5, 7, 7].index(7)`

# Liste

- Uklanjanje elemenata iz liste
  - Prema indeksu – operator `del` (nije metoda objekta)
    - Uklanja n-ti element liste, ne stvara novu listu  
`del lista[3]`
  - Prema vrijednosti elementa – `remove` metoda
    - Uklanja prvo pojavljivanje elementa u listi, ne stvara novu listu  
`[1, 3, 5, 7, 7].remove(5)`
  - Prema indeksu ili prvi element s kraja liste – metoda `pop`
    - Uklanja zadnji ili n-ti element, ne stvara novu listu  
`[1, 3, 5, 7, 7].pop()`  
`[1, 3, 5, 7, 7].pop(2)`

# Liste

## ▪ Zadatak

- Napisati funkciju koja vraća listu prvih 10 brojeva koristeći petlju while i funkciju `append`

```
i=0
numbers = []
while i<10:
    ...
```

- Napisati funkciju `rem(x, l)` koja briše element s vrijednosti `x` iz liste (koristiti `index + del`)

- `def rem(x, l): ...`
- Funkcija vraća promijenjenu listu
- Provjeriti tipove argumenata (`x → int, l → list`)!

# Liste

- Plitko i duboko kopiranje

- Pridruživanje liste

- ```
l1 = [0, 1, 2]
```

- ```
l2 = l1
```

- Obje varijable pokazuju na istu listu (istu mem. adresu)

- ```
print id(l1), id(l2)
```

- Promjena se vidi u obje varijable

- ```
l1[0] = 10
```

- ```
print l2
```

# Liste

- Plitko i duboko kopiranje

- Kopiranje liste (plitko)

- ```
l1 = [0, 1, 2]
```

- ```
l2 = l1[:]
```

- Sada imena pokazuju na različite liste

- ```
l1[0] = 10
```

- ```
print l1, l2
```

- ```
[10, 1, 2], [0, 1, 2]
```

- Napomena: ako lista sadrži podliste onda je potrebno duboko kopiranje

- ```
import copy
```

- ```
copy.deepcopy(x)
```

# Liste

- Kopiranje na raspon

- Kopiranje na parcijalni raspon

```
l = [0, 1, 2, 3, 4, 5]
```

```
l[1:5] = [10]
```

```
print l
```

```
[0, 10, 5]
```

- Kopiranje na cijeli raspon (mijenjanje in-place)

```
l[:] = [100]
```

```
print l
```

```
[100]
```



# N-torque



# N-torke

- Nepromjenjiva lista
  - Brže izvođenje operacija nego kod listi
  - Nema uklanjanja, dodavanja ili zamjene elemenata
  - Ne podržava metode `append`, `extend`, `insert`, `remove`, `pop` i `del`
- Podržava ostale metode kao i nad listama
  - Jednak način dohvaćanja elemenata
  - Podržava konkatenciju (jer stvara novu n-torku)
  - Podržava `index`, `count`, `len` i `in`

```
ntorka = ('prvi', 'drugi', 3, 4)
```

# N-torke

- Konverzija između n-torke i liste
  - Pomoću razreda `list` i `tuple`
  - Stvaranje novog objekta razreda uz predaju postojeće liste/ntorke

```
ntorka = tuple([1,2,3])
```

```
lista = list(ntorka)
```

# N-torke

- Pridruživanje više vrijednosti pomoću n-torki

$(x, y, z) = (1, 3, 5)$

- Varijable n-torke s lijeve strane pridruživanja poprimaju vrijednosti elemenata n-torke s desne strane
  - Korisno za vraćanje više vrijednosti iz funkcija!

# N-torke

- Pridruživanje više vrijednosti pomoću n-torki

$(x, y, z) = (1, 3, 5)$

- Varijable n-torke s lijeve strane pridruživanja poprimaju vrijednosti elemenata n-torke s desne strane
  - Korisno za vraćanje više vrijednosti iz funkcija!

# N-torke

- Zadatak
- Napisati funkciju `head_tail(l, n)` koja vraća prvih `n` i krajnjih `n` elemenata liste

```
def head_tail(l, n):  
    ...
```

# Skupovi



# Skupovi

- Neuređeni skupovi jedinstvenih elemenata
  - Nema duplikata!
  - Elementi skupa mogu biti različitog tipa
  - Moguće mijenjati sadržaj skupa
  - Elementi skupa mogu biti samo nepromjenjivi objekti
    - Dopušteno: konstante, n-torke, ...
    - Nije dopušteno: liste, ..
  - Redoslijed nije definiran unaprijed

$s = \{1, (2, 3), '4', 5, 5\}$



# Skupovi

- Zadatak
  - Stvoriti nekoliko skupova
  - Pokušati stvoriti skup koji sadrži listu
  - Ispitati tip nekog skupa (funkcija `type`)

# Skupovi

- Stvaranje skupa

- Prazni skup

- ```
skup = set()
```

- Klasično, definiranjem elemenata

- Pretvorbom iz liste ili n-torke pomoću razreda `set`

```
skup = set( (1, (2, 3), '4', 5, 5) )
```

- Veličina skupa – funkcija `len`

# Skupovi

- Dodavanje elemenata u skup
  - Ako element koji se dodaje već postoji – ne dodaje se
  - Metoda `add`
    - Argument je jedan element
  - Metoda `update`
    - Argument je skup, lista ili n-torka elemenata

```
skup = { 1, 2, 3, 4 }  
skup.add( 4 )  
skup.add( 5 )  
skup.update( { 4, 6 } )  
skup.update( [4, 7] )
```

# Skupovi

- Uklanjanje elemenata iz skupa
  - Metoda `discard`
    - Uklanja element iz skupa, ne izaziva iznimku ako element nije u skupu
  - Metoda `remove`
    - Uklanja element iz skupa, izaziva iznimku ako element nije u skupu
  - Metoda `pop`
    - Uklanja neki element iz skupa, izaziva iznimku ako je skup prazan
- `skup = { 1, '2', 3, 4 }`
- `skup.discard( 4 )`
- `skup.discard( 4 )`
- `skup.remove( '2' )`
- `skup.pop()`

# Skupovi

- Izbacivanje duplikata iz liste

```
l = list(set(l))
```

# Skupovi

- Zadatak
  - Napisati funkciju `dupli(l)` koja ispisuje broj duplih elemenata u listi `l`

```
def dupli(l):  
    # create an empty dictionary to store element counts  
    counts = {}  
  
    # loop over each element in the list and count occurrences  
    for elem in l:  
        if elem in counts:  
            counts[elem] += 1  
        else:  
            counts[elem] = 1  
  
    # count number of elements with counts greater than 1  
    num_duplicates = sum(1 for count in counts.values() if count > 1)  
  
    # print the number of duplicate elements  
    print("Number of duplicates:", num_duplicates)
```

# Skupovi

- Operacije nad skupovima
- Ispitivanje pripadnosti – operator `in`
- Unija skupova – metoda `union (|)`
- Presjek skupova – metoda `intersection (&)`
- Razlika skupova – metoda `difference (-)`
- Jednakost skupova – operator `==`
- Ispitivanje nadskupa – metoda `issuperset`
- Ispitivanje podskupa – metoda `issubset`

# Skupovi

- Zadatak
  - Napisati funkciju `jaccard(a, b)` koja računa jaccardov indeks skupova `a` i `b`

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

```
jaccard([1, 2, 3], [3, 4, 5])  
0.2
```



# Mape



# Mape

- Neuređen skup parova ključ-vrijednost
  - Ključevi su jedinstveni
  - Vrijednosti ključeva ne moraju biti jedinstveni
- Definiranje elementa mape uključuje dodavanje ključa i istovremeno definiranje vrijednosti za ključ
  - Vrijednost ključa može se obrisati ili promijeniti
  - Vrijednost ključa može se lako dohvatiti

# Mape

- Stvaranje mape
  - Slično skupu, ali elementi su parovi ključ-vrijednost
  - Ključevi su jedinstveni, nema duplikata
  - Vrijednosti različitih ključeva mogu biti različitih i bilo kojih tipova
  - Ključevi mogu imati različite tipove
  - Ali samo neke – znakovni nizovi, brojevi i još neke



```
mapa = { 1: 'prvi', '2': 'drugi', 'treci': [3,2,1]}
```

# Mape

- Stvaranje prazne mape

```
mapa = {}
```

```
mapa = dict()
```

# Mape

- Dohvaćanje elemenata
  - Za zadani ključ dohvaća se vrijednost, slično listama

```
mapa = {'1000' : 1000}  
v1 = mapa['1000']  
v2 = mapa.get('1000')  
v3 = mapa.get('1000', -1)
```

- Definicija, promjena i uklanjanje parova ključ vrijednost
  - Slično listama

```
mapa['10'] = 1001  
del mapa['10']
```

# Mape

- Veličina mape = broj parova ključ-vrijednost – funkcija `len`
- Popis svih ključeva mape – metoda `keys`
- Popis svih vrijednosti – metoda `values`
- Postojanje ključa u mapi – operator `in`

```
len(mapa)
```

```
mapa.keys()    # python3: list(mapa.keys())
```

```
mapa.values()  # python3: list(mapa.values())
```

```
'1000' in mapa
```

# Mape

- Još ugrađenih funkcija

`.items()`

`.pop(x)`

`.update(d)`

# Mape

## ■ Zadatak

### ■ Definirati mapu:

```
CITIES = {  
    'zg': 790000,  
    'st': 167000,  
    'ri': 128000,  
    'os': 84000  
}
```

### ■ Napisati funkcije

- `add_city(cities, name, population)`
  - dodaje grad i broj stanovnika u mapu `cities`
- `remove_city(cities, name)`
  - briše grad iz mape `cities`
- `max_city(cities)`
  - Vraća najveći broj stanovnika (ne ime grada, nego broj stanovnika)
  - Ugrađene funkcije `min(list)`, `max(list)`
- `min_city(cities)`
  - Vraća najmanji broj stanovnika

```
CITIES = {  
    'zg': 1,  
    'st': 2  
}  
  
def add(cities: map, name: str, pop: int):  
    cities[name] = pop  
    return cities  
  
def remove(cities : map, name : str):  
    if name in cities:  
        del cities[name]  
    return cities  
  
a = add(CITIES, "a", 1)  
print(a)  
b = remove(CITIES, "a1")  
print(b)  
  
print(max(CITIES.values()))  
print(min(CITIES.values()))
```



# Mape

- Zadatak

- Napisati funkciju `max_city_name(cities)`

- Vraća ime grada s najvećim broj stanovnika

- Koristiti

- `dict.values()`

- `dict.keys()`

- `list.index(element)`

- `max(list)`

- Napomena: `dict.keys()` i `dict.values()` vraćaju ključeve, odnosno vrijednosti, a njihove pozicije odgovaraju parovima ključ-vrijednost iz mape

# List vs Map (Set)



# List vs Map (Set)

## ■ List

- Kada je potrebno čuvati redoslijed
- Dohvaćanje preko cjelobrojnog indeksa
- Prosječne vremenske složenosti
  - `append: O(1)`
  - `insert: O(n)`
  - `get/set: O(1)`
  - `delete: O(n)`
  - `x in s: O(n)`

## ■ Dict

- Ne osigurava redoslijed
- Dohvaćanje preko proizvoljnog ključa
- Prosječne (amortizirane) vremenske složenosti
  - `get/set: O(1)`
  - `delete: O(1)`
  - `x in s: O(1)`

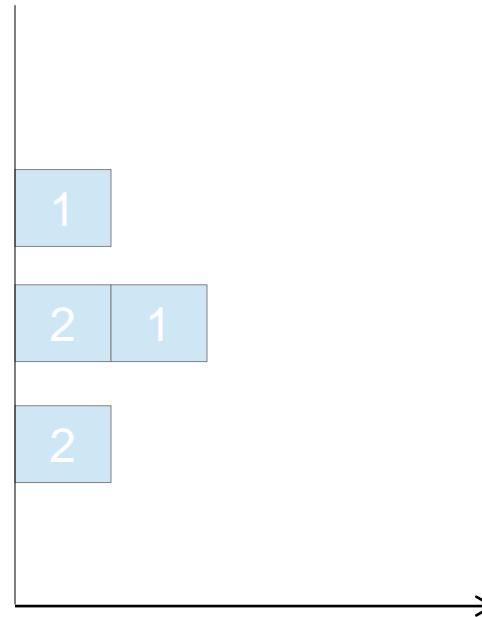
# Red kao List/Deque



# Red kao List/Deque

- Red se može implementirati preko `list`
- Koriste se metode `insert/pop`

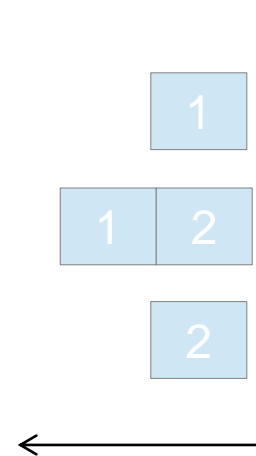
```
l = []  
l.insert(0, 1)  
l.insert(0, 2)  
l.pop()  
l.pop()
```



# Red kao List/Deque

- Obična Python lista
  - Operacije uglavnom  $O(n)$
- Ugrađena kolekcija `deque`
  - Vrlo učinkovita, npr. `pop`  $\rightarrow O(1)$

```
from collections import deque  
q = deque()  
q.append(1)  
q.append(2)
```



# Programska potpora komunikacijskim sustavima

• Dr. sc. Adrian Satja Kurdija

## Programski jezik Python - 3. predavanje



# Sadržaj predavanja

- Funkcije
- Stringovi
- For petlja, range
- Komprehenzija
- Rad s datotekama
- Stdin/stdout
- Doseg varijabli



# Funkcije



# Funkcije

- Pomoćni dio programa koji: prima nula ili više objekata (npr. niz brojeva), obavlja neki zadatak i (neobavezno) vraća rezultat dijelu programa koji je pozvao funkciju
- Primjer: funkcija koja pretvara broj u niz znamenaka

```
def f(x):
```

```
    znamenke = []
```

```
    while x > 0:
```

```
        znamenke.append(x % 10)
```

```
        x //= 10
```

```
    return znamenke
```

```
...
```

```
n = int(input())
```

```
lista = f(n)
```

# Stringovi



# Stringovi

- Znakovni nizovi - navode se unutar jednostrukih ili dvostrukih navodnika  
`ime = 'Joe'`
- Dohvaćanje elementa (`ime[k]`), slicing, `len` isto je kao za listu
- Brojenje/traženje znaka ili podstringa:

`s.count(podriječ), s.index(podriječ)`

- *Immutable tip* - Ne možemo mijenjati pojedini znak; ako to želimo stvaramo novi string:

`s = s[0:5] + 'A' + s[6:]` # 5. znak postaje A

# Stringovi

- Rastavljanje u listu stringova prema separatoru:

`s.split()`

`s.split(znak)`

`s.split(',')`

- Spajanje više stringova u jedan:

`<separator>.join(niz_stringova)`

`>>> ' ? '.join(['prva', 'druga', 'treca'])`

`'prva ? druga ? treca'`

# Stringovi

- Stringove odvojene razmakom (ili nekim drugim separatorom) možemo učitati u listu stringova:

```
a = input().split()
```

- Ako je riječ o brojevima:

```
a = list(map(int, input().split()))
```

- Tri broja u istom retku:

```
x, y, z = map(float, input().split())
```

# Zadatak

Napisati funkciju koja prima rečenicu te vraća njenu **kraticu** tako da redom za svaku riječ iz rečenice uzmemo prvo slovo, osim ako riječ ima jedno ili dva slova. Kraticu pišemo velikim slovima. (npr. Public relations → PR)

- `str.split()`
- `str.upper()`, `str.lower()`

# Zadatak

- Napisati funkciju koja prima rečenicu te vraća postotak riječi u rečenici koje nisu veznici ('i', 'pa', 'te', 'ni', 'niti', 'a', 'ali', 'nego', 'no', 'ili')
- Napisati funkciju koja prima rečenicu te vraća broj slova (duljinu riječi) za sve riječi u rečenici koje nisu veznici
  - Izlaz je mapa gdje su ključevi riječi, a vrijednosti duljina riječi



# Vježba

- Napisati funkciju koji iz dane riječi izbacuje sve samoglasnike.
- Napisati funkciju koja prima string te vraća novi string tako da su sva mala slova zamijenjena velikim slovima, a sva velika slova zamijenjena malim slovima.
- Napisati funkciju koji će danu riječ ispisati tako da ubaci razmak između svakih dvaju susjednih slova riječi (npr. *Python* → *P y t h o n*).
- Napisati funkciju koji učitava dva stringa te odgovara na pitanje može li se drugi string dobiti premetanjem znakova prvog stringa.

# Zadatak

- Pomoću naredbe `help(str)` i `help(str.metoda)` proučiti ostale metode string objekata
- Napisati funkciju koja prima rečenicu i vraća istu rečenicu, ali
  - Bez višestrukih praznina
  - Ispravljenog prvog početnog slova (ako je dano malim slovom)
  - Dodana točka na kraj (ako ne postoji)
  - npr. **“ovo je neki tekst”** → **“Ovo je neki tekst.”**

# Range, komprehenzija



# For petlja i range

- Funkcija range generira  $n$ -torku cijelih brojeva
  - Dohvaćanje raspona brojeva
  - Po njemu je moguće iterirati for petljom`range(pocetak, kraj, korak)`
- Na primjer, sljedeći odsječak koda generira brojeve 1, 3, 5, ..., 19 i ispisuje ih:

```
for i in range(1, 20, 2):  
    print(i)
```
- Moguće je navesti samo *kraj* (za niz 0, 1, 2, ..., kraj - 1):

```
for i in range(10):  
    print(i)
```
- Primjer - obilazak liste:

```
for i in range(len(lista)):  
    print(i, lista[i])
```

# Komprehenzija

- Želimo stvoriti listu na osnovi nekog postojećeg niza
- “Dulji” način: for petljom prođemo po postojećem nizu i dodajemo elemente u novu listu (*append*)

```
for i in range(10):
```

```
    novaLista.append('Number ' + str(i))
```

- Kraći način: *list comprehension*

```
nova = ['Number ' + str(i) for i in range(10)]
```

- Općenito: [**<izraz> for <varijabla> in <niz>**]

- Primjeri:

- unos više redaka podataka: `redci = [input() for i in range(n)]`

- pretvorba stringa u listu slova:

```
h_letters = [letter for letter in 'human']
```

```
print(h_letters)
```

# Datoteke, stdin/stdout



# Rad s datotekama

- Čitanje/pisanje u datoteku

`f = open(path, 'r|w')` - otvara datoteku za čitanje/pisanje

`f.read()` - vraća cijeli sadržaj kao string

`f.readline()` - vraća idući redak kao string

`f.readlines()` - vraća retke kao listu stringova

`f.write()` - zapisuje string u datoteku

`f.writelines()` - zapisuje listu stringova kao retke

`f.close()` - zatvara datoteku

# Rad s datotekama

- Iteracija po datoteci redak po redak

```
with open(path) as f:  
    for line in f:  
        print(line)
```



# Stdin/stdout

- Prilikom pokretanja programa iz komandne linije (Terminal, Command Prompt, PowerShell) možemo datoteku s ulaznim podacima preusmjeriti na standardni ulaz (stdin) operatorom redirekcije ulaza (znak <):

```
$ python obrada.py < podatci.txt
```

- Tada program učitava podatke kao “s tipkovnice” (*input()* i slično)
- Za ispis na standardni izlaz (stdout) koristimo `print()`, ali i njega možemo preusmjeriti u datoteku (znak >):

```
$ python obrada.py < podatci.txt > izlaz.txt
```

# Vježba

- Napišimo program koji iz tekstualne datoteke učitava retke s podacima oblika:
  - *ime prezime, OIB*
- Program treba ispisati:
  - broj osoba (redaka),
  - osobe sortirane po prezimenu silazno,
  - najčešće ime.
- Program napišimo kao funkciju koja prima ime tekstualne datoteke iz koje čita podatke.
- Prilagodimo funkciju da ako dobije prazno ime datoteke, čita sa standardnog ulaza.

# Doseg varijabli - primjer

```
def scope_test():
    def do_local():
        x = "local x"

    def do_nonlocal():
        nonlocal x
        x = "nonlocal x"

    def do_global():
        global x
        x = "global x"

    x = "test x"
    do_local()
    print("After local assignment:", x)    # ispis: test x
    do_nonlocal()
    print("After nonlocal assignment:", x) # ispis: nonlocal x
    do_global()
    print("After global assignment:", x)   # ispis: nonlocal x

scope_test()
print("In global scope:", x)              # ispis: global x
```

# Programska potpora komunikacijskim sustavima

• Dr. sc. Adrian Satja Kurdija

**Programski jezik  
Python - 4. predavanje  
(Objektno orijentirano  
programiranje)**



# Sadržaj predavanja

- Moduli
- Paketi
- Klase
- Principi objektno orijentiranog programiranja:
  - nasljeđivanje
  - polimorfizam
  - enkapsulacija
  - apstrakcija

# Moduli, paketi



# Moduli

- Mnogi Python programi započinju import naredbama
- Time se uključuje neki drugi Python kod ("naš" ili "tuđi")
- Sintaksa:

```
import moj_modul
```

--> objekte dohvaćamo s prefiksom (`moj_modul.funkcija`)

```
from moj_modul import *
```

```
from moj_modul import funkcija2, klasa3
```

- Moduli standardne biblioteke: *os, sys, math, pickle, time, collections...*
  - instalirani zajedno s Python interpreterom
- Korišćenje kraćeg imena:

```
import numpy as np
```

# Paketi

- Modul je neka .py datoteka; paket je direktorij modula
- Sintaksa:

```
import paket.modul  
import scipy.stats  
scipy.stats.variation(a)
```

```
from paket import modul  
from scipy import stats  
stats.variation(a)
```

```
from paket.modul import objekt  
from scipy.stats import variation  
variation(a)
```



# Paketi

- Instalacija paketa iz Python Package Index (PyPI):

```
pip install ime_paketa
```

```
pip install ime_paketa==verzija
```

```
pip install --upgrade ime_paketa
```

- Provjera instaliranih paketa:

```
pip freeze
```

```
pip show ime_paketa
```

- Deinstalacija paketa:

```
pip uninstall ime_paketa
```

# Import putanje

- Import će uspjeti ako se direktorij modula/paketa nalazi u nizu sys.path
- To će biti zadovoljeno:
  - ako je modul dio standardne biblioteke
  - ako je instaliran nekim od standardnih alata (npr. pip, apt)
  - ako se nalazi u istom direktoriju kao i pokrenuti program (kao dio istog projekta)
- Ako se nalazi negdje drugdje, možemo ažurirati niz sys.path:

```
import sys
print(sys.path)
sys.path.append('/home/adrian/my_helper_modules')
import my_module
```

# Import putanje

- Tipična situacija:
  - glavni program je *main.py*
  - pomoćni kodovi nalaze se u poddirektorijima (npr. *data\_models/myClass.py*) istog direktorija

```
from data_models.myClass import ImeKlase
```

(unutar *main.py*)

# Vježba

- U modulu *pomocni.py* napisati funkciju `median(x, y, z)` koja vraća srednji po veličini od triju brojeva.
- Pozvati funkciju iz programa *glavni.py* u istom direktoriju/projektu.
- Modul premjestiti u neki poddirektorij istog projekta.
- Modul premjestiti u neki direktorij izvan projekta (npr. Desktop).

# Klase



# Klase

- Klasa je skup organiziranih podataka i operacija nad tim podatcima
  - npr. račun u trgovini
- Konkretni primjerak klase je **objekt**
- Podatci unutar klase zovu se **atributi**  
*objekt.ime\_atributa*  
*moj\_racun.artikli*
- Funkcije unutar klase zovu se **metode**  
*objekt.ime\_funkcije(...)*  
*moj\_racun.dodaj\_artikl(bajadera)*

# Klase

- Definiranje klase:

- Ključna riječ `class` + ime klase + dvotočka

- ```
class Trokut:
```

- ```
...
```

- Izvode se unutrašnje naredbe, obično definicije atributa i metoda

- Stvaranje objekta:

- ```
ime_objekta = ImeKlase(...parametri...)
```

- > poziva se metoda `__init__` unutar klase koja prima navedene parametre (nula ili više njih) te postavlja attribute klase na početne vrijednosti

# Primjer klase

```
class Zaposlenik:
    def __init__(self, ime_i_prezime, placa):
        self.ime, self.prezime = ime_i_prezime.split()
        self.placa = placa

    def daj_povisicu(self, postotak):
        self.placa = self.placa * (1 + postotak / 100)

    def porez(self):
        return 0.24 * self.placa

ana = Zaposlenik("Ana Anic", 9000)
ana.daj_povisicu(15)
print(ana.porez())
```



# Parametar *self*

- Članske metode na prvom mjestu trebaju imati definiran parametar imena *self*
- On je referenca na objekt unutar kojeg se izvodi metoda
- Pruža pristup metodama i atributima trenutnog objekta, a ne nekog drugog objekta
- Kad pozivamo metodu, ne zadajemo vrijednost za parametar *self* već interpreter automatski definira tu referencu:

`objekt.metoda(podatak1, podatak2)`

prevodi se u

`metoda(self=objekt, podatak1, podatak2)`

# Neobavezni (*optional*) parametri

- Parametri koji se prilikom poziva metode ne moraju zadati
- Funkciji se onda prosljeđuju zadane *default* vrijednosti

```
def __init__(self, ime_i_prezime, placa=6000):  
    ...
```

- Atribut se može definirati/promijeniti poslije

```
pero = Zaposlenik("Pero Peric")  
pero.placa = 7000  
print(hasattr(pero, 'placa')) # ispis: True
```

# Statički atributi i metode

- Statička metoda ne odnosi se na određeni objekt
- Definira se bez parametra `self`
- Poziva se kao `ImeKlase.imeMetode(...)`
- Statički atributi ne odnose se na određeni objekt, nego na klasu općenito
- Definiraju se bez `self`-a, ispod definicije klase, izvan svih metoda
- Pristupamo im kao `ImeKlase.imeAtributa`

# Statički atributi i metode

- Primjer: klasi *Zaposlenik* dodajmo statičku metodu `izbroji()` koja vraća broj objekata - konkretnih zaposlenika

```
class Zaposlenik:
    brojac = 0
    def __init__(self, ime_i_prezime, placa):
        Zaposlenik.brojac += 1
        ...
    def izbroji():
        return Zaposlenik.brojac
    ...
print(Zaposlenik.izbroji())
```

# Posebne metode klase

- Python magic methods - ne pozivaju se eksplicitno, nego automatski u određenim slučajevima (ako ih definiramo)
- Predefinirana imena oblika `__metoda__`
- Najčešći primjeri:
  - `__init__`
  - `__str__`
  - `__add__`, `__mul__`, `__sub__`, `__div__`
  - `__cmp__`, `__eq__`, `__ne__`, `__lt__`, `__gt__`
  - `__getitem__`, `__setitem__`

# Vježba

- Krenimo od klase *Zaposlenik* kao što je definirana na jednom od prethodnih slajdova.
- Zadavanjem *default* plaće u `__init__` funkciji omogućimo stvaranje zaposlenika definirajući mu samo ime i prezime.
- Dodatno ostvarimo da, u slučaju plaće koja nije zadana, uopće ne definiramo odgovarajući atribut.
- Klasi *Zaposlenik* dodajmo statičku metodu `izbroji()` koja vraća broj objekata - konkretnih zaposlenika.
- Implementirati metodu `__str__` tako da npr. poziv `print(ana)` ispiše ime, prezime i plaću zaposlenika.

# Principi objektno orijentiranog programiranja



# Nasljeđivanje

- U zagradi je moguće definirati nadklasu (ili više njih):

```
class Zaposlenik(Osoba):
```

- Tada podklasa nasljeđuje sve attribute i metode nadklase (uz vlastite)
- Klasa *Zaposlenik* nasljeđuje i proširuje klasu *Osoba*
  - svaki *Zaposlenik* je *Osoba* (obrat ne vrijedi)



# Nasljeđivanje: primjer (1/2)

```
class Poligon:
    def __init__(self, broj_stranica):
        self.n = broj_stranica

    def unesi_duljine_stranica(self):
        self.stranice = [float(input()) for i in range(self.n)]

    def opseg(self):
        return sum(self.stranice)

p = Poligon(5)
p.unesi_duljine_stranica()
print(p.opseg())
```

# Nasljeđivanje: primjer (2/2)

```
class Trokut(Poligon):
    def __init__(self):
        self.n = 3
        # alternative:
        # Poligon.__init__(self, 3)
        # super().__init__(self, 3)

    def površina(self):
        a, b, c = self.stranice
        s = (a + b + c) / 2
        return (s*(s-a)*(s-b)*(s-c)) ** 0.5

t = Trokut()
t.unesi_duljine_stranica()      # naslijeđena metoda!
print(t.opseg())                # naslijeđena metoda!
print(t.površina())
```

# Vježba

- Nasljeđivanjem omogućiti da neki zaposlenici budu studenti, kojima se porez računa kao 10% umjesto 24% od plaće.
- Napisati drugu klasu *Poduzece* čiji su atributi ime i lokacija poduzeća te lista zaposlenika. Klasa treba metodama podržati:
  - dodavanje novog zaposlenika,
  - davanje povišice svim zaposlenicima za zadani postotak,
  - računanje ukupnog poreza na plaće zaposlenika,
  - izbacivanje zaposlenika.
- Implementirati metodu `__str__` koja omogućuje da npr. poziv `print(moje_poduzece)` ispiše podatke o poduzeću i zaposlenicima.

# Polimorfizam

- Princip koji omogućuje da neku klasu koristimo kao i njezinu nadklasu
- Objekti podklase formalno pripadaju i nadklasi
- Objekt klase *Student* može bilo gdje zamijeniti objekt klase *Zaposlenik* i odigrati njegovu ulogu jer je naslijedio sve njegove metode
- Npr. imamo listu zaposlenika poduzeća u kojoj su i neki studenti
- Svaku metodu koju treba pozvati na svim zaposlenicima (npr. davanje povišice) moguće je izvesti i na onima koji su studenti

```
branko = Student("Branko Brankic", 8000)
print(type(branko))           # <class '__main__.Student'>
print(isinstance(branko, Zaposlenik)) # True
```

# Enkapsulacija

- **Enkapsulacija:** "skrivamo" attribute klase, smatramo ih privatnim i ne pristupamo im izvan tijela klase
- Stavljamo dvije donje crte ispred imena (npr. `self.__placa`)
- Smanjuje mogućnost pogreške
- Smanjuje međuovisnosti različitih komponenata
- Metode također mogu biti privatne - pozivaju se samo interno

# Apstrakcija

- **Apstrakcija:** javne su samo one metode koje su nam potrebne da izvana koristimo klasu
- Primjer:

```
class Zaposlenik:  
    ...  
    def __obavijesti_racunovodstvo(self):  
        ...  
    def postavi_placu(self, vrijednost):  
        self.__placa = vrijednost  
        self.prirez = 0.18 * self.__placa  
        self.__obavijesti_racunovodstvo()
```

# Programska potpora komunikacijskim sustavima

• Dr. sc. Adrian Satja Kurdija

## Programski jezik Python - 5. predavanje



# Sadržaj predavanja

- Iteratori
- Iznimke
- Serijalizacija
- Interakcija s operacijskim sustavom
- HTTP zahtjevi



# Iteratori



# Iteratori

- Iteratori se koriste u for petljama i komprehenzijama
- Iterator predstavlja strukturu od više elemenata koju je moguće obići element po element
- Liste, skupovi, n-torke i mape su iteratori (između ostalog)
- Ako nam klasa predstavlja niz podataka koji želimo obići npr. ovako:  
`for artikl in moj_racun: ...`  
... onda u klasi treba definirati metode `__iter__` i `__next__`
- Metoda `__iter__` stvara iterator
  - obično inicijalizira neki brojač i vraća `self`
- Metoda `__next__` vraća idući element
  - npr. `self.artikli[self.brojac]`

# Iteratori

- Kako radi for petlja?
- Kad napišemo:

```
for i in range(3): ...
```

... u pozadini se poziva:

```
it = range(3).__iter__()          # ili it = iter(range(3))  
i = it.__next__() --> vraća 0     # ili i = next(it)  
i = it.__next__() --> vraća 1  
i = it.__next__() --> vraća 2  
i = it.__next__() --> vraća iznimku StopIteration
```

# Klasa kao iterator

## ■ Primjer - Fibonaccijevi brojevi:

```
class Fib:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 1
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

```
for x in Fib(100):
    print(x)
```

*Ispis:*

1 1 2 3 5 8 13 21 34 55 89

# Klasa kao iterator

- **Zadatak**
- Omogućiti iteriranje po zaposlenicima poduzeća iz prethodnog zadatka, tako da je moguće izvesti npr.:

```
for zaposlenik in moje_poduzece:  
    print(zaposlenik)
```

# Iznimke



# Iznimke

- Iznimke su sintaksne, semantičke ili namjerne greške prilikom izvođenja programa
- Npr. dijeljenje nulom, korištenje nepostojećeg objekta ili datoteke, pristup nepostojećem elementu niza ili rječnika...
- To su klase koje nasljeđuju klasu *Exception*
- Po defaultu uzrokuju prekid izvršavanja i ispis odgovarajuće poruke:

```
>>> '2' + 2
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: Can't convert 'int' object to str implicitly
```

# try i except blokovi

- Ako ne želimo prekid izvršavanja, iznimku možemo prepoznati unutar bloka `try`:

```
while True:
    try:
        x = int(input("Unesite broj: "))
        break
    except ValueError:
        print("Neispravan broj! Pokusajte ponovno.")
```

- Možemo prepoznati više vrsta iznimki:

```
except (RuntimeError, TypeError, NameError):
    pass # ignoriranje iznimke - ne preporučuje se!
```



# Višestruki `except` blokovi

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print('I/O error: {0}'.format(err))
except ValueError:
    print('Could not convert data to an integer.')
except:
    print('Unexpected error:', sys.exc_info())
```

# Uzrokovanje iznimki

- Naredba raise izaziva bilo koju iznimku:  
`>>> raise NameError('HiThere')`
- Možemo definirati vlastite iznimke:

```
class MyError(Exception):  
    def __init__(self, value):  
        self.value = value  
    def __str__(self):  
        return "MyError with value " + str(self.value)  
  
try:  
    raise MyError(5)  
except MyError as e:  
    print(e)
```

# Iznimke - vježba

- Definirati tip iznimke `ErrZap`
  - inicijalno bez ikakvih metoda (npr. `pass` umjesto tijela klase)
- Proširiti klasu `Poduzece` tako da baca iznimku `ErrZap` ako se:
  - dodaje zaposlenik koji je već u poduzeću
  - izbacuje zaposlenik koji nije u poduzeću
- Napisati klase `ErrZapDodaj` i `ErrZapIzbaci` koje nasljeđuju `ErrZap` i omogućuju razlikovanje gornjih dvaju slučajeva
  - Implementirati metode `__init__` i `__str__` radi ispisa poruke o detaljima greške
  - npr. `Greska pri izbacivanju: Ana Anic ne postoji!`
- “Uhvatiti” iznimke u glavnom programu (`try...except`)

# Serijalizacija



# Serijalizacija: `pickle`

- *Pickling* – serijalizacija Python objekta u string ili niz bajtova
- *Unpickling* – suprotna operacija

```
import pickle
```

```
>>> pickle.dumps((1,2))  
'(I1\nI2\ntp0\n.'
```

```
>>> pickled = pickle.dumps((1,2))  
>>> pickle.loads(pickled)  
(1, 2)
```

# Serijalizacija: `pickle`

- Binarna serijalizacija u datoteku (`dump`, `load`) koristi se unutar odgovarajućeg bloka:

```
with open(filename, "rb | wb") as file:
    pickle.dump(obj, file)    # za pisanje, ili
    obj = pickle.load(file)   # za citanje
```

- **Zadatak:** serijalizirati neki objekt klase `Poduzece` u datoteku. Potom u drugom programu učitati i ispisati serijalizirani objekt iz datoteke.

# Interakcija s operacijskim sustavom



# Modul **os**

- Portabilno sučelje prema operacijskom sustavu
- Primjer: traženje i brisanje datoteke u direktoriju:

```
import os
```

```
def findfile(start, name):  
    for dirpath, dirs, files in os.walk(start):  
        if name in files:  
            full_path = os.path.join(dirpath, name)  
            os.remove(full_path)
```



# Modul os

- Izvršavanje naredbe kao iz terminala:  
`os.system("firefox skripta.pdf")`
- Tako možemo pokrenuti bilo koji drugi program

- Fleksibilnija alternativa: **subprocess.run**

```
subprocess.run(["firefox", "skripta.pdf"])
```

```
subprocess.run(  
    ["/usr/bin/git", "commit", "-m",  
    "Fixes a bug."])
```

# Direktoriji i datoteke

`os.getcwd()`

putanja trenutnog direktorija

`os.chdir(dirname), os.chdir('..')`

navigacija: `cd`, `cd ..`

`os.listdir()`

lista - sadržaj direktorija: `dir` (win), `ls` (unix)

`os.mkdir(path)`

novi direktorij

`os.rename(old_path, new_path)`

preimenovanje ili premještanje

`os.path.basename(filepath)`

vraća ime datoteke bez putanje

`os.path.splitext(filepath)`

 odvajanje ekstenzije, vraća (file, ext) npr. `(' /home/datoteka', '.txt')`

# Modul os: vježba

- Napisati program koji:
  - pronalazi abecedno prvu datoteku u zadanom direktoriju,
  - stvara novi poddirektorij `tmp`,
  - premješta datoteku u `tmp` ali joj mijenja ime u `prva` (zadržava ekstenziju).
- Npr. datoteka `/home/adrian/Desktop/fotka.jpg` postaje `/home/adrian/Desktop/tmp/prva.jpg`
- Koristiti:
  - `os.listdir(path)`
  - `os.path.isdir(filepath)`
  - `os.path.basename(filepath)`, `os.path.splitext(filepath)`
  - `os.mkdir(path)`
  - `os.path.join(path, file)`
  - `os.rename(old, new)`

# HTTP zahtjevi



# HTTP zahtjevi

- Modul **requests**
- Pojednostavljuje slanje HTTP zahtjeva
  - zaglavlja (*headers*), forme, datoteke
  - parametri su mape (*dict*)
  - povratne vrijednosti su također Python objekti
- Korištenje:
  - `pip install requests`
  - `import requests`
  - `r = requests.get(...)`
  - `r = requests.post(...)` # *put, delete, ...*
  - `r` je Response objekt
    - `r.status_code`, `r.content`, `r.text`, `r.headers`

# HTTP zahtjevi: GET

```
r = requests.get('https://google.hr')  
print(r.text)
```

```
query = {'lat': '45', 'lon': '180'}  
r = requests.get(  
    'http://api.open-notify.org/iss-pass.json',  
    params=query)  
print(r.headers)  
print(r.json())
```

```
url = 'https://api.github.com/some/endpoint'  
r = requests.get(url,  
    headers={'user-agent': 'my-app/0.0.1'})
```

# HTTP zahtjevi: POST

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.post("https://httpbin.org/post",
                        data=payload)

>>> r.text
{
    ...
    "form": {
        "key2": "value2",
        "key1": "value1"
    },
    ...
}
```

# HTTP zahtjevi: POST

```
>>> url = 'https://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}
>>> r = requests.post(url, files=files)
>>> r.text
{
    ...
    "files": {
        "file": "<censored...binary...data>"
    },
    ...
}
```



# HTTP zahtjevi

- Bad requests

```
bad_r = requests.get('http://httpbin.org/status/404')
>>> bad_r.status_code
404
>>> bad_r.raise_for_status()
requests.exceptions.HTTPError: 404 Client Error
```

# HTTP zahtjevi: vježba

- Napisati program koji (uz pomoć GET zahtjeva) ispisuje imena astronauta koji se trenutno nalaze u ISS
- API: <http://api.open-notify.org/astros.json>
  - koristiti `response.json()` kao mapu (dict)
  - dokumentacija: <http://open-notify.org/Open-Notify-API/People-In-Space/>

# Programska potpora komunikacijskim sustavima

• Dr. sc. Adrian Satja Kurdija

## Programski jezik Python - 6. predavanje



# Sadržaj predavanja

- Paralelizam
  - Višedretvenost
  - Višeprocesnost
- Mrežno programiranje: *socket API*

# Paralelizam: višedretvenost



# Procesi vs. dretve

- Proces
  - Program koji se izvršava
  - Ima vlastiti adresni prostor, memoriju, stog podataka...
  - Visoka izolacija
- Dretve (engl. *threads*)
  - Izvršavaju se unutar istog procesa
  - Niska izolacija
    - Dretve dijele isti kontekst
    - Programeri se moraju brinuti za sinkronizaciju na zajedničkim podatcima

# Višedretvenost

- Python GIL: *Global Interpreter Lock*
  - Samo jedna dretva može istodobno pristupiti Python interpreteru
    - jer većina interpretera nije thread-safe
  - Ako dretve sadrže čisti Python kod (tj. ako su CPU-bound), onda nema smisla koristiti dretve za ubrzanje
- Kada koristiti dretve u Pythonu?
  - Za ubrzanje koda koji koristi vanjske resurse ili poziva npr. C kod
  - Responzivna sučelja
    - Čekanje na neki događaj delegira se dretvi
  - Delegacija zadataka
    - Više dretvi + red poruka
  - Višekorisničke aplikacije
    - Npr. web poslužitelji

# Višedretvenost

```
import threading
import time

# Kod koji se izvršava u neovisnoj dretvi
def countdown(n):
    while n > 0:
        print('t-minus', n)
        n -= 1
        time.sleep(1)

# Stvori i pokreni dretvu
t = threading.Thread(target=countdown, args=(10,))
t.start() # eksplicitni početak
```



# Višedretvenost

- Upravljanje stanjem dretve

*# provjeri je li dretva živa*

**t.is\_alive()**

*# blokiraj trenutnu dretvu dok se dretva t ne završi*

**t.join()**

- Kritični odsječci (za dijeljeni pristup)

**lock = threading.Lock()**

**lock.acquire()**

**lock.release()**

... ili jednostavnije:

**with lock:**

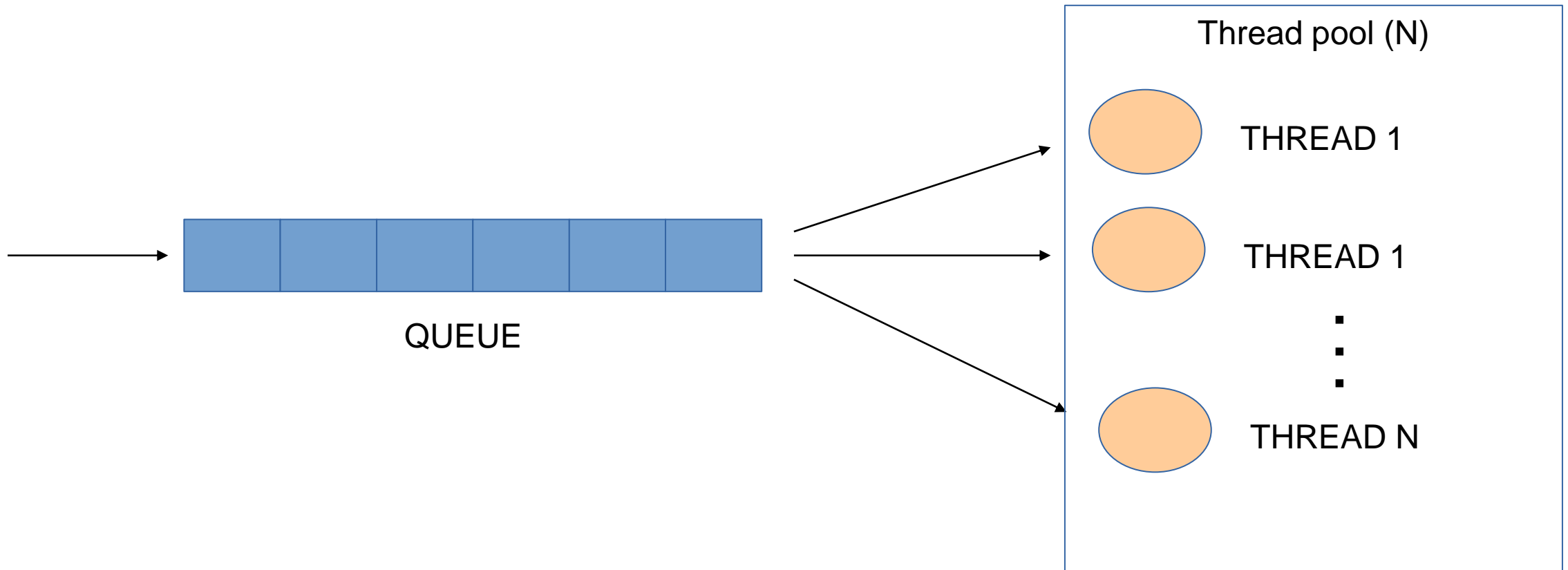
*# kritični odsječak...*

# Višedretvenost: *Queue*

- Dretve mogu komunicirati koristeći strukturu `Queue` iz modula `queue`
  - To je *thread-safe* struktura što znači da sama brine o sinkronizaciji (dijeljenom pristupu)
- Instanciranje: `red = queue.Queue()`
- Ubacivanje podatka u red: `red.put(value)`
- Vađenje (čekanje) podatka iz reda: `value = red.get()`

# Višedretvenost - primjer

- Bazen dretvi + red podataka



# Višedretvenost - primjer

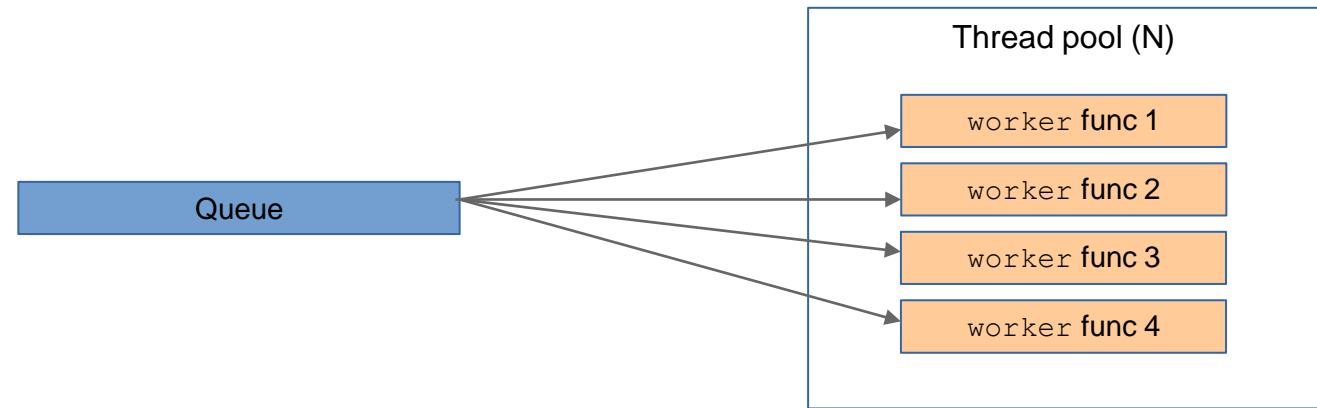
```
import threading
import queue
work_queue = queue.Queue()
```

- Stvaranje dretvi:

```
NUM_THREADS = 4
threads = [
    threading.Thread(target=worker, args=(work_queue,))
    for i in range(NUM_THREADS)
]
```

- Pokreni dretve

```
for thread in threads:
    thread.start()
```



# Višedretvenost - primjer

- Funkcija koju izvodi pojedina dretva:

```
def worker(work_queue):  
    while True:  
        item = work_queue.get()  
        # ... obradi item i ispisi rezultat ...  
        work_queue.task_done()
```

- Glavna dretva ubacuje podatke za obradu:

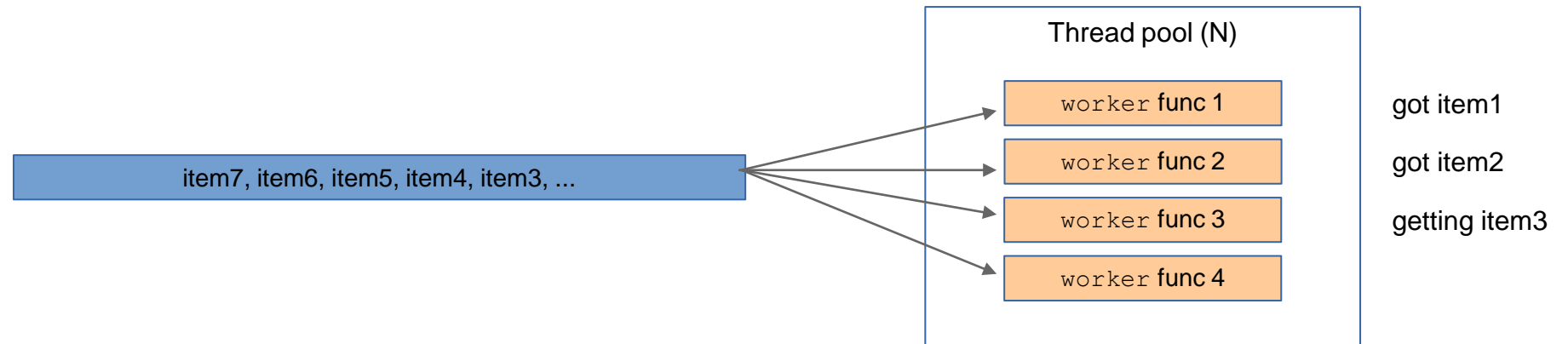
```
for i in items:  
    work_queue.put(i)
```

# Višedretvenost - primjer

- Čišćenje:

```
# čekaj dok se ne dobiju i ne obrade svi elementi reda  
work_queue.join()
```

```
# čekaj da sve dretve završe  
while threads:  
    threads.pop().join()
```

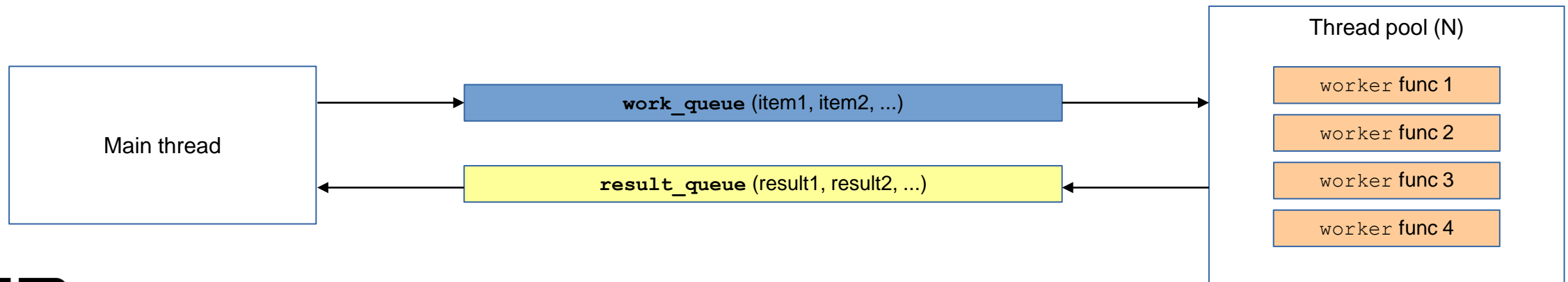


# Višedretvenost - vježba

- Problem: dretve se izvršavaju beskonačno dugo jer u petlji čekaju na novi element (`work_queue.get()`) iako ih više nema
- Zadatak: prepraviti kod tako da dretve prepoznaju kada su svi elementi obrađeni i završe.

# Višedretvenost - primjer

- Poboljšanje:
  - Ispis rezultata unutar dretvi nije praktičan (npr. nemamo kontrolu nad redoslijedom)
  - Najbolja je praksa tu odgovornost ostaviti glavnoj dretvi
  - Rješenje: dodaj još jedan red za rezultate





# Višedretvenost - primjer

- Stvori još jedan red:

```
results_queue = queue.Queue()
```

- Kreiraj bazen dretvi koje kao argumente primaju oba reda:

```
Thread(target=worker, args=(work_queue, results_queue))
```

- U worker funkciji, dodaj rezultat u red za rezultate:

```
results_queue.put(rezultat obrade itema)
```

- Ispiši sve rezultate:

```
while not results_queue.empty():  
    print(results_queue.get())
```

# Višedretvenost - vježba

- Ispisati HTML sadržaj jedne od web stranica Google i Bing, one koja prva odgovori na zahtjev.
  - Svaka od dvije dretve paralelno dohvaća jedan URL i rezultat sprema u red
  - Glavna dretva ispisuje prvi element iz reda rezultata čim se on pojavi (`red.get()`)
  - Dohvaćanje HTML-a u worker funkciji dretve:

```
import urllib.request
sadrzaj = urllib.request.urlopen(url).read()
```
  - Dohvaćeni HTML treba ispisati u datoteku
  - Na kraju pozvati naredbu (web browser) koja će ga otvoriti

# Paralelizam: višeprocenost



# Višeprocenost

- Za ubrzanje čistog (CPU-bound) Python koda
  - Nema GIL ograničenja na jednu jezgru
- Sustavski poziv `fork()`
  - Stvara se novi (izolirani) process
  - Nema dijeljenja konteksta

```
import os
child_pid = os.fork()
if child_pid == 0:
    print('Child Process: PID', os.getpid())
else:
    print('Parent Process: PID', os.getpid())
```

# Modul multiprocessing

- Elegantniji način stvaranja i pokretanja novog procesa (interno se poziva *fork*)

```
from multiprocessing import Process
```

```
def f(name):  
    print('hello', name)
```

```
if __name__ == '__main__':  
    p = Process(target=f, args=('bob',))  
    p.start()    # pokretanje  
    p.join()     # čeka završetak
```

# Primjer - multiprocessing

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())    # roditeljev ID
    print('process id:', os.getpid())        # moj ID

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

# Sinkronizacija (Lock)

```
from multiprocessing import Process, Lock

def f(lock, i):
    with lock:
        print('hello world')
        print(i)

if __name__ == '__main__':
    l = Lock()
    for num in range(10):
        Process(target=f, args=(l, num)).start()
```

# Komunikacija (Queue)

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())      # ispis: [42, None, 'hello']
    p.join()
```



# Vježba - zadatak

- Kreirajte dva procesa: jedan računa zbroj svih vrijednosti  $\sin(x)$  za  $x = 1, 2, \dots, 10^7$ , a drugi zbroj svih  $\cos(x)$  za  $x = 1, 2, \dots, 10^7$ . Glavni proces treba ispisati rezultat koji prije završi.

# multiprocessing.Pool

- Primjenu iste funkcije na više objekata paraleliziramo među procesima
- Smisleno je definirati onoliko procesa koliko ima jezgri procesora  
(`os.cpu_count()`)

```
from multiprocessing import Pool
```

```
def f(x):  
    return x * x
```

```
if __name__ == '__main__':  
    with Pool(4) as p:  
        argumenti = [1, 2, 3, 4, 5, 6, 7, , 9]  
        rezultati = p.map(f, argumenti)
```

# Višeprocесnost - vježba

- Napisati program koji što brže računa sumu kvadrata svih brojeva od 1 do  $N$ . Posao treba podijeliti tako da interval od 1 do  $N$  podijelimo na četiri dijela – manja intervala.
- Testirati program za velike  $N$  (npr.  $10^7$ ) te izmjeriti vrijeme izvođenja za rješenja s  $K = 1, 2$  i  $4$  procesa.

# Mrežno programiranje: *socket API*



# Socket API

- Sučelje za mrežnu komunikaciju (*Berkeley sockets*)
- Socket je „utičnica” za slanje i primanje podataka
- Stvaramo je modulom `socket` standardne biblioteke:

```
s = socket.socket(address family, socket type)
```

  - Za IPv4 adrese upotrebljavamo `socket.AF_INET`
  - Za TCP protokol upotrebljavamo `socket.SOCK_STREAM`
- Povezujemo je na određeno sučelje metodom:

```
s.bind( (HOST, PORT) )
```
- Omogućujemo joj prihvati konekcija metodom:

```
s.listen()
```
- Na klijentskoj strani:

```
s.connect( (SERVER_HOST, SERVER_PORT) )
```

# Socket API: primanje i slanje podataka

- Čekanje na konekciju: `s.accept()` - blokira izvođenje
- Vraća uređeni par `conn, addr`: novi *socket* koji odgovara dolaznoj konekciji, te adresu spojenog klijenta (*host, port*)
- Komunikaciju ostvarujemo preko novog socketa metodama:
  - `conn.recv(max_bytes)`
    - prima podatke kao niz bajtova
    - ako vrati prazan *bytes* objekt, klijent je zatvorio konekciju
  - `conn.send(bytes)`
    - vraća broj uspješno poslanih bajtova
      - možda je potrebno ponovno pozvati
    - opetovano pozivanje može se automatizirati:
      - `conn.sendall(bytes)`

# Socket API: primanje i slanje podataka

- Podatci se šalju i primaju kao niz bajtova – objekt tipa `bytes`
- Pretvorba stringa u bajtove:  
`my_bytes = str.encode(my_str)`
- Pretvorba bajtova u string:  
`my_str = my_bytes.decode()`
- Za proizvoljne Python objekte možemo koristiti:  
`pickle.dumps(obj)`
  - vraća niz bajtova`pickle.loads(bytes)`
  - vraća originalni objekt

# Socket API: server

- `s.close()` zatvara konekciju i socket
- Automatsko zatvaranje na kraju bloka:  
`with socket.socket(...) as s:`

...

- Primjer jednostavnog *echo* servera:

```
import socket
HOST = "127.0.0.1" # localhost
PORT = 65432
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```



# Socket API: klijent

- Odgovarajući klijent:

```
import socket
```

```
HOST = "127.0.0.1"    # server's hostname/IP
```

```
PORT = 65432          # server's port
```

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
    s.connect((HOST, PORT))
```

```
    s.sendall(b"Hello, world")
```

```
    data = s.recv(1024)
```

```
print(f"Received {data!r}")
```

- Zadatak: simulirati dulju komunikaciju slanjem 10 poruka uz čekanje od 1 sekunde nakon svake poruke.

# Socket API: višestruke konekcije

- Kako omogućiti posluživanje većeg broja aktivnih klijenata?
- Svakog klijenta obrađuje zasebna dretva:

```
def on_new_client(conn, addr):  
    with conn:  
        print(f"Connected by {addr}")  
        while True:  
            data = conn.recv(1024)  
            if not data:  
                break  
            conn.sendall(data)  
  
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:  
    s.bind((HOST, PORT))  
    s.listen()  
    while True:  
        conn, addr = s.accept()  
        t = Thread(target=on_new_client, args=(conn, addr))  
        t.start()
```

# Socket API: vježba (*chatroom*)

- Napišimo *chat-server.py* i *chat-client.py*
- Port zadajemo kao argument pri pokretanju iz komandne linije:  

```
$ python chat-server.py 12000
```

```
$ python chat-client.py 12000
```
- U programu ga čitamo iz liste argumenata:  

```
port = int(sys.argv[1])
```
- Klijent treba imati dvije dretve: jednu za upisivanje i slanje, drugu za primanje poruka
- Server održava globalnu listu konekcija da bi mogao proslijediti poruku (*broadcast*)