UNIVЕRZIТЕТ U ISТОČNОM SАRАЈЕVUetf+grb

ЕLЕKТRОТЕHNIČKI FАKULТЕТ

**Sah**

SEMINARSKI RAD IZ PARALELNIH RAČUNARSKIH SISTEMA

|  |  |
| --- | --- |
| Studenti: | Predmetni profesor: |
| Dusan Marilovic  Goran Milanovic | Prof. dr Nikola Davidovic |

Istočno Sarajevo, jun, 2025

Inhalt

[1. Uvod 3](#_Toc202173873)

[2. Istorijski pregled jezika 4](#_Toc202173874)

[3. Programski jezik Python 5](#_Toc202173875)

[3.1 Komentar 5](#_Toc202173876)

[3.2 Promjenljive 6](#_Toc202173877)

[3.3 Dodjela vrijednosti 7](#_Toc202173878)

[3.4 Izrazi 7](#_Toc202173879)

[**3.4.1** **Evaluacija izraza** 7](#_Toc202173880)

[**3.4.2** **Operatori** 8](#_Toc202173881)

[3.5 Osnovne strukture podataka 9](#_Toc202173882)

[3.6 Grananje 9](#_Toc202173883)

[3.7 Iteracija 10](#_Toc202173884)

[3.8 Funkcije 11](#_Toc202173885)

[**3.8.1** **Ugradjene Funkcije** 11](#_Toc202173886)

[**3.8.2** **Korisnicke Funkcije** 11](#_Toc202173887)

[3.9 Interakcija korisnika 12](#_Toc202173888)

[4. Objektno Orjentisano programiranje 13](#_Toc202173889)

[4.1 Klasa i objekat 13](#_Toc202173890)

[4.2 Kreiranje objekta 14](#_Toc202173891)

[4.3 Definisanje korisnicke klase 14](#_Toc202173892)

[4.4 Kreiranje objekta pomocu konstruktora 14](#_Toc202173893)

[4.5 Pristup clanovima objekta 14](#_Toc202173894)

[4.6 Parametar self 14](#_Toc202173895)

[5. Moduli u programskom jeziku Python 14](#_Toc202173896)

[6. PyGame 14](#_Toc202173897)

[7. Paralelizacija u jeziku Python 14](#_Toc202173898)

[7.1 Kreiranje niti 15](#_Toc202173899)

[7.2 Kreiranje niti pomocu threading modula 16](#_Toc202173900)

[7.3 Sinhronizacija niti 18](#_Toc202173901)

[8. Sah 19](#_Toc202173902)

[8.1 Kreiranje table 19](#_Toc202173903)

[8.2 Chess Engine 20](#_Toc202173904)

[8.3 GameState 20](#_Toc202173905)

[**8.3.1** Konstruktor 21](#_Toc202173906)

[**8.3.2** **Zakoni kretanja figura** 21](#_Toc202173907)

[**8.3.3** **Kretanje po povrsi** 21](#_Toc202173908)

[**8.3.4** **Moguci i legalni potezi** 22](#_Toc202173909)

[**8.3.5** **Pinovi I sahovi** 23](#_Toc202173910)

[**8.3.6** **Specijalne kretnje** 23](#_Toc202173911)

[8.4 Klasa Move 24](#_Toc202173912)

[9. Montekarlo algoritam 24](#_Toc202173913)

[9.1 Uvod 24](#_Toc202173914)

[9.2 Osnovne faze Monte Carlo Tree Search algoritma 25](#_Toc202173915)

[9.3 Matematičke osnove: UCT formula 25](#_Toc202173916)

[9.4 Konkretna primjena 26](#_Toc202173917)

[9.5 Prednosti i izazovi MCTS algoritma 26](#_Toc202173918)

[9.6 Moguća poboljšanja i budući pravci razvoja 26](#_Toc202173919)

[10. Testiranje 27](#_Toc202173920)

[11. Zakljucak 27](#_Toc202173921)

# Uvod

Kroz istoriju razvoja računarskih sistema, kontinuirano se javljala potreba za ubrzanjem izvršavanja programa i povećanjem efikasnosti obrade podataka. Već tokom pedesetih godina prošlog vijeka započeta su prva istraživanja i eksperimentisanja sa višenitnim (multithreading) izvršavanjem, ali se istinsko razumijevanje i razvoj ovog koncepta intenziviralo tek krajem šezdesetih godina, kada je IBM 1968. godine prvi detaljnije istražio mogućnosti simultanog izvođenja više niti. Vremenom, kako je paralelizam na nivou instrukcija dostigao svoj maksimum krajem devedesetih godina, pažnja istraživača i inženjera usmjerava se ka višem nivou paralelizacije — onom na nivou procesa i niti, odnosno višenitnog izvršavanja.

Višenitno izvršavanje danas predstavlja ključnu tehniku za postizanje visokih performansi, posebno u aplikacijama koje uključuju kompleksne izračune ili simultane aktivnosti. Osobito je korisno u domenu razvoja igara i inteligentnih sistema za donošenje odluka, gdje paralelna obrada omogućava brže reagovanje, prirodniji tok igre i naprednije ponašanje agenata u stvarnom vremenu.

Cilj ovog projekta jeste prikaz praktične implementacije paralelizacije u Python programskom jeziku korištenjem višenitnog izvršavanja, u kontekstu igre šaha, pri čemu se posebna pažnja posvećuje primjeni Monte Carlo algoritma za donošenje odluka. Umjesto klasične igre sa unaprijed definisanim pravilima ponašanja računara, fokus ovog rada je na kreiranju sistema koji koristi stohastičke metode kako bi inteligentno birao poteze na osnovu probabilističke procjene mogućih ishoda partije.

Kao razvojno okruženje korišten je programski jezik Python zbog svoje fleksibilnosti, široke podrške za paralelno programiranje (putem biblioteka kao što su threading, multiprocessing i concurrent.futures) te bogatog ekosistema za razvoj igara i algoritama za donošenje odluka. Igra šaha je izabrana zbog svoje strateške dubine, velikog broja mogućih stanja i pogodnosti za primjenu Monte Carlo metoda, koje se zasnivaju na simulaciji velikog broja slučajnih partija kako bi se statistički procijenilo koji je potez najpovoljniji.

U nastavku rada detaljno će biti opisani programski jezik Python, korištene biblioteke, koncepti paralelizacije i višenitnog izvršavanja, osnove Monte Carlo algoritma, te konkretni koraci u razvoju igre šaha u kojoj računar donosi odluke koristeći paralelne simulacije. Također, biće predstavljena poređenja performansi između sekvencijalnog i paralelnog izvođenja algoritma, s ciljem prikazivanja stvarnih poboljšanja koja paralelizacija donosi u ovakvim aplikacijama.

# Istorijski pregled jezika

Python je objektno orijentisani, interpretirani programski jezik opšte namjene. Njegov autor je holandski programer Guido van Rossum, koji je početkom devedesetih godina započeo razvoj jezika tokom svog slobodnog vremena, s ciljem da napravi moderan i jednostavan jezik privlačan Unix/C programerima. Osnovu za Python predstavljao je jezik ABC, razvijen u istraživačkom institutu CWI (Centrum Wiskunde & Informatica), gdje je Van Rossum tada radio. ABC je bio zamišljen kao edukativni jezik i zamjena za tada popularne jezike poput BASIC-a i Pascala.

Naziv "Python" inspirisan je britanskom humorističkom serijom Monty Python’s Flying Circus, dok je razvojno okruženje IDLE dobilo ime po članu te grupe, Ericu Idleu.

Prva verzija Pythona objavljena je 1994. godine. Danas se istovremeno koriste dvije verzije jezika: Python 2, koji je objavljen 2000. godine i i dalje je prisutan zbog velikog broja postojećih aplikacija, te Python 3, koji je izašao 2008. godine i nije kompatibilan s prethodnom verzijom. Iako se Python 2 više ne razvija aktivno, zbog naslijeđenog softvera još uvijek se koristi u određenim okruženjima, dok se razvoj novih aplikacija preporučuje isključivo u verziji 3. Postoje alati za automatsku konverziju koda iz verzije 2 u verziju 3, ali potpuna kompatibilnost nije zagarantovana.

Važnija izdanja pojedinih verzija jezika Python su:

* Python 1 (1994)
* Python 2 (2000)
* Python 3 (2008)
* Python 3.4 (2014)
* Python 3.5 (2015)
* Python 3.6 (2016)
* Python 3.7 (2018)
* Python 3.8 (2019)
* Python 3.9 (2020)

Jezik Python je relativno jednostavan, ali se oslanja na obimne i kvalitetne programske bibliteke. Standardna programska biblioteka jezika Python (Python Standard Library) obuhvata tipove i strukture podataka, ugrađene funkcije i obradu izuzetaka, koji su uvek raspoloživi, kao i veliki broj međusobno zavisnih modula, koji proširuju mogućnosti jezika nakon što se uključe u program naredbom import.

Pošto je i sam naziv jezika šaljiv, softverski principi koji su izvorno uticali na koncipiranje i realizaciju jezika izloženi su kroz 20 aforizama.

Principi jezika Python :

• Lijepo je bolje nego ružno

• Eksplicitno je bolje nego implicitno

• Jednostavno je bolje nego složeno

• Složeno je bolje nego komplikovano

• Ravno je bolje nego ugnježdeno

• Retko je bolje nego gusto

• Čitljivost je najvažnija

• Posebni slučajevi nisu dovoljno posebni da krše pravila

• Ipak je praktičnost važnija od čistote

• Greške nikad ne treba prihvatiti ćutke

• ... osim ako se eksplicitno ne ućuti

• Kad se pojavi dvosmislenost, ne treba pogađati

• Trebalo bi da postoji jedan - poželjno i samo jedan - očigledan način da se nešto uradi

• ... mada taj način ne mora da bude vidljiv na prvi pogled, osim ako niste Holanđanin

• Sada je bolje nego nikada

• ... mada je nikada često bolje nego upravo sada

• Ako je implementaciju teško objasniti, ideja je loša

• Ako je implementaciju lako objasniti, mora da je ideja dobra

• Prostori imena su sjajna ideja - hajde da ih napravimo još!

# Programski jezik Python

Prema godišnjoj analizi Svjetskog udruženja inženjera elektrotehnike i elektronike (Institute of Electrical and Electronics Engineers – IEEE), Python je trenutno najpopularniji programski jezik. Ova pozicija rezultat je niza prednosti koje Python čine izuzetno privlačnim za programere svih profila. Najvažniji razlozi njegove popularnosti su:

* jednostavna i intuitivna sintaksa koja omogućava pisanje čitljivog i preglednog koda
* snažne programske mogućnosti i širok spektar primjene, od automatizacije i analize podataka do razvoja web i naučnih aplikacija
* koristi se u radu mnogih velikih svjetskih kompanija i organizacija, kao što su YouTube, Google, Yahoo i NASA
* jezik je otvorenog koda (open source), besplatan za korištenje i ima veliku i aktivnu zajednicu korisnika koja pruža podršku

Dalji razvoj jezika i održavanje njegovog ekosistema nadgleda neprofitna organizacija Python Software Foundation, koja je nosilac svih prava intelektualne svojine vezanih za Python. Softver otvorenog koda može se slobodno preuzeti sa zvanične web stranice: www.python.org.

Programi u Pythonu se pišu kao niz naredbi, pri čemu je kraj linije ujedno i kraj naredbe (delimiter). U pisanju složenih naredbi, posebnu ulogu ima uvlačenje (indentacija), koje u Pythonu nije stvar stila već nužan element sintakse, jer definiše blokove koda. Jezik razlikuje velika i mala slova (case sensitive), što znači da su, na primjer, naredbe print i Print različite i ne mogu se koristiti naizmjenično.

U ovom poglavlju će biti ukratko predstavljene osnovne osobine programskog jezika Python, uz prikaz tipičnih primjera njegove upotrebe.

## Komentar

Komentari su proizvoljni dijelovi teksta koje programski prevodilac (interpreter) zanemaruje prilikom izvršavanja koda, ali imaju ključnu ulogu u razumijevanju i održavanju softvera. Oni služe kao objašnjenja ili napomene namijenjene osobama koje čitaju program, te predstavljaju osnovnu formu dokumentacije unutar izvornog koda.

U jeziku Python postoje dvije osnovne vrste komentara:

* **Jednolinijski komentari**

Ovi komentari započinju znakom # i protežu se do kraja linije. **Primjer:**

# Ovo je jednolinijski komentar

* **Višelinijski komentari**

Za unošenje komentara koji obuhvataju više redova, u praksi se koristi sintaksa sa trostrukim navodnicima (bilo jednostruki ''' ili dvostruki """). Iako su tehnički gledano to zapravo višelinijski string literal-i (koji se ne dodjeljuju nijednoj varijabli), Python ih ignoriše ako nisu dio koda, pa se često koriste kao komentari. **Primjer:**

"""

Ovo je primjer

višelinijskog komentara.

"""

## Promjenljive

Promjenljiva (variable) predstavlja imenovanu memorijsku lokaciju u računaru koja sadrži određenu vrijednost i koja se može mijenjati tokom izvršavanja programa. U jeziku Python, nazivi promjenljivih mogu sadržavati:

* mala i velika slova engleske abecede,
* cifre (0–9),
* znak donje crte (\_),

ali naziv ne smije započinjati cifrom. Dozvoljeno je da počinje slovom ili donjom crtom (\_). Prema konvenciji stila jezika Python (PEP 8), preporučuje se da se za imena promjenljivih koriste isključivo mala slova, eventualno razdvojena donjim crtama za čitljivost (npr. broj\_studenata). Iako tehnički nema ograničenja u dužini imena promjenljivih, preporučuje se da nazivi budu dovoljno deskriptivni, ali ne predugi, u skladu s preporukom o maksimalnoj dužini linije koda (tipično 79 karaktera).

Nazivi promjenljivih ne smiju biti isti kao rezervisane riječi (ključne riječi) u Pythonu, jer su one unaprijed definisane za specijalne svrhe unutar jezika. Lista rezervisanih riječi u Pythonu 3 uključuje:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield

Konstanta je vrijednost koja se, u načelu, ne bi trebala mijenjati tokom izvođenja programa. Iako Python ne podržava konstante kao posebnu kategoriju (nema mehanizam za njihovo zaključavanje), ustaljena praksa je da se konstante definišu kao obične promjenljive čije vrijednosti programer dobrovoljno ne mijenja.

Stilska konvencija nalaže da se konstante pišu velikim slovima, uz eventualno korištenje donje crte za razdvajanje riječi. **Primjeri:**

PI = 3.14159

E = 2.71828

BRZINA\_SVJETLOSTI = 299\_792\_458 # u metrima po sekundi

GOOGOL = 10\*\*100

## Dodjela vrijednosti

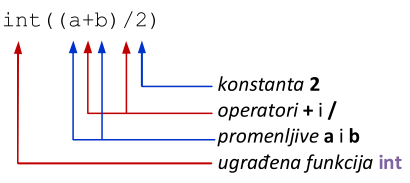
U jeziku Python, naredba za dodjelu vrijednosti ima relativno jednostavnu i učestalu formu:

<promjenljiva> = <izraz>

Tip promjenljive se određuje **automatski**, na osnovu tipa dodijeljene vrijednosti. **Na primjer:**

Python takođe podržava **višestruke dodjele** vrijednosti u jednoj liniji, što je korisno za inicijalizaciju više promjenljivih odjednom:

## Izrazi

Izrazi su pravilna kombinacija vrijednosti (konstanti), promjenljivih, funkcija i operatora, **Na primjer:**

Nakon izračunavanja ili evaluacije izraza dobija se rezultat određenog tipa, koji se može dodijeliiti nekoj promjenljivoj. Rezultat može biti numerički, nenumerički ili logički (koji je u stvari podtip tipa int). U gornjem primeru, za a=1 i b=2, rezultat evaluacije je 1, vrednost tipa int.

### **Evaluacija izraza**

Redoslijed izvršavanja operacija pri izračunavanju izraza u Pythonu zavisi od tipova upotrebljenih vrijednosti i operatora. Kod aritmetičkih izraza, primjenjuju se pravila iz matematike – posebno prioritet operatora i korišćenje zagrada. **Na primjer:**

3 + 4 \* 5 \*\* 2

Rezultat evaluacije ovog izraza je 103, jer se operacije izvršavaju u sljedećem redoslijedu:

* Eksponenciranje: (\*\*): 5 \*\* 2 → 25
* Množenje: (\*): 4 \* 25 → 100
* Sabiranje: (+): 3 + 100 → 103

Međutim, ako se koriste zagrade:

(3 + 4) \* 5 \*\* 2

Rezultat je `175`, jer se prvo izvršava sabiranje unutar zagrada, pa zatim ostale operacije:

* 3 + 4 → 7
* 5 \*\* 2 → 25
* 7 \* 25 → 175

Kod logičkih izraza, operator and ima viši prioritet od operatora or. To znači da se and evaluira prije or, osim ako se eksplicitno ne promijeni redoslijed zagradama. **Primjeri:**

True or False and False # rezultat: True

False and False → False

True or False → True

False or True and False # rezultat: False

True and False → False

False or False → False

Svi operatori istog prioriteta evaluiraju se s leva u desno. Zagrade menjaju redoslijed evaluacije svih tipova operatora.

### **Operatori**

Operatori u jeziku Python definisani su za određeni tip vrijednosti na koje se mogu primjeniti. Razlikuju se aritmetički, operatori poređenja, specijalni operatori (koji se koriste uz operator dodjele vrijednosti), logički operatori, operatori za rad s bitovima, operatori pripadnosti i operatori identifikacije. U odnosu na broj argumenata operatori mogu biti unarni (jedan argument) i binarni (dva argumenta).

U ovom poglavlju će se kratko nabrojati aritmetički i relacioni operatori. Aritmetički operatori su

+ → sabiranje numeričkih vrednosti i unarni operator predznaka,

- → oduzimanje numeričkih vrednosti i unarni operator predznaka,

\* → množenje numeričkih vrednosti,

/ → dijeljenje numeričkih vrednosti koje proizvodi decimalni rezultat,

// → cjelobrojno dijeljenje kojim se decimalni dio rezultata odbacuje,

\*\* → stepenovanje ili eksponenciranje,

% → ostatak od dijeljenja ili modul.

Binarni operator modul kao rezultat daje ostatak od dijeljenja vrijednosti navedene ispred operatora vrijednošću navedenom iza njega. Treba napomenuti da je ostatak od dijeljenja parnog broja brojem 2 jednak je nuli. Ovaj operator je primjer jednostavne jednosmjerne funkcije, za koju ne postoji inverzna funkcija. Takve funkcije su pogodne za šifrovanje podataka, jer se bez podatka o djeliocu, ne može znati koji broj je bio djeljenik.

Relacioni ili operatori poređenja u izrazima su:

< → manje od

<= → manje ili jednako

> → veće od

>= → veće ili jednako

== → jednako

!= → različito (nije jednako)

## Osnovne strukture podataka

U jeziku Python osnovne strukture podataka uključuju stringove, liste, n-torke i rječnike. String predstavlja niz znakova koji su organizovani tako da im se može pristupiti pomoću pozicija, odnosno indeksa, pri čemu brojanje počinje od nule. Stringovi su nepromjenljivi, što znači da se njihov sadržaj ne može mijenjati nakon što su kreirani.

Liste su uređene kolekcije elemenata koje mogu biti različitih tipova i koje se mogu mijenjati. Svakom elementu u listi može se pristupiti pomoću njegovog indeksa. Python omogućava rad sa ugnježdenim listama, tj. listama koje kao elemente sadrže druge liste, čime se mogu modelovati složenije strukture podataka poput hijerarhija.

N-torke su slične listama po načinu organizacije elemenata, ali za razliku od lista, one su nepromjenljive. Koriste se kada je potrebno zaštititi podatke od izmjene ili kada se podaci koriste kao ključevi u drugim strukturama zbog zahtjeva za nemijenjivošću.

Rječnici predstavljaju strukturirane podatke u obliku parova ključ:vrijednost. Ključevi moraju biti jedinstveni i nemijenjivi tipovi podataka, dok vrijednosti mogu biti bilo kog tipa i mogu se mijenjati. Rječnici omogućavaju brz pristup podacima na osnovu ključa, što ih čini pogodnim za organizovanje i pretraživanje većih količina podataka.

## Grananje

Grananje u jeziku Python predstavlja način da se u programu donose odluke na osnovu ispunjenosti određenih uslova. Osnovna struktura koristi ključnu riječ if, a po potrebi i elif (else if) i else. Kada se napiše if naredba, ona provjerava da li je uslov tačan (logički istinit). Ako jeste, izvršava se blok naredbi ispod nje, koji mora biti uvučen. Ako uslov nije ispunjen, prelazi se na elif (ako postoji), a ako nijedan uslov nije ispunjen, izvršava se else blok.

Primjer jednostavnog grananja izgleda ovako:

lijepo\_vrijeme = False

if lijepo\_vrijeme:

print("Nije vam potreban kišobran.")

U ovom primjeru se neće ispisati ništa jer je lijepo\_vrijeme postavljeno na False, pa se print naredba ne izvršava.

Kada želimo da obradimo dvije moguće situacije, koristimo if zajedno sa else, kao u sljedećem primjeru:

vrijeme = "kiša"

if vrijeme == "sunčano":

print("Ponesi naočare za sunce.")

else:

print("Ponesi kišobran.")

Ako je vrijednost promjenljive vrijeme jednaka "kiša", izvršiće se naredba iz else bloka.

Za situacije sa više uslova, koristi se i elif, čime se omogućava grananje u više pravaca:

starost = 70

if starost < 18:

print("Popust za djecu i učenike.")

elif starost > 65:

print("Nije potrebna karta.")

else:

print("Puna cijena karte.")

Ovdje će se ispisati "Nije potrebna karta.", jer je uslov starost > 65 tačan, dok se ostali preskaču.

U Pythonu se svaki uslov tretira kao tačan osim ako je eksplicitno False, 0, "" (prazan string), [] (prazna lista), None, ili neki drugi "prazan" ili "nulti" podatak. Ključno je da svaki blok koda koji pripada if, elif ili else mora biti uvučen – najčešće četiri razmaka – jer Python koristi uvlake za označavanje blokova, a ne zagrade kao neki drugi jezici.

## Iteracija

U programskom jeziku Python, ponavljanje određenih dijelova koda omogućeno je korištenjem naredbi za ponavljanje, poznatih i kao petlje. Postoje dvije osnovne vrste petlji: for i while.

* **for petlja** (kada je broj ponavljanja poznat)

For petlja se koristi kada unaprijed znamo koliko puta treba da se neki blok naredbi izvrši. Najčešće se koristi za prolazak kroz elemente nekog niza, kao što su liste, stringovi ili rasponi brojeva.

**Sintaksa:**

for varijabla in sekvenca:

blok\_naredbi

**Primjer:**

for broj in [1, 3, 7]:

print("Indeks =", broj)

U ovom primjeru, for petlja prolazi kroz listu [1, 3, 7], i u svakom prolazu ispisuje vrijednost varijable broj. Petlja će se izvršiti tačno tri puta. Takođe možemo koristiti funkciju range() da generišemo niz brojeva:

for i in range(5):

print("Broj =", i)

* **while petlja** (kada broj ponavljanja nije poznat)

While petlja se koristi kada nije poznato koliko puta treba da se neki blok izvrši, već se izvršavanje ponavlja dok je zadani logički uslov istinit.

**Sintaksa:**

while uslov:

blok\_naredbi

**Primjer:**

broj = 16

while broj > 0:

print("Broj =", broj)

broj = broj - 4

Ova while petlja će se izvršavati dok je vrijednost promjenljive broj veća od nule. U svakom prolazu, vrijednost se smanjuje za 4. Kada broj postane 0, uslov broj > 0 više nije zadovoljen i petlja se prekida.

## Funkcije

### **Ugradjene Funkcije**

Python ima veliki broj ugrađenih funkcija za različite primjene, koje su uvek na raspolaganju:

abs(), all(), any(), ascii(), bin(), bool(), breakpoint(), bytearray(), bytes(), callable(), chr(), classmethod(), compile(), complex(), delattr(), dict(), dir(), divmod(), enumerate(), eval(), exec(), filter(), float(), format(), frozenset(), getattr(), globals(), hasattr(), hash(), help(), hex(), id(), input(), int(), isinstance(), issubclass(), iter(), len(), list(), locals(), map(), max(), memoryview(), min(), next(), object(), oct(), open(), ord(), pow(), print(), property(), range(), repr(), reversed(), round(), set(), setattr(), slice(), sorted(), staticmethod(), str(), sum(), super(), tuple(), type(), vars(), zip(), **import**()

MOZDA JOS NESTO DODATI O NJIMA NE ZNAM

### **Korisnicke Funkcije**

U Pythonu, korisničke funkcije su imenovane cjeline koda koje izvršavaju određeni zadatak i mogu se pozivati više puta unutar programa. Funkcije doprinose preglednosti, organizaciji i smanjenju dužine koda. One se definišu pomoću ključne riječi def, a po potrebi mogu imati parametre, kao i povratne vrijednosti.

Osnovna sintaksa funkcije izgleda ovako:

def ime\_funkcije(parametri):

blok\_naredbi

return vrijednost

Na primjer, funkcija koja sabira dva broja može se definisati na sljedeći način:

def zbir(i1, i2):

rezultat = i1 + i2

return rezultat

U ovom slučaju, zbir je ime funkcije, a i1 i i2 su formalni parametri. Funkcija vraća rezultat sabiranja pomoću naredbe return. Nakon što se funkcija definiše, može se više puta koristiti u programu.

Funkcije ne moraju uvijek vraćati rezultat. Postoje i funkcije koje obavljaju određenu radnju, ali ne vraćaju vrijednost. U tom slučaju, one koriste naredbu return bez ikakve vrijednosti, ili izostave return u potpunosti. Na primjer:

def pozdrav(ime):

print(f"Zdravo, {ime}!")

Ova funkcija ispisuje poruku pozdrava, ali ne vraća nikakav rezultat.

Korišćenje funkcija omogućava jednostavno testiranje i ponovno korišćenje logike unutar različitih dijelova programa. Parametri omogućavaju funkcijama da budu fleksibilne i prilagodljive, dok naredba return omogućava da rezultat bude dalje upotrebljen u programu.

## Interakcija korisnika

U programskom jeziku Python, naredbe za ulaz i izlaz omogućavaju interakciju korisnika sa programom – unos podataka sa tastature, ispis rezultata na ekran, kao i formatirano prikazivanje podataka. Osnovne naredbe za interaktivni ulaz i izlaz su ugrađene funkcije input() i print().

Funkcija input() omogućava da program postavi pitanje korisniku i primi njegov odgovor. Kao rezultat izvršavanja, ova funkcija uvijek vraća vrijednost tipa string, bez obzira na to da li je korisnik uneo broj ili tekst. **Na primjer:**

ime = input("Kako se zovete? ")

Ako korisnik unese "Jovana", vrijednost promjenljive ime će biti string 'Jovana'. Slično tome:

broj = input("Unesite broj: ")

Čak i ako korisnik unese 12, vrijednost promjenljive broj će biti string '12'. Da bi se ova vrijednost koristila kao broj, neophodno je izvršiti konverziju tipa.

S druge strane, funkcija print() se koristi za prikaz rezultata na ekranu. Ona ne vraća nikakvu korisnu vrijednost, već None. Na primjer:

ime = "Jovana"

print("Korisnik je", ime)

Python omogućava i precizniju kontrolu nad načinom ispisa podataka pomoću funkcije format(). Ova funkcija omogućava formatiranje stringova, cijelih i decimalnih brojeva, uz zadavanje dužine ispisa, broja decimala, poravnanja i slično. **Na primjer:**

print(format("tekst", "10s")) # ispis stringa širine 10 karaktera

print(format(3, "10d")) # ispis cijelog broja u polju širine 10

print(format(3.14, "10.2f")) # ispis decimalnog broja s 2 decimale

Rezultati će biti desno poravnati u zadatom prostoru, što je podrazumijevano ponašanje za brojeve.

Takođe, moguće je eksplicitno definisati poravnanje:

* < za lijevo poravnanje
* > za desno poravnanje

Na primjer:

print(format("tekst", ">10s")) # desno poravnanje stringa

print(format(3, "<10d")) # lijevo poravnanje cijelog broja

print(format(3.14, "<10.2f")) # lijevo poravnanje decimalnog broja

Ove mogućnosti formatiranja posebno su korisne kada se rezultati ispisa prikazuju u tabelama ili kada se generišu uredno strukturirani izvještaji.

# Objektno Orjentisano programiranje

Objektno orijentisano programiranje (OOP) predstavlja jedan od najvažnijih pristupa u savremenom razvoju softvera. Ono se temelji na konceptima koji omogućavaju modeliranje stvarnog sveta putem objekata – entiteta koji kombinuju podatke i funkcionalnost u jedinstvene cjeline. Na taj način, OOP nudi prirodniji i modularniji način razmišljanja i dizajniranja softverskih sistema, posebno kada su u pitanju kompleksni projekti.

Zašto OOP? Objektno orijentisani pristup rješava mnoge probleme koji su svojstveni proceduralnom programiranju, gdje su podaci i funkcije (metode) strogo odvojeni. U proceduralnim jezicima, poput C-a ili Pascala, podaci se često šalju funkcijama koje ih obrađuju, što otežava organizaciju i ponovnu upotrebu koda.

S druge strane, OOP omogućava da podaci i operacije koje ih koriste budu spakovani zajedno u objekte. Svaki objekat je instanca određene klase i poseduje sopstvene atribute (podatke) i metode (funkcije). Na taj način, objekti mogu međusobno sarađivati i formirati funkcionalne cjeline unutar aplikacije.

U jeziku Python, objektno orijentisano programiranje je prirodni deo jezika. Iako Python podržava i proceduralni stil programiranja, njegov OOP model je moćan i fleksibilan. Programi se u ovom kontekstu posmatraju kao kolekcije objekata koji međusobno komuniciraju i razmenjuju informacije. Pisanje programa u Pythonu na objektno orijentisan način podrazumeva razmišljanje u terminima klasa, instanci, nasleđivanja, enkapsulacije i polimorfizma — što su ključni principi OOP-a.

## Klasa i objekat

U objektno orijentisanom programiranju (OOP), klasa i objekt su osnovni pojmovi. Klasa se može posmatrati kao šablon ili plan prema kojem se kreiraju objekti. Ona definiše koje osobine (atribute) i ponašanja (metode) objekti imaju. Objekt je konkretna instanca klase – stvarni entitet koji ima određeno stanje i ponašanje.

U jeziku Python, sve su vrijednosti objekti. Brojevi, stringovi, liste – svi su objekti svojih odgovarajućih klasa. Kada se napravi nova vrijednost, Python automatski dodjeljuje jedinstvenu identifikaciju tom objektu, dok se njegova vrsta određuje klasom kojoj pripada. Na primjer, broj 3 je objekt klase int, dok je tekstualna vrijednost "Zdravo" objekt klase str. Pomoću ugrađenih funkcija id() i type() možemo ispitati identitet i tip objekta.

Objekti u Pythonu imaju metode, tj. funkcije koje su specifične za njihovu klasu. Na primjer, string objekti imaju metode lower(), upper(), strip() i mnoge druge koje omogućavaju rad s tekstom. Sintaksa <objekt>.<metod>() omogućava jednostavno pozivanje ovih metoda, a svaki objekt zna koje metode može da koristi, jer su one definisane unutar njegove klase.

Svaki objekt ima tri ključne osobine: identitet, stanje i ponašanje. Identitet se odnosi na jedinstvenu oznaku objekta u memoriji. Stanje predstavlja vrijednosti koje objekt sadrži, dok ponašanje obuhvata radnje koje objekt može izvršiti, odnosno metode koje su mu dostupne.

Klase omogućavaju apstrakciju, enkapsulaciju, nasljeđivanje i polimorfizam. Apstrakcija znači da korisnik klase ne mora znati kako su metode implementirane, već samo kako se koriste. Enkapsulacija štiti unutrašnje podatke objekta tako što ih skriva od vanjskog svijeta, izlažući samo ono što je neophodno. Nasljeđivanje omogućava kreiranje novih klasa koje nasljeđuju osobine i ponašanja iz postojećih klasa, čime se smanjuje potreba za ponavljanjem koda. Polimorfizam omogućava da različiti objekti odgovaraju na iste metode na svoj način, što dodatno povećava fleksibilnost programa.

U praksi, korišćenje klasa i objekata dovodi do modularnijeg, preglednijeg i lakše održivog koda. Softver napisan objektno orijentisanim pristupom lakše se razvija, proširuje i koristi u različitim kontekstima, što ga čini dominantnim paradigmom u savremenom programiranju.

## Kreiranje objekta

## Definisanje korisnicke klase

## Kreiranje objekta pomocu konstruktora

## Pristup clanovima objekta

## Parametar self

# Moduli u programskom jeziku Python

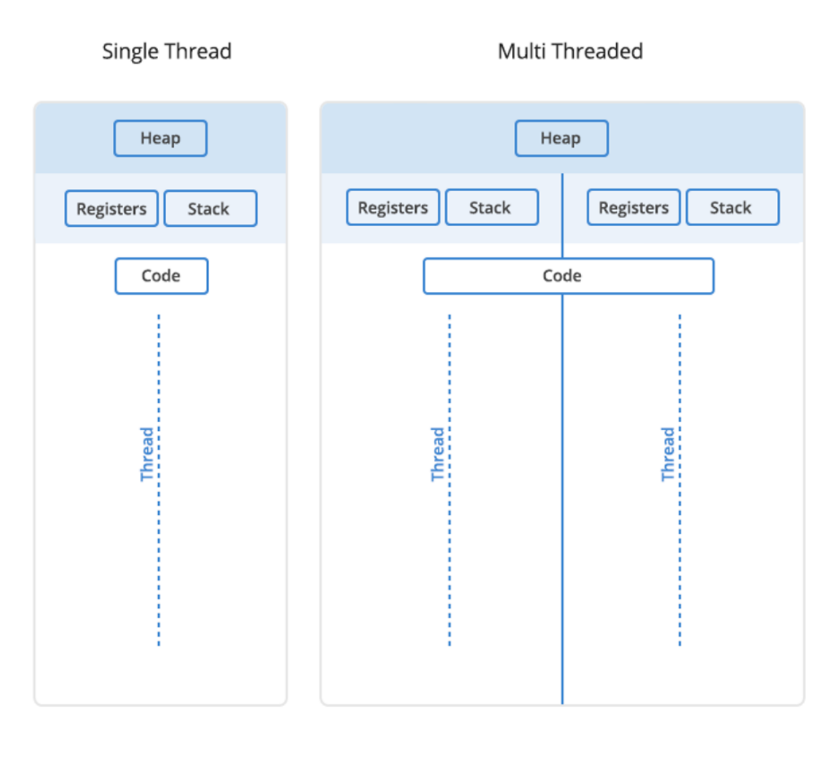
# PyGame

Pygame je open-source biblioteka namijenjena razvoju multimedijalnih aplikacija, s posebnim naglaskom na razvoj 2D videoigara u programskom jeziku Python. Izgrađena na osnovu biblioteke Simple DirectMedia Layer (SDL), Pygame pruža visokonivojski interfejs za upravljanje grafičkim prikazom, obradom događaja, reprodukcijom zvuka i unosom korisnika, čime apstrahuje složenost nižeg nivoa grafičkog programiranja. Modularna arhitektura i podrška za više platformi čine je pogodnom za korištenje u obrazovnim okruženjima, kao i u nezavisnom razvoju softvera. Jedna od glavnih prednosti biblioteke Pygame je njena jednostavnost i pristupačnost, što omogućava brzo prototipiranje i razvoj interaktivnih aplikacija bez potrebe za dubokim razumijevanjem niskonivojskog grafičkog koda ili hardverske akceleracije. U naučnim i akademskim projektima, Pygame se često koristi za vizualizaciju algoritama, testiranje korisničkog interfejsa i razvoj simulacionih okruženja, posebno u oblastima poput vještačke inteligencije, robotike i interakcije čovjek-računar. Njena opširna dokumentacija i aktivna zajednica dodatno doprinose njenoj primjenjivosti kao edukativnog i prototipskog alata.

# Paralelizacija u jeziku Python

U računarskim sistemima, proces predstavlja instancu računarskog programa koji je trenutno u fazi izvršavanja. Svaki proces se sastoji od tri osnovne komponente: izvršnog programa, pridruženih podataka koji su programu potrebni za rad (kao što su varijable, radni prostor i baferi), te konteksta izvršavanja koji opisuje trenutno stanje procesa. Unutar procesa može postojati više niti (engl. threads), koje predstavljaju najmanju jedinicu obrade koju operativni sistem može samostalno zakazati za izvršenje. Nit se može posmatrati kao niz instrukcija unutar programa koji se može izvršavati nezavisno od ostatka koda, omogućavajući paralelizam i efikasnije iskorištavanje resursa.

Svaka nit posjeduje vlastiti blok kontrole niti (Thread Control Block – TCB), u kojem se čuvaju ključne informacije o njenom stanju i izvršavanju. TCB sadrži jedinstveni identifikator niti (TID), pokazivač na stek niti koji sadrži lokalne varijable dostupne samo toj niti, programski brojač koji ukazuje na adresu instrukcije koja se trenutno izvršava, te stanje niti koje može biti jedno od sljedećih: aktivno, spremno, čeka, pokreće se ili završeno. Osim toga, tu su i registri koji se koriste tokom obrade, kao i pokazivač na roditeljski proces, odnosno na kontrolni blok procesa (PCB) unutar kojeg nit postoji. Zahvaljujući ovom modelu, niti omogućavaju lakšu implementaciju konkurentnog programiranja, pri čemu se različiti dijelovi programa mogu izvršavati istovremeno, uz dijeljenje istog adresnog prostora i resursa procesa.

Više niti može postojati unutar jednog procesa, pri čemu svaka nit ima svoj vlastiti skup registara i lokalne varijable koje su pohranjene u njenom steku. Nasuprot tome, sve niti unutar istog procesa dijele zajedničke resurse kao što su globalne varijable (smještene u memorijskoj hrpi) i zajednički programski kod. Ovakva arhitektura omogućava efikasnu razmjenu podataka i koordinaciju među nitima, bez potrebe za složenim mehanizmima međuprocesne komunikacije. Paralelizam se u ovom kontekstu definiše kao sposobnost procesora ili višejezgarnih sistema da istovremeno izvršavaju više niti, čime se postiže veća brzina obrade i bolja iskorištenost sistemskih resursa.

Više niti unutar jendog procesa

## Kreiranje niti

Za kreiranje nove niti u Pythonu, može se koristiti metoda dostupna u starijem thread modulu:

thread.start\_new\_thread(function, args[, kwargs])

Ova metoda omogućava jednostavan i efikasan način za pokretanje novih niti na operativnim sistemima kao što su Linux i Windows. Kada se pozove, funkcija se odmah vraća, a nova nit počinje s izvršavanjem specificirane funkcije, kojoj se prosljeđuju argumenti iz args. Kada funkcija završi svoje izvršavanje, nit automatski prestaje sa radom.

* args: Tuple koji sadrži argumente koji se prosljeđuju funkciji.
* kwargs: (opciono) Rječnik sa argumentima ključ-vrijednost, ako je potrebno.

Za pozivanje funkcije bez argumenata koristi se prazan tuple ().

**Primjer**: Računanje kvadrata i kubova brojeva pomoću niti:

import thread

import time

# Funkcija za računanje kvadrata brojeva

def kvadrati(brojevi):

for broj in brojevi:

print(f"Kvadrat broja {broj} je {broj \*\* 2}")

time.sleep(1)

# Funkcija za računanje kubova brojeva

def kubovi(brojevi):

for broj in brojevi:

print(f"Kub broja {broj} je {broj \*\* 3}")

time.sleep(1)

# Pokretanje niti

try:

thread.start\_new\_thread(kvadrati, ([2, 3, 4, 5],))

thread.start\_new\_thread(kubovi, ([2, 3, 4, 5],))

except:

print("Greška: nije moguće pokrenuti niti")

# Glavna nit ostaje aktivna kako bi omogućila završetak ostalih niti

while True:

pass

## Kreiranje niti pomocu threading modula

Modul threading u Pythonu pruža snažan i fleksibilan način za implementaciju višestrukih niti unutar jednog programa. Za razliku od starijeg i ograničenijeg thread modula, threading nudi dodatne funkcije i klasu Thread koja omogućava lakše kreiranje i upravljanje nitima. Osnovni način korištenja ovog modula jeste definisanje podklase Thread u kojoj se implementira metoda run(), što predstavlja kod koji će nit izvršavati. Metoda start() pokreće nit i automatski poziva run() u posebnoj izvršnoj niti.

U nastavku je primjer gdje kreiramo dvije niti koje paralelno izvršavaju funkciju za ispis trenutnog vremena, ali sa različitim intervalima čekanja. Ova demonstracija jasno pokazuje kako se dvije niti izvode istovremeno, dok glavna nit nastavlja svoj tok bez čekanja na njihovo završavanje.

import threading

import time

# Funkcija koju niti izvršavaju

def ispis\_vremena(ime\_niti, delay, brojac):

while brojac:

time.sleep(delay)

print(f"{ime\_niti}: {time.ctime(time.time())}")

brojac -= 1

# Definicija podklase Thread

class MojaNit(threading.Thread):

def \_\_init\_\_(self, nitID, ime, brojac, delay):

threading.Thread.\_\_init\_\_(self)

self.nitID = nitID

self.ime = ime

self.brojac = brojac

self.delay = delay

def run(self):

print(f"Pokreće se {self.ime}")

ispis\_vremena(self.ime, self.delay, self.brojac)

print(f"Zatvara se {self.ime}")

# Kreiranje i pokretanje niti

nit1 = MojaNit(1, "Nit-1", 5, 1)

nit2 = MojaNit(2, "Nit-2", 3, 2)

nit1.start()

nit2.start()

print("Zatvara se glavna nit")

U ovom primjeru, Nit-1 ispisuje vrijeme pet puta sa pauzom od jedne sekunde, dok Nit-2 to radi tri puta sa pauzom od dvije sekunde. Primijetit ćete da se ispis poruka prepliće, jer se niti izvršavaju paralelno, nezavisno jedna od druge. Takođe, glavna nit odmah nastavlja sa izvršavanjem, što se vidi u poruci "Zatvara se glavna nit", i ne čeka na završetak ovih dvije niti.

Pored jednostavnog pokretanja niti, threading modul nudi i dodatne metode kao što su join(), koja omogućava glavnoj niti da sačeka završetak neke druge niti, te is\_alive() koja provjerava da li je određena nit još uvijek aktivna. Ove metode su korisne u situacijama kada je potrebno sinhronizovati izvršavanje niti i osigurati da određeni zadaci budu kompletirani prije nastavka programa.

Upotreba threading modula je ključna u razvoju aplikacija koje zahtijevaju konkurentnost ili paralelizam, kao što su serveri, interaktivni programi ili aplikacije za obradu velikih podataka. Ovakav pristup značajno poboljšava iskorištenost procesorskih resursa i responzivnost programa, naročito na modernim višejedarnim sistemima.

## Sinhronizacija niti

Modul threading koji dolazi sa Pythonom sadrži jednostavan, ali efikasan mehanizam za zaključavanje (engl. lock), koji omogućava sinhronizaciju niti. Zaključavanje se kreira pozivom funkcije Lock(), koja vraća novi objekat zaključavanja. Da bi se spriječilo istovremeno izvršavanje kritičnog dijela koda od strane više niti, koristi se metoda acquire(). Ova metoda može imati opcioni parametar blocking koji određuje ponašanje niti prilikom pokušaja preuzimanja zaključavanja. Kada je blocking postavljen na 0 (False), nit odmah vraća False ako zaključavanje nije dostupno, a True ako jeste. Kada je blocking 1 (True) ili nije naveden, nit čeka dok ne dobije zaključavanje.

Nakon što nit završi rad u kritičnoj sekciji, poziva se metoda release() koja oslobađa zaključavanje, omogućavajući drugim nitima da preuzmu kontrolu. Ovaj mehanizam je ključan za izbjegavanje problema poput uslova trke, gdje bi istovremeni pristup zajedničkim resursima mogao dovesti do nepredvidivih rezultata.

import threading

import time

# Kreiranje globalnog zaključavanja

threadLock = threading.Lock()

def ispis\_vremena(ime\_niti, brojac, delay):

while brojac:

time.sleep(delay)

print(f"{ime\_niti}: {time.ctime(time.time())}")

brojac -= 1

class MojaNit(threading.Thread):

def \_\_init\_\_(self, nitID, ime, brojac):

threading.Thread.\_\_init\_\_(self)

self.nitID = nitID

self.ime = ime

self.brojac = brojac

def run(self):

print(f"Pokreće se {self.ime}")

# Preuzimanje zaključavanja prije ulaska u kritičnu sekciju

threadLock.acquire()

ispis\_vremena(self.ime, self.brojac, 3)

# Oslobađanje zaključavanja nakon završetka kritične sekcije

threadLock.release()

print(f"Zatvara se {self.ime}")

# Kreiranje i pokretanje niti

nit1 = MojaNit(1, "Nit-1", 3)

nit2 = MojaNit(2, "Nit-2", 3)

nit1.start()

nit2.start()

# Čekanje da niti završe

nit1.join()

nit2.join()

print("Zatvara se glavna nit")

**Mogući rezultat izvršavanja:**

Pokreće se Nit-1

Pokreće se Nit-2

Nit-1: Thu May 28 09:11:28 2022

Nit-1: Thu May 28 09:11:31 2022

Nit-1: Thu May 28 09:11:34 2022

Nit-2: Thu May 28 09:11:37 2022

Nit-2: Thu May 28 09:11:40 2022

Nit-2: Thu May 28 09:11:43 2022

Zatvara se Nit-1

Zatvara se Nit-2

Zatvara se glavna nit

U ovom primjeru, zaključavanje osigurava da samo jedna nit u datom trenutku izvršava funkciju ispis\_vremena, čime se sprječava preklapanje ispisa i druge potencijalne greške uslijed istovremenog pristupa zajedničkim resursima. Metoda join() se koristi da glavna nit sačeka završetak obje pokrenute niti prije nego što se zatvori program. Ovakav pristup omogućava sigurnu i kontrolisanu višestruku paralelnu obradu unutar Pythona.

# Sah

## Kreiranje table

Prije nego što se započne sa samim iscrtavanjem šahovske table i figura, potrebno je inicijalizovati osnovne parametre i kreirati prozor igrice koji će služiti kao glavni prikaz igre. U ovom koraku definišu se dimenzije prozora (WIDTH i HEIGHT), koje određuju ukupnu veličinu prikaza – u ovom slučaju kvadratnog oblika dimenzija 1024x1024 piksela. Šahovska tabla je klasičnog formata 8x8 polja, što je definisano konstantom DIMENSION. Na osnovu visine prozora i broja redova/kolona, izračunava se veličina svakog pojedinačnog polja (SQ\_SIZE), čime se osigurava da tabla zauzima čitav prozor bez deformacija. Dodatno, postavlja se maksimalni broj frejmova po sekundi (MAX\_FPS), što kontroliše brzinu osvježavanja ekrana i omogućava glatko iscrtavanje. Na kraju, kreira se prazan rječnik IMAGES, u koji će se kasnije učitavati slike svih figura kako bi se one mogle efikasno prikazivati na tabli. Ove pripreme predstavljaju temelj za pravilno funkcionisanje vizualnog dijela igre.

Slike primjer dimenzija

U prvoj fazi implementacije šahovske igre fokus je na vizuelnom prikazu – iscrtavanju šahovske table i figura. Ova komponenta je ključna za korisnički interfejs jer omogućava igraču da vizualno prati stanje igre. Centralna funkcija u ovom dijelu je drawGameState, koja objedinjuje pozive za iscrtavanje table (drawBoar), figura (drawPieces), označenog polja (highlightSquare) i trenutnog statusa kralja (highlightKing). Funkcija `drawBoard` koristi jednostavnu logiku za iscrtavanje standardne šahovske table sa naizmjenično obojenim poljima – bijelom i smeđom bojom – koja se biraju prema paritetu koordinata svakog polja. S druge strane, drawPieces prolazi kroz dvodimenzionalnu listu koja predstavlja stanje table i, gdje god se nađe figura (npr. "white pawn" ili "black queen"), ona se iscrta pomoću unaprijed učitanih slika figura (IMAGE). Ovaj dio koda osigurava osnovnu vizualnu funkcionalnost igre i omogućava dalje interaktivne nadogradnje poput odabira figura, poteza i vizualnog označavanja specijalnih stanja poput šaha i mata.

Slike primjer draw board

U ovom trenutku, korisnik je u stanju da ispred sebe vidi sahovsku tablu sa svim figurama na njoj. Medjutim, trenutno, nije u stanju Nista sa time da uradi. Iz tog razloga se trenutacno moramo udaljiti od nase „Main” implementacije i zapoceti rad na srcu sahovske logike, u ovom slucaju, to je Chess Engine

## Chess Engine

Fajl chessEngine.py predstavlja srce logike šahovske igre. U njemu se nalazi klasa GameState, koja čuva cjelokupno stanje igre, uključujući raspored figura na tabli, čiji je red za potez, poziciju kraljeva, informaciju o šahu, matu i patu, kao i mogućnosti specijalnih poteza poput en passant hvatanja i rokade. Ovaj fajl je zadužen za generisanje svih pravno dozvoljenih poteza u datom trenutku, kao i za provjeru uslova kao što su pinovanje figura, prijetnje kralju, dozvoljeni potezi tokom šaha i drugo.

Osim osnovne manipulacije stanjem igre, chessEngine.py sadrži i niz pomoćnih funkcija za generisanje poteza specifičnih figura: pješaka, topa, lovca, konja, kraljice i kralja. Poseban fokus posvećen je implementaciji šahovskih pravila – uključujući kompleksne situacije poput šaha, piona pod pinom, kao i uslove za legalnu rokadu.

Na kraju, ovaj fajl također omogućava "undo" poteza, što omogućava korisniku (ili algoritmu) da se vrati na prethodno stanje igre. Zahvaljujući tome, chessEngine.py je ujedno i ključna komponenta u razvoju AI algoritama, jer omogućava simulaciju velikog broja poteza unaprijed

U nastavku cemo se osvrnuti na glavne klase ovog fajla.

## GameState

Klasa GameState predstavlja osnovni model šahovske partije i središnji je element koji drži informacije o trenutnom stanju igre. Ona ne sadrži vizuelni prikaz niti direktnu interakciju sa korisnikom, već isključivo podatke relevantne za tok partije i pravila igre. Unutar nje se nalaze sve ključne varijable koje opisuju pozicije figura na tabli, informacije o tome koji igrač je na potezu, kao i evidencija svih do sada odigranih poteza. Ova klasa funkcioniše kao temeljna logička komponenta koja omogućava generisanje validnih poteza na osnovu trenutnog stanja table, primjenu pravila šaha (poput šaha, mata, en passant hvatanja i rokade), te detekciju složenijih situacija kao što su pinovi i prijetnje kralju. GameState u tom smislu služi kao most između grafičkog prikaza i same mehanike igre te omogućavajući svim ostalim dijelovima programa da pristupe dosljednoj i ažuriranoj reprezentaciji šahovske partije.

### Konstruktor

### **Zakoni kretanja figura**

Zakonitosti kretanja figura predstavljaju temeljni aspekt svakog šahovskog programa jer određuju koje poteze svaka figura smije napraviti na tabli. Bez pravilne implementacije ovih pravila, simulacija igre ne bi bila vjerodostojna, a AI ne bi mogao pravilno procjenjivati stanje igre niti donositi smislene odluke. U našem projektu, za svaku figuru postoji zasebna funkcija koja generiše sve moguće legalne poteze u skladu s pravilima šaha.

Svaka od ovih funkcija ima zadatak da na osnovu trenutne pozicije figure i stanja table izračuna sva mjesta na koja figura može da se pomjeri. Pri tome se poštuju osnovna pravila kretanja, kao što su pravolinijsko kretanje topa i kraljice, dijagonalno kretanje lovca, karakteristični “L” potezi skakača i poseban način kretanja pješaka. Važan aspekt implementacije je i provjera da li je figura “pinovana” (vezana) tj. da li je kretanje ograničeno kako se ne bi otkrio kralj u šahu. Ova provjera se ugrađuje u sve funkcije za generisanje poteza i omogućava da se odabrani potezi filtriraju u skladu s tim.

Kod pješaka posebno se implementiraju pravila kretanja koja uključuju pomjeranje za jedno ili dva polja unaprijed, kao i specifične situacije jedenja dijagonalno i pravilo en passant. Ovo je realizovano tako što funkcija najprije provjerava stanje polja ispred pješaka, a zatim i dijagonalnih polja za eventualno jedenje protivničkih figura ili aktiviranje en passant poteza. Funkcija uzima u obzir i boju igrača, jer pješaci bijelih i crnih figura idu u suprotnim smjerovima.

Top se može kretati po četiri pravca — gore, dolje, lijevo i desno — dokle god su polja slobodna. U kodu je to realizovano iterativnim prolaskom kroz polja u svakom pravcu do trenutka kad naiđe na prepreku ili granicu table. Ako je prepreka protivnička figura, potez se dodaje, ali se dalji potezi u tom pravcu prekidaju. Ako je prepreka saveznik, kretanje u tom smjeru se odmah zaustavlja. Ovakav način osigurava da se svi mogući potezi topa generišu efikasno i tačno.

Skakač je specifičan jer se kreće u obliku slova L i može preskakati druge figure. U implementaciji su navedeni svi mogući skokovi koje konj može napraviti, a zatim se filtriraju oni koji su unutar table i nisu zauzeti savezničkim figurama. Budući da skakač ne može biti pinovan zbog svoje prirode kretanja, ova provjera u funkciji je minimalna i služi kao sigurnosna mjera.

Lovac se kreće dijagonalno i njegova funkcija radi slično kao i funkcija za topa, s tim što se kreće po četiri dijagonalna pravca. Ponovo se iterativno provjeravaju polja u svakom smjeru dok se ne naiđe na prepreku. Ukoliko je figura pinovana, potezi su dodatno ograničeni na pravac pina kako ne bi došlo do šaha. Tako se efikasno osigurava validnost generisanih poteza.

Kraljica je kombinacija topa i lovca, što znači da njena funkcija koristi već postojeće funkcije za topa i lovca da bi generisala sve legalne poteze. Na ovaj način se postiže modularnost i ponovna upotreba koda, što pojednostavljuje održavanje i dalje nadogradnje sistema.

Konačno, kraljeva funkcija generiše poteze na susjedna polja, vodeći računa da kralj ne smije ući u polje koje je pod napadom protivnika. U kodu se to realizuje privremenim pomjeranjem kralja i provjerom da li je time doveden u šah. Ova provjera je ključna za validaciju legalnosti kraljevih poteza i omogućava sigurnu igru bez kršenja pravila šaha.

Sve ove funkcije zajedno čine srž modula za generisanje legalnih poteza, a njihova precizna implementacija omogućava dalji rad na složenijim dijelovima poput provjere šaha, mat pozicija i inteligentnih algoritama za izbor poteza.

### **Kretanje po povrsi**

Nakon što smo definisali zakone kretanja za svaku figuru, naredni ključni korak jeste omogućiti korisniku da izvrši poteze u okviru igre. Funkcija makeMove ima zadatak da realizuje pomjeranje figure sa početne pozicije na ciljanu, uz sve prateće izmjene na tabli i stanju igre koje takav potez nosi sa sobom.

U okviru ove funkcije prvo se osigurava da je argument koji se prosleđuje zaista validan objekat klase Move. Zatim se na tabli početno polje na kojem se nalazi figura označava kao prazno, dok se na polju na koje se figura pomjera postavlja odgovarajuća figura. Ukoliko se radi o promociji pješaka, funkcija će u skladu s tim zamijeniti pješaka za željenu figuru, što je važan specijalni slučaj u šahu.

Svaki odigrani potez se zatim evidentira u listu odigranih poteza, što omogućava praćenje toka igre i eventualno vraćanje poteza unazad. Funkcija potom mijenja indikator koji označava koji je igrač na redu, čime se prelazi red na protivnika.

Posebna pažnja je posvećena i ažuriranju pozicije kralja ukoliko je on pomjeren, jer je to od presudne važnosti za provjeru šaha i mat pozicija u daljem toku igre. Takođe, funkcija obrađuje i druge specifične situacije povezane s potezima, kao što su en passant i rošada, ali detaljnije o tim posebnim pravilima će biti obrađeno u zasebnim poglavljima.

Pored funkcije za izvršavanje poteza, implementirana je i funkcija undoMove koja omogućava vraćanje posljednjeg poteza. Ovo je posebno korisno za vraćanje stanja igre u slučaju greške ili prilikom implementacije algoritama koji istražuju različite opcije poteza (kao što je Monte Carlo Tree Search ili minimax). Funkcija provjerava da li postoji potez za poništavanje, zatim vraća figuru na početnu poziciju i figuru koja je eventualno bila pojedena na polje sa kojeg je figura pomjerena. Takođe, vraća se red na igrača koji je prethodno imao potez, kao i pozicije kraljeva ukoliko su pomjereni.

Osim osnovnih poteza, undoMove funkcija vraća i stanje koje se odnosi na specijalne situacije kao što su prava na rošadu i stanje povezano sa en passant pravilom, čime se obezbjeđuje kompletna konzistentnost stanja igre nakon vraćanja poteza. Na ovaj način, korisnik ima potpunu kontrolu nad istorijom igre, a sistem je pripremljen za dalji razvoj i testiranje složenijih algoritama.

### **Moguci i legalni potezi**

Funkcija getValidMoves predstavlja srž logike za generisanje legalnih poteza u šahovskoj igri, odnosno onih poteza koji ne izlažu kralja šahu, čime se održava pravilo da igrač ne smije sebi dozvoliti da ostane u šahu. Prvo se poziva funkcija checkForPinsAndChecks koja provjerava da li je kralj trenutno ugrožen, kao i da identifikuje figuru koje su pinovane (vezane) i figure koje prijete kralju. Na osnovu ovih informacija, igra prilagođava generisanje poteza.

Ako je kralj pod šahom, funkcija dodatno filtrira poteze: ukoliko postoji samo jedan napadač, dozvoljeni potezi su oni koji ili pomjeraju kralja ili blokiraju liniju napada ili uklanjaju napadača. U slučaju dvostrukog šaha (dva ili više napadača), jedina opcija je pomjeranje kralja, jer nije moguće blokirati ili uzeti više napadača odjednom. Ukoliko kralj nije pod šahom, generišu se svi mogući potezi za aktivnog igrača.

Na kraju, getValidMoves funkcija detektuje krajnje stanje igre – ako nema nijednog legalnog poteza i kralj je u šahu, to znači da je mat, dok odsustvo poteza a bez šaha znači remi (stalemate). Ova funkcija vraća listu svih validnih poteza koji su dozvoljeni prema pravilima šaha.

S druge strane, pomoćna funkcija getAllPossibleMoves generiše sve moguće poteze bez dodatnih provjera šahovskih pravila kao što je izlaganje kralja šahu. Ova funkcija iterira kroz svaki položaj na tabli i poziva specifične funkcije za generisanje poteza za svaku figuru aktivnog igrača. Za svaku figuru koristi se mapiranje sa naziva figure na odgovarajući karakter koji se koristi kao ključ u rječniku funkcija kretanja, što omogućava fleksibilnost i modularnost koda. Posebna pažnja je posvećena kralju, jer mu se može proslijediti dodatni parametar za uključenje ili isključenje generisanja poteza vezanih za rošadu.

Ovim pristupom, implementacija jasno odvaja koncept generisanja svih teoretskih poteza od generisanja samo onih koji su dozvoljeni u skladu sa šahovskim pravilima, što predstavlja važnu osnovu za dalje implementacije logike igre, umjetne inteligencije ili korisničkog interfejsa.

### **Pinovi I sahovi**

Funkcija checkForPinsAndChecks je ključna za razumijevanje trenutne sigurnosne situacije kralja na tabli, jer određuje da li je kralj pod šahom, kao i koje figure su “pinovane” odnosno vezane, tj. koje se ne smiju pomjeriti jer bi time otkrile kralja napadu. Ova funkcija detaljno provjerava sve moguće pravce napada protivničkih figura.

Prvo se određuju boje igrača na potezu i njihova suprotna boja, kao i trenutna pozicija kralja koji je na potezu. Zatim se kreće provjera u svih osam mogućih pravaca (vertikalno, horizontalno i dijagonalno) gdje protivničke figure poput topa, lovca i kraljice mogu direktno ugroziti kralja. Tokom ove provjere, ako se nađe saveznički komad koji može biti pinovan, funkcija ga označava kao potencijalni pin, ali ako naiđe na drugu savezničku figuru u tom pravcu, zaustavlja provjeru jer pin ne može postojati iza više figura. Ukoliko se naiđe na neprijateljsku figuru koja može napasti iz tog pravca (uzimajući u obzir tip figure i udaljenost), funkcija odlučuje da li je kralj direktno u šahu ili je neka saveznička figura vezana.

Posebna pažnja se obraća i na specifične napade pješaka (koji mogu napasti samo dijagonalno i samo ako su u određenim redovima) i kralja koji može napasti susjedna polja. Nakon toga, funkcija provjerava i napade skakača (konja), jer oni ne mogu biti detektovani u linearnoj provjeri, već se pregledaju njihovi specifični L-oblik potezi.

Na kraju, funkcija vraća tri vrijednosti: da li je kralj u šahu (inCheck), listu pinovanih figura (pins) i listu prijetnji šahom (checks). Ove informacije se koriste u kasnijim fazama programa za generisanje samo legalnih poteza koji neće ugroziti kralja, što predstavlja osnovu sigurnosti igre i pravilnog provođenja pravila šaha.

### **Specijalne kretnje**

Nakon što smo detaljno obradili osnovna kretanja figura i pravila koja određuju legalnost poteza, važno je obraditi specijalne šahovske poteze kao što su rošada (castling) i en passant. Ovi potezi predstavljaju izuzetke u odnosu na standardna pravila kretanja figura i zahtijevaju dodatnu logiku i provjere da bi se ispravno implementirali.

Rošada je posebna vrsta poteza kralja i topa koji omogućava brzo pomjeranje kralja za dvije pozicije ka jednoj strani table, dok se top istovremeno pomjera na polje odmah do kralja. Ovaj potez je dozvoljen samo ako kralj nije u šahu, ako se kralj ni u jednom trenutku kretanja kroz polja ne nalazi pod napadom, i ako su polja između kralja i topa slobodna. Funkcija getCastleMoves upravlja ovom logikom tako što prvo provjerava da kralj nije u šahu, zatim poziva zasebne funkcije za kratku i dugu rošadu (getKingsideCastleMoves i getQueensideCastleMoves). One provjeravaju da li su potrebna polja prazna i da nisu pod napadom koristeći funkciju squareUnderAttack, koja simulira poteze protivnika da provjeri prijetnju na određenim poljima. Ukoliko su uslovi zadovoljeni, u listu mogućih poteza se dodaje potez rošade označen posebnom zastavicom.

Implementacija ove logike omogućava da program prepozna i legalno izvrši rošadu u svim situacijama koje pravila šaha dopuštaju, ali i sprječava nelegalne pokušaje rošade kada su uslovi narušeni, čime se čuva integritet igre.

Na sličan način, potez en passant zahtijeva posebnu provjeru jer predstavlja mogućnost da pješak uhvati protivničkog pješaka koji je pomjeren za dva polja unaprijed i time “preskočio” polje gdje bi inače mogao biti uhvaćen. Ova posebnost igre se u kodu prati kroz varijablu enPassantPossible, koja pamti trenutno polje gdje je en passant moguć. Funkcije koje generišu poteze pješaka uzimaju u obzir ovu situaciju i dodaju en passant poteze u listu validnih poteza ako su uslovi zadovoljeni.

Sve ove posebne situacije predstavljaju dodatne slojeve logike u igri i njihova pravilna implementacija je ključna za punu funkcionalnost šahovskog motora. Time se omogućava igranje šaha u skladu sa pravilima, bez izuzetaka ili nelegalnih poteza, što je od posebnog značaja za razvoj kvalitetnog i pouzdanog šahovskog programa.

## Klasa Move

Klasa Move predstavlja jedan šahovski potez i služi za lakše rukovanje informacijama o pomjeranju figura na tabli. Njena osnovna funkcija je da čuva podatke o početnoj i krajnjoj poziciji figure, kao i dodatne informacije koje su specifične za šahovski potez, poput en passant hvatanja, promocije pješaka i rošade.

Za internu reprezentaciju koristi se sistem koordinata gdje su redovi i kolone mape između oznaka na tabli (kao što su "a1", "h8") i indeksa u listi koja predstavlja tablu. Ova konverzija omogućava lakšu komunikaciju između šahovske notacije koju ljudi koriste i numeričkih indeksa koje računar koristi za praćenje stanja igre.

U konstruktoru klase čuvaju se početne i krajnje pozicije poteza, figura koja se pomjera, kao i figura koja je eventualno pojedena na odredišnoj poziciji. Posebna pažnja posvećena je specijalnim slučajevima: ako je potez en passant, figura koja se uklanja nije na krajnjoj poziciji poteza već na posebnom polju; ako je pješak dostigao protivnički kraj table, potez može biti promocija i tada se čuva informacija o tipu figure u koju pješak biva promovisan; ako je potez rošada, to se također evidentira zastavicom. Klasa također implementira jedinstveni identifikator za svaki potez koji se računa na osnovu koordinata početnog i krajnjeg polja, što omogućava jednostavno poređenje poteza u kasnijim dijelovima programa. To znači da se dva poteza smatraju jednakim ako imaju iste početne i krajnje koordinate, bez obzira na druge detalje.

Osim toga, klasa pruža funkciju koja potez pretvara u standardnu šahovsku notaciju, što je korisno za prikaz poteza igračima ili za zapisivanje partije.

Sve u svemu, klasa Move omogućava efikasno i pregledno upravljanje svim relevantnim podacima vezanim za pojedinačne poteze u šahovskoj igri, olakšavajući implementaciju logike igre i interakciju s korisnikom.

# Montekarlo algoritam

## Uvod

Monte Carlo Tree Search (MCTS) je savremena algoritamska tehnika koja se koristi za donošenje odluka u složenim problemima, posebno u igrama s potpunom informacijom poput šaha, go-a ili drugih strateških igara. Ovaj algoritam predstavlja kombinaciju istraživanja i eksploatacije kroz nasumične simulacije (Monte Carlo simulacije) koje se koriste za procjenu kvaliteta mogućih poteza. Za razliku od tradicionalnih pretraživačkih algoritama koji pokušavaju ispitati svaki mogući slijed poteza, MCTS gradi pretraživačko stablo selektivno, fokusirajući se na najperspektivnije dijelove prostora poteza.

Glavna prednost MCTS-a je njegova sposobnost da samostalno balansira između istraživanja novih, neistraženih poteza i iskorištavanja već poznatih dobrih poteza. Ovo se postiže iterativnim procesom koji se sastoji od četiri ključne faze: selekcije, ekspanzije, simulacije i propagacije rezultata unazad kroz stablo. Time MCTS može efikasno pristupiti velikim i složenim prostorima mogućih poteza bez potrebe za ručnim definisanjem heuristika ili opsežnim pretraživanjem.

Zbog svoje fleksibilnosti i relativno jednostavne implementacije, MCTS je postao vrlo popularan u oblasti vještačke inteligencije za igre, ali se koristi i u drugim domenima gdje je potrebno donositi odluke u složenim i neizvjesnim okruženjima. Primjeri takvih primjena uključuju robotsku navigaciju, optimizaciju i planiranje u realnom vremenu.

## Osnovne faze Monte Carlo Tree Search algoritma

Monte Carlo Tree Search algoritam sastoji se od četiri osnovne faze koje se ponavljaju u svakoj iteraciji kako bi se postupno izgradio i unaprijedio pretraživački stablo. Te faze su: selekcija, ekspanzija, simulacija i propagacija (backpropagation).

**Selekcija** predstavlja početnu fazu u kojoj se, počevši od korijenskog čvora, biraju podčvorovi na osnovu određenih kriterija, najčešće koristeći formulu Upper Confidence Bound for Trees (UCT). Cilj selekcije je pronaći čvor koji nije u potpunosti istražen, a koji obećava najbolje šanse za dobar ishod. Tokom ovog procesa, algoritam balansira između istraživanja novih, neistraženih puteva i eksploatacije već poznatih, profitabilnih puteva.

Nakon što se dođe do čvora koji nije potpuno istražen ili koji predstavlja završno stanje igre, prelazi se u fazu **ekspanzije**. U ovoj fazi se u stablo dodaje jedan ili više novih čvorova koji predstavljaju nove moguće poteze ili stanja igre. Ekspanzija omogućava algoritmu da proširi područje pretraživanja i uključi nove informacije u donošenje odluka.

Sljedeća faza je **simulacija**, poznata i kao "rollout". U ovoj fazi, sa novog proširenog čvora, algoritam nasumično odigrava partiju ili seriju poteza do kraja igre, odnosno do neke terminalne pozicije. Simulacija daje procjenu potencijalnog ishoda pozicije bez potrebe za kompletnom analizom svih mogućnosti, što znatno ubrzava proces pretraživanja.

Posljednja faza je **propagacija rezultata** (backpropagation), u kojoj se dobijeni rezultat simulacije prenosi nazad kroz čvorove koji su učestvovali u selekciji i ekspanziji. Tokom propagacije, ažuriraju se statistike poput broja posjeta i uspješnosti poteza, što omogućava algoritmu da u budućim iteracijama donosi bolje odluke na osnovu prikupljenih podataka.

Ove četiri faze se neprestano ponavljaju dok god traje vrijeme za razmišljanje ili dok se ne postigne određeni broj iteracija. Tako MCTS algoritam izgrađuje i usavršava svoje pretraživačko stablo, fokusirajući se na najperspektivnije opcije i time povećavajući šanse za donošenje optimalne odluke.

## Matematičke osnove: UCT formula

Jedan od ključnih elemenata Monte Carlo Tree Search algoritma je metoda odabira čvora tokom faze selekcije. Za ovu svrhu koristi se tzv. UCT formula (Upper Confidence Bound for Trees), koja balansira između dva suprotstavljena cilja: eksploatacije i istraživanja. Eksploatacija podrazumijeva fokusiranje na poteze za koje se već zna da daju dobre rezultate, dok istraživanje znači testiranje manje poznatih ili neistraženih poteza kako bi se otkrile nove, potencijalno bolje opcije.

UCT formula matematički izražava ovaj balans i računa vrijednost svakog čvora na sljedeći način:

gdje je:

* broj pobjeda (ili ukupni rezultat) ostvareni kroz taj čvor,
* broj posjeta tom čvoru,
* ukupan broj posjeta roditeljskom čvoru,
* parametar istraživanja (eng. exploration constant) koji određuje koliko će algoritam preferirati istraživanje naspram eksploatacije.

Prvi član formule, ​​, predstavlja prosječni uspjeh čvora, odnosno koliko je dobar taj potez na osnovu do sada prikupljenih podataka. Drugi član, , nagrađuje čvorove koji su manje posjećeni, podstičući istraživanje novih ili nedovoljno isprobanih poteza.

Parametar cc ima ključnu ulogu u ponašanju algoritma – veće vrijednosti cc rezultiraju većim fokusom na istraživanje, dok manje vrijednosti povećavaju eksploataciju poznatih dobrih poteza. Odabir optimalne vrijednosti zavisi od prirode problema i zahtjeva aplikacije.

Korištenjem UCT formule, MCTS algoritam inteligentno i efikasno bira koje grane stabla će dalje širiti i ispitivati, čime izbjegava prekomjerno ispitivanje nepotrebnih ili loših poteza i usmjerava računske resurse ka najperspektivnijim opcijama.

## Konkretna primjena

## Prednosti i izazovi MCTS algoritma

Monte Carlo Tree Search (MCTS) donosi značajne prednosti u oblasti algoritamskog pretraživanja i donošenja odluka, naročito u složenim igrama i problemima sa velikim prostorom mogućnosti. Jedna od najvažnijih prednosti jeste njegova skalabilnost – algoritam može raditi efikasno i u situacijama kada je broj mogućih poteza ogroman, jer ne zahtijeva potpunu pretragu stabla. Umjesto toga, fokusira se na najperspektivnije puteve koristeći balans istraživanja i eksploatacije, što omogućava brzo pronalaženje kvalitetnih rješenja čak i sa ograničenim vremenom za razmišljanje.

Još jedna važna prednost MCTS-a je njegova nezavisnost od domene. Za razliku od klasičnih heurističkih algoritama, MCTS ne zahtijeva unaprijed definisane heuristike ili ekspertsko znanje o problemu, već samostalno uči kroz simulacije i statističku analizu. Ovo ga čini univerzalnim alatom koji se može primijeniti na različite igre i probleme, od društvenih igara do optimizacije i robotske navigacije.

Međutim, pored svojih prednosti, MCTS ima i određene izazove. Prije svega, algoritam je računski zahtjevan, jer zahtijeva veliki broj simulacija da bi dao pouzdane procjene. To može biti ograničavajuće u realnim aplikacijama sa strogim vremenskim ograničenjima ili na uređajima sa ograničenim resursima. Također, kvalitet rezultata jako zavisi od kvaliteta simulacija – nasumične simulacije mogu biti neprecizne u složenim domenama, što utiče na preciznost procjena.

Još jedan izazov predstavlja upravljanje eksplozijom stabla u veoma kompleksnim igrama, gdje broj mogućih poteza brzo raste. Iako MCTS efikasno usmjerava pretragu, bez dodatnih optimizacija ili integracije sa heuristikama, može izgubiti efikasnost na vrlo dubokim ili specijaliziranim problemima.

Zbog ovih izazova, u praksi se često kombinuju MCTS sa dodatnim tehnikama, kao što su neuronske mreže za evaluaciju pozicija, paralelizacija algoritma radi ubrzanja, ili primjena specifičnih heuristika koje pomažu u bržem sužavanju pretrage. Time se postiže bolje balansiranje između tačnosti i brzine, te omogućava primjena MCTS-a u realnim, složenim okruženjima.

## Moguća poboljšanja i budući pravci razvoja

Iako je Monte Carlo Tree Search (MCTS) već dokazao svoju efikasnost u mnogim primjenama, postoje brojni pravci za njegovo dalje unapređenje i optimizaciju. Jedan od najznačajnijih pravaca je dublja integracija neuronskih mreža u samu strukturu algoritma. Korištenjem dubokog učenja za evaluaciju pozicija i procjenu rezultata simulacija, moguće je značajno poboljšati tačnost i brzinu donošenja odluka. Primjeri takvog pristupa vidimo u savremenim AI sistemima za igre poput AlphaGo, gdje je MCTS povezan s konvolucionim neuronskim mrežama koje procjenjuju vrijednost pozicije i daju preporuke za poteze.

Još jedan pravac razvoja odnosi se na paralelizaciju MCTS algoritma, odnosno korištenje više procesorskih jezgara ili distribuiranih sistema za izvođenje većeg broja simulacija u istom vremenskom okviru. Paralelizacija može značajno ubrzati pretragu i omogućiti dublju analizu stabla poteza, što je naročito korisno u realno-vremenskim aplikacijama i igrama sa velikim brojem mogućnosti.

Također, moguće je raditi na poboljšanju politika selekcije i ekspanzije unutar MCTS-a, uključujući razvoj složenijih heuristika ili korištenje metoda učenja sa pojačanjem (reinforcement learning) koje bi dodatno usmjeravale pretragu ka optimalnijim granama stabla. Ovo može smanjiti broj potrebnih simulacija i povećati kvalitet konačnih odluka.

Upravljanje memorijskim zahtjevima i efikasna optimizacija strukture podataka također predstavljaju važne oblasti za budući rad, jer u nekim složenim igrama ili problemima stablo može brzo narasti do veličina koje otežavaju njegovo skladištenje i obradu.

Konačno, istraživanja u pravcu kombinovanja MCTS sa drugim algoritmima pretrage i optimizacije, kao što su minimax, alfa-beta rezanje ili evolucioni algoritmi, mogu donijeti nove sinergije i dodatno unaprijediti performanse u specifičnim domenama.

# Testiranje

# Zakljucak