

**priručnik za polaznike 2019 Srce**

**Programiranje u *Pythonu***

D460

Ovu inačicu priručnika izradio je autorski tim Srca u sastavu:

Autor: Marko Hruška

Recenzent: Hrvoje Backović

Urednica: Petra Gmajner

Lektorica: Mia Kožul



Svučilište u Zagrebu

Sveučilišni računski centar

Josipa Marohnića 5, 10000 Zagreb

edu@srce.hr

ISBN 978-953-8172-49-6 (meki uvez)

ISBN 978-953-8172-50-2 (PDF)

Verzija priručnika D460-20200712



|  |  |
| --- | --- |
|  | Ovo djelo dano je na korištenje pod licencom *Creative Commons Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 4.0 međunarodna*. Licenca je dostupna na stranici:  http://creativecommons.org/licenses/by-nc-sa/4.0/. |

**Sadržaj**

[Uvod 1](#_Toc45526248)

[1. Specifičnosti programskoga jezika *Python* 3](#_Toc45526249)

[1.1. Lista u obrnutom poretku 3](#_Toc45526250)

[1.2. Niz znakova u obrnutom poretku 5](#_Toc45526251)

[1.3. Zamjena vrijednosti dviju varijabli u jednoj liniji 5](#_Toc45526252)

[1.4. Spremanje vrijednosti u varijable iz liste 6](#_Toc45526253)

[1.5. Ulančani operatori usporedbe 6](#_Toc45526254)

[1.6. Konstrukcija: for/else 7](#_Toc45526255)

[1.7. Ternarni operator 7](#_Toc45526256)

[1.8. Poziv funkcije pomoću ternarnog operatora 8](#_Toc45526257)

[1.9. Naredba pass 8](#_Toc45526258)

[1.10. Povratna vrijednost funkcije – vraćanje više vrijednosti 10](#_Toc45526259)

[1.11. Interaktivni operator '\_' 10](#_Toc45526260)

[1.12. Vježba: Specifičnosti programskoga jezika *Python* 11](#_Toc45526261)

[2. Napredno korištenje funkcija 13](#_Toc45526262)

[2.1. Rekurzivne funkcije 13](#_Toc45526263)

[2.2. Bezimene funkcije 16](#_Toc45526264)

[2.3. Ugniježđene funkcije 19](#_Toc45526265)

[2.4. Generatorske funkcije 22](#_Toc45526266)

[2.5. Vježba: Napredno korištenje funkcija 26](#_Toc45526267)

[2.6. Pitanja za ponavljanje: Napredno korištenje funkcija 28](#_Toc45526268)

[3. Iznimke 29](#_Toc45526269)

[3.1. Primjeri iznimaka 32](#_Toc45526270)

[3.2. Obrada iznimaka 35](#_Toc45526271)

[3.3. Podizanje iznimaka 38](#_Toc45526272)

[3.4. Istovremena obrada više tipova iznimaka 39](#_Toc45526273)

[3.5. Prosljeđivanje iznimaka 39](#_Toc45526274)

[3.6. Završne akcije: else i finally 40](#_Toc45526275)

[3.7. Vježba: Iznimke 44](#_Toc45526276)

[3.8. Pitanja za ponavljanje: Iznimke 44](#_Toc45526277)

[4. Objektno orijentirano programiranje 45](#_Toc45526278)

[4.1. Razredi i objekti 46](#_Toc45526279)

[4.2. Vrste metoda 48](#_Toc45526280)

[4.3. Vlasništvo varijabli 54](#_Toc45526281)

[4.4. Vježba: Objektno orijentirano programiranje - I 57](#_Toc45526282)

[4.5. Vidljivost varijable – public, protected, private 58](#_Toc45526283)

[4.6. Svojstva 60](#_Toc45526284)

[4.7. Vježba: Objektno orijentirano programiranje - II 63](#_Toc45526285)

[4.8. Nasljeđivanje (engl. *Inheritance*) 64](#_Toc45526286)

[4.9. Preopterećenje (engl. *Overloading*) 67](#_Toc45526287)

[4.10. Nadjačavanje (engl. *Overriding*) 69](#_Toc45526288)

[4.11. Vježba: Objektno orijentirano programiranje - III 71](#_Toc45526289)

[4.12. Apstraktni razred (engl. *Abtract Base Classes*) 72](#_Toc45526290)

[4.13. Ulančavanje metoda 75](#_Toc45526291)

[4.14. Vježba: Objektno orijentirano programiranje - IV 77](#_Toc45526292)

[4.15. Pitanja za ponavljanje: Objektno orijentirano programiranje 77](#_Toc45526293)

[5. Algoritmi sortiranja 79](#_Toc45526294)

[5.1. Sortiranje biranjem (engl. *Selection sort*) 81](#_Toc45526295)

[5.2. Sortiranje zamjenom susjednih elemenata (engl. *Bubble sort*) 82](#_Toc45526296)

[5.3. Poboljšani algoritam zamjene susjednih elemenata 85](#_Toc45526297)

[5.4. Usporedba funkcije sorted() i metode sort() 86](#_Toc45526298)

[5.5. Vježba: Algoritmi sortiranja 88](#_Toc45526299)

[5.6. Pitanja za ponavljanje: Algoritmi sortiranja 88](#_Toc45526300)

[6. Oblikovni obrasci 89](#_Toc45526301)

[6.1. Upoznavanje s oblikovnim obrascima 90](#_Toc45526302)

[6.2. Jedinstveni objekt (engl. *Singleton Pattern*) 91](#_Toc45526303)

[6.3. Promatrač (engl. *Observer Pattern*) 93](#_Toc45526304)

[6.4. Vježba: Oblikovni obrasci 98](#_Toc45526305)

[6.5. Pitanja za ponavljanje: Oblikovni obrasci 98](#_Toc45526306)

[Dodatak: Rješenja vježbi 99](#_Toc45526307)

[Literatura 132](#_Toc45526308)

# Uvod

Svrha ovoga priručnika jest upoznavanje s naprednijim konceptima za konstrukciju programskoga kôda u programskom jeziku *Python*.

Sadržaj ovoga priručnika nadovezuje se na priručnik naziva: *Osnove programiranja (Python)*. Preduvjet za razumijevanje gradiva ovoga priručnika jest poznavanje rada na računalu i poznavanje osnova programiranja u programskom jeziku *Python*. Priručnik se sastoji od devet poglavlja koja se odrađuju u četiri dana, po četiri sata dnevno, koliko traje tečaj. Na kraju svakoga poglavlja nalaze se vježbe i pitanja za ponavljanje koje će polaznici prolaziti zajedno s predavačem. Mogući savjeti i zanimljivosti istaknuti su u okvirima sa strane.

Na ovom tečaju polaznici će naučiti naprednije koncepte programiranja u programskom jeziku *Python*. Tako naučeni koncepti primjenjivi su i na većinu ostalih programskih jezika, te istodobno omogućavaju lakše snalaženje prilikom nadograđivanja vlastitoga znanja nakon tečaja uz pomoć *web*-pretraživača i ostale literature. Nakon tečaja polaznici će posjedovati kompetencije za rješavanje složenijih programskih zadataka u programskom jeziku *Python*.

*Python* je interpreterski programski jezik. Kod interpreterskih programskih jezika, naredbe se prevode u trenutku izvođenja programa te se svaka naredba prevodi u jednu ili više strojnih naredbi što ovisi o složenosti. Kod ovakvog načina izvršavanja programskoga kôda nema potrebe za kompajliranjem prije izvršavanja, tj. prevođenjem u strojni jezik. Programi pisani u *Pythonu* se za posljedicu izvršavaju sporije od istih programa koji su pisani u jezicima čiji se programski kôd prije izvršavanja prevodi u:

* *strojni jezik*, primjeri takvih jezika su: *C*, *C++*
* *bytecode*, primjer takvoga jezika je: *Java*.

Programski kôd napisan u *Pythonu* također je moguće prevesti u *bytecode* te se time povećava brzina izvođenja programa. Prevođenje u *bytecode* neće se obrađivati u ovom priručniku.

***Python* – verzije**

Verzija *Pythona* je označena s A.B.C. Svako od ta tri slova označava promjene koje su se dogodile prema važnosti. Slovo C označava sitne promjene, slovo B označava veće promjene, dok slovo A označava velike promjene.

Programi pisani u programskom jeziku *Python* su kraći, a i za njihovo pisanje utrošak vremena je puno manji. *Python* programerima dopušta nekoliko programskih paradigmi: strukturno, objektno orijentirano i aspektno orijentirano programiranje. Svojstvo koje krasi *Python* je čista i jednostavna sintaksa.

U ovom tečaju obrađuje se strukturno programiranje i objektno orijentirano programiranje. Kod strukturnog programiranja glavni je naglasak za kontrolu programa na korištenju struktura poput funkcija, metoda, odluka, petlji. Ovakav način programiranja omogućava programerima da se program razlomi na manje logičke dijelove, tj. u funkcije te se tako povećava modularnost programskoga kôda koji je moguće višekratno iskoristiti. Ovakav pristup idealan je za kraće logičke programe, no kod velikih projekata nije dovoljno učinkovit i nakon nekog vremena programerima je sve teže razvijati nove komponente, tj. funkcionalnosti. Objektno orijentirano programiranje (kraće OOP) jedan je od najpopularnijih pristupa razvoju programskoga kôda današnjice. Kod ovog pristupa razvoja programskoga kôda glavni je cilj razvijanje objekata koji imaju funkcionalnosti implementirane u metode. Ovaj pristup je trenutačno idealan za razvoj većih projekata. Aspektno orijentirano programiranje neće se obrađivati u ovom tečaju.

*Python* je 1991. godine kreirao programer Guido van Rossum. Verzije *Pythona* su:

***Službena dokumentacija***

Službenu *Python 3* dokumentaciju možete pronaći na URL-u: <https://docs.python.org/3/>

* *Python 1.0.* – 1991. godina
* *Python 2.0.* – 2000. godina, zadnja podverzija *Python 2.7* je objavljena 2010. godine te će *Python 2.7* biti podržan do 2020. godine
* *Python 3.0.* – 2008. godina.

U trenutku pisanja ovoga priručnika zadnje stabilne verzije su: *Python 3.7.2* i *Python 2.7.15*. U ovom priručniku koristi se *Python 3*. Za razvoj *Pythona* brine se neprofitna organizacija *Python Software Foundation*. Sintaksa *Python 2* i *Python 3* verzije nije kompatibilna. Razvijen je alat *2to3* koji prevodi programski kôd pisan u verziji *Python 2* u verziju *Python 3*.

U priručniku važni pojmovi pisani su **podebljano**. Ključne riječi, imena varijabli, funkcija, metoda i ostale programske konstrukcije pisane su drugačijim fontom od uobičajenog, na primjer: print(**"Hello World!"**). Nazivi na engleskom jeziku pisani su *kurzivom* i u zagradi, na primjer: "varijabla(engl. *variable*)"**.**

Također, zadatke za vježbu možemo podijeliti u 3 kategorije, a to su zadaci bez zvjezdice, zadaci s jednom zvjezdicom (\*) i zadaci s dvije zvjezdice (\*\*). Zadaci bez zvjezdice se karakteriziraju kao lagani zadaci, s jednom zvjezdicom kao zadaci srednje težine, dok se zadaci s dvije zvjezdice karakteriziraju kao teži zadaci.

**Programski kôd** pisan je na sljedeći način:

U prvom dijelu bit će napisan programski kôd.

|  |
| --- |
| **for** i **in** range(2):  print(**"Hello World!"**) |
| Izlaz:  Hello World!  U drugom dijelu bit će prikazan dobiven izlaz programskoga kôda.  Hello World! |

1. Specifičnosti programskoga jezika *Python*

Po završetku ovoga poglavlja polaznik će moći:

koristiti neke od mnogobrojnih specifičnosti programskoga jezika Python koje olakšavaju razvoj programskoga kôda.

Programski jezik *Python* u sebi sadrži neke specifičnosti koje nisu implementirane u svim programskim jezicima. Takve specifičnosti programerima omogućavaju brži razvoj programskoga kôda. U nastavku slijedi opis s primjerima nekoliko takvih specifičnosti koje su ugrađene u *Python*.

* 1. Lista u obrnutom poretku

***Metoda reversed()***

Metoda reversed() je često bolja varijanta od reverse() jer **ne kopira** objekt.

Metoda reversed() vraća iterator koji pristupa predanoj sekvenci u obrnutom poretku.

Uz korištenje ugrađene metode reverse() te nekih drugih zaobilaznih načina, *Python* ima ugrađen još jedan način kako elemente liste dohvatiti u obrnutom poretku. Sintaksa takvog ugrađenog načina dohvaćanja elemenata liste u obrnutom poretku je:

lista[::-1]

|  |
| --- |
| lista = [1, 2, 5, 4, 8, 3] obrnutaLista = lista[::-1]  print(**"Originalna lista:"**, lista) print(**"Obrnuta lista:"**, obrnutaLista) |
| Izlaz:  Originalna lista: [1, 2, 5, 4, 8, 3]  Obrnuta lista: [3, 8, 4, 5, 2, 1] |

Detaljnije objašnjenje ovoga postupka:

***Za one koji žele znati više***

Pronađite i proučite kako funkcionira klasa:

*slice(start, stop[, step])*

* Dohvaćanje elemenata od indeksa *start* do indeksa *stop-1*

lista[start:stop]

* Dohvaćanje elemenata od indeksa *start* pa do kraja liste

lista[start:]

* Dohvaćanje elemenata od početka liste pa do indeksa *stop-1*

lista[:stop]

Vrijednosti parametara start i stop moguće je zadati i kao negativne, tada se primijenjuju sljedeća pravila:

* Dohvaćanje zadnjeg elementa liste

lista[-1]

* Dohvaćanje zadnjih dvaju elemenata liste

lista[-2:]

* Dohvaćanje svih elemenata liste osim zadnjih dvaju elemenata

lista[:-2]

Uz parametre start i stop moguće je zadati i treći parametar step. Parametar step označava korak ispisa. U nastavku slijedi detaljniji opis:

* Dohvaćanje svakoga drugog elementa liste

lista[::2]

* Dohvaćanje svakoga drugog elementa liste, između indeksa 1 i 5

lista[1:5:2]

* Dohvaćanje svih elemenata liste u obrnutom poretku

lista[::-1]

* Dohvaćanje prvih dvaju elemenata liste u obrnutom poretku

lista[1::-1]

* Dohvaćanje zadnjih triju elementa u obrnutom poretku

lista[:-3:-1]

* Dohvaćanje svih elemenata osim zadnjih dvaju u obrnutom poretku

a[-3::-1]

|  |
| --- |
| >>> lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  >>> lista[2:5]  [2, 3, 4]  >>> lista[2]  2  >>> lista[:5]  [0, 1, 2, 3, 4]  >>> lista[:]  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  >>> lista[-1]  9  >>> lista[-2:]  [8, 9]  >>> lista[:-2]  [0, 1, 2, 3, 4, 5, 6, 7]  >>> lista[::2]  [0, 2, 4, 6, 8]  >>> lista[1:5:2]  [1, 3]  >>> lista[::-2]  [9, 7, 5, 3, 1]  >>> lista[::-1]  [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  >>> lista[1::-1]  [1, 0]  >>> lista[:-3:-1]  [9, 8]  >>> lista[-3::-1]  [7, 6, 5, 4, 3, 2, 1, 0] |

* 1. Niz znakova u obrnutom poretku

U prethodnom odjeljku obrađeno je dohvaćanje elemenata liste u obrnutom poretku dok se u ovom odjeljku obrađuje dohvaćanje elemenata niza znakova u obrnutom poretku. Nad nizom znakova nije moguće pozvati metodu reverse(). Sintaksa dohvaćanja elemenata niza znakova u obrnutom poretku jest:

***Metoda reversed()***

Metodu reversed() je moguće koristiti i kod nizova znakova.

nizZnakova[::-1]

|  |
| --- |
| nizZnakova = **"abcdefghijkl"** obrnutiNizZnakova = nizZnakova[::-1]  print(**"Originalni niz znakova:"**, nizZnakova) print(**"Obrnuti niz znakova:"**, obrnutiNizZnakova) |
| Izlaz:  Originalni niz znakova: abcdefghijkl  Obrnuti niz znakova: lkjihgfedcba |

* 1. Zamjena vrijednosti dviju varijabli u jednoj liniji

*Python* omogućava jednostavan način zamjene vrijednosti dviju varijabli u jednoj liniji. Sintaksa i grafički prikaz zamjene vrijednosti dviju varijabli:

b, a = a, b

|  |
| --- |
| a = 10 b = 20  print(a, b)  b, a = a, b  print(a, b) |
| Izlaz:  10 20  20 10 |

Kod ovakvog načina pridjeljivanja vrijednosti varijablama u pozadini se obavlja pakiranje i raspakiravanje n-terca (engl. *Tuple Packing/Unpacking*). U nastavku slijedi detaljnije pojašnjenje.

1. Pakiranje n-terca (engl. *Packing a tuple*) – sintaksa koja je prikazana u nastavku je jednostavna sintaksa koja omogućava kreiranje n-terca bez korištenja ustaljene sintakse.

x = 1, 2

Kreira se:

x = (1, 2)

1. Raspakiravanje n-terca (engl. *Unpacking a tuple*) – sintaksa koja omogućava da se dane vrijednosti zapakirane u n-terac raspakiraju i spreme u varijable.

x = 1, 2

a, b = x # a, b = (1, 2)

Nakon izvršavanja ovoga kôda, imamo: a = 1, b = 2. Vrijednost n-terca x raspakirava se u 2 varijable a, b. Broj varijabli mora točno odgovarati broju elemenata u n-tercu ili će doći do pogreške.

* 1. Spremanje vrijednosti u varijable iz liste

Programski jezik *Python* ima funkcionalnost pomoću koje se vrijednosti elemenata upisanih u listu mogu na jednostavan način preslikati u pojedinačne varijable. Sintaksa i grafički prikaz preslikavanja vrijednosti:

***Napomena***

Identično kao i kod prethodnog primjera, u pozadini se radi pakiranje i raspakiravanje n-terca (engl. *Tuple Packing/Unpacking*). U pozadini se izvršava iteracija te se n-tercu slijedno dodijeljuju vrijednosti.

var1, var2, var3 = [1, 2, 3]

|  |
| --- |
| lista = [1, 2, 3]  a, b, c = lista  print(a, b, c) |
| Izlaz:  1 2 3 |

* 1. Ulančani operatori usporedbe

Ulančavanje operatora usporedbe funkcionalnost je koja kod mnogobrojnih programskih jezika ne postoji. Ako želimo provjeriti je li vrijednost neke varijable u intervalu [10, 20], u programskom jeziku *Pythonu* dovoljno je napisati kôd sljedeće sintakse:

10 <= a <= 20

Dok je u većini ostalih programskih jezika za takvu usporedbu potrebno kreirati ovakav izraz:

10 <= a && a <= 20

|  |
| --- |
| a = 10 print(3 < a < 20) print(10 < a < 20) |
| Izlaz:  True  False |

Prednost ulančavanja operatora usporedbe je u tome da ako je a složeni izraz, na primjer, poziv neke spore funkcije, kod ulančanog operatora, takav izraz će se evaluirati samo jednom, a ne više puta čime se ubrzava izvršavanje programa.

* 1. Konstrukcija: for/else

Petlje su od vitalnog značenja za svaki programski jezik. U programskom jeziku *Python* koriste se petlje for i while. Svaki programer je upoznat s njihovim funkcioniranjem, no malo programera je upoznato sa specifičnošću kojom se bavi ovo potpogavlje.

Petlja for može imati i svoj else dio. else dio petlje for izvršava se samo u slučaju kada petlja for bez prekida odradi svoj posao, tj. kada nije prekinuta pozivom naredbe break.

Ovakva konstrukcija najviše se koristi kada for petlja prolazi kroz neku strukturu podataka i traži neki određeni element. Kroz takvo pretraživanje mogu se dogoditi dva slučaja:

* Traženi element **postoji** i nakon što petlja for dođe do njega poziva se naredba break kojom se petlja prekida, u tom slučaju else dio petlje for se ne izvršava. U nastavku je prikaz programskoga kôda koji zorno prikazuje ovaj slučaj:

|  |
| --- |
| lista = [1, 2, 3, 4, 5, 6]  **for** element **in** lista:  **if** element == 5:  **break else**:  print(**"Nije pronađeno!"**) |
| Izlaz:  *{Ništa}* |

* Traženi element **ne postoji** i nakon što petlja for prođe po svim elementima ona završava s daljnjim radom. U tom slučaju izvršava se programski kôd koji se nalazi u else dijelu:

|  |
| --- |
| lista = [1, 2, 3, 4, 5, 6]  **for** element **in** lista:  **if** element == 0:  **break else**:   print(**"Nije pronađeno!"**) |
| Izlaz:  Nije pronađeno! |

* 1. Ternarni operator

Ternarni operator omogućava da se funkcionalnost if-else naredbe napiše u jednom retku. Ternarni operator izgleda ovako:

[istinit\_izraz] if [logički\_izraz] else [lažan\_izraz]

* logički\_izraz – rezultat ovoga logičkog izraza je True ili pak False te se na temelju rezultata ovog izraza odlučuje koji dio programskoga kôda će se izvršiti
* istinit\_izraz – ovaj programski kôd izvršava se ako je rezultat logičkog izraza True
* lažan\_izraz – ovaj programski kôd izvršava se ako je rezultat logičkog izraza False.

|  |
| --- |
| x = 10 **if** (**True**) **else** 20 print(x)  x = 10 **if** (**False**) **else** 20 print(x) |
| Izlaz:  10  20 |

* 1. Poziv funkcije pomoću ternarnog operatora

U prethodnom potpoglavlju obrađen je ternarni operator koji omogućava pisanje if-else naredbe u jednom retku. U ovom poglavlju obrađuje se ternarni operator koji omogućava poziv jedne funkcije u slučaju da je logički izraz istinit (engl. *True*), a poziv druge funkcije u slučaju da je logički izraz laž (engl. *False*). Broj parametara koje primaju obje te funkcije mora biti identičan. U nastavku slijedi primjer poziva funkcije pomoću ternarnog operatora:

***Za one koji žele znati više***

Ovaj način pozivanja funkcija funkcionira jer ternarni operator vraća funkcijski objekt – svaki razred može biti funkcija ako implementira metodu \_\_call\_\_.

([fun1] if [izraz] else [fun2])([param1, …])

* fun1 – ova funkcija će se izvršiti ako je izraz istinit (engl. *True*)
* fun2 – ova funkcija će se izvršiti ako je izraz laž (engl. *False*)

|  |
| --- |
| **def** funPrva(vrijednost):  print(**"funPrva(), vrijednost:"**, vrijednost)  **def** funDruga(vrijednost):  print(**"funDruga(), vrijednost:"**, vrijednost)   x = **True** (funPrva **if** x **else** funDruga)(5) |
| Izlaz:  funPrva(), vrijednost: 5 |

* 1. Naredba pass

Ova naredba koristi se kada je potreban logički izraz, ali za taj izraz ne želimo da se izvrši nijedna naredba. Na primjer, gradimo if-elif-else uvjete i ako za neki uvjet ne želimo da se išta izvrši, tada ne možemo ostaviti samo praznu liniju u tijelu tog uvjeta, već je potrebno napisati naredbu pass.

|  |
| --- |
| rijec = **"Srxce"  for** e **in** rijec:  **if** e == **'x'**:   **else**:  print(e) |
| Izlaz: |

U gornjem primjeru možemo primijetiti da je logički uvjet postavljen tako da ako se u varijabli e nalazi malo slovo 'x', tada želimo da se izvrši "*ništa*", a u suprotnom neka se sva ostala slova ispisuju. Kada se ovako napisan program pokuša pokrenuti, dobiva se greška da program ne zadovoljava sintaksna pravila. U nastavku je primjer programskoga kôda u kojem se koristi naredba pass koja omogućava da se izvrši "*ništa*".

|  |
| --- |
| rijec = **"Srxce"  for** e **in** rijec:  **if** e == **'x'**:  **pass  else**:  print(e) |
| Izlaz:  S  r  c  e |

Također, moramo biti svjesni da naredba pass ne prekida izvršavanje linija programskoga kôda koje se u nekom tijelu nalaze ispod nje. Primjer programskoga kôda za ovaj slučaj nalazi se u nastavku.

|  |
| --- |
| rijec = **"Srxce" for** e **in** rijec:  **if** e == **'x'**:  **pass** print(**"Ključna riječ pass"**)  **else**:  print(e) |
| Izlaz:  S  r  Ključna riječ pass  c  e |

* 1. Povratna vrijednost funkcije – vraćanje više vrijednosti

Za razliku od mnogobrojnih programskih jezika, kao što su: C, C++, Java, programski jezik *Python* omogućava vraćanje više povratnih vrijednosti u pozivajući dio programa. U nastavku je prikazan primjer u kojem funkcija imena fun() vraća u pozivajući dio programa 5 povratnih vrijednosti te se tako vraćene vrijednosti spremaju u 5 različitih varijabli. **Napomena**: povratne vrijednosti koje se vraćaju zapakirane su u n-terac (engl. *tuple*).

|  |
| --- |
| **def** fun():  **return** 1, 2, 3, 4, 5  a, b, c, d, e = fun() print(a, b, c, d, e) |
| Izlaz:  1 2 3 4 5 |

* 1. Interaktivni operator '\_'

Ako radimo u konzoli programskoga jezika *Python*, svaki zadnji rezultat nekog izraza ili funkcije sprema se u privremenu varijablu imena '\_'. U nastavku slijedi primjer gdje se na konzolu ispisuje rezultat zbroja dvaju brojeva, nakon ispisa rezultata, pomoću varijable imena '\_' ponovo se ispisuje zadnja ispisana vrijednost.

|  |
| --- |
| >>> 10 + 6  16  >>> \_  16  >>> from math import sqrt  >>> sqrt(\_ + 9)  5.0  >>> \_  5.0 |

* 1. Vježba: Specifičnosti programskoga jezika *Python*

1. Napišite program unutar kojeg ćete u varijablu inicijalizirati listu proizvoljnog sadržaja (lista od barem 3 vrijednosti). Elemente liste spremite u pojedinačne varijable u jednoj liniji. Nakon što elemente liste imate spremljene u zasebnim varijablama, ispišite vrijednosti tih varijabli.
2. U dvije varijable inicijalizirajte dva različita proizvoljna broja. Nakon inicijalizacije ispišite vrijednosti varijabli na zaslon. Zamijenite vrijednosti tih dviju varijabli te ponovo ispišite njihove nove vrijednosti.
3. S tipkovnice učitajte broj proizvoljne vrijednosti. Za tako učitan broj provjerite zadovoljava li on uvjet: 10 < broj < 30, ako je uvjet zadovoljen, ispišite na ekran: "Zadovoljava!", ako pak uvjet nije zadovoljen, ispišite na ekran: "Ne zadovoljava!". Ovaj zadatak potrebno je riješiti korištenjem operatora usporedbe, bez korištenja logičkih operatora i logičkih izraza.
4. Kreirajte listu te ju popunite s 5 proizvoljnih vrijednosti (brojeva). Nakon toga učitajte proizvoljnu vrijednost (broj) s tipkovnice. Unutar petlje for provjerite nalazi li se vrijednost koju ste učitali s tipkovnice unutar liste. Ako tražena vrijednost (broj) postoji u listi, ispišite na ekran "Postoji!", u suprotnom ispišite na ekran "Ne postoji!".
5. S tipkovnice učitajte broj proizvoljne vrijednosti. Pomoću ternarnog operatora detektirajte je li učitani broj paran ili neparan. Ako je učitani broj paran, ispišite na ekran: "Paran!", ako pak je neparan, ispišite na ekran: "Neparan!".
6. Implementirajte dvije funkcije. Prva funkcija neka se zove paran() i ona neka ispisuje na ekran: "Paran!", druga funkcija neka se zove neparan() i ona neka ispisuje na ekran: "Neparan!". S tipkovnice učitajte broj proizvoljne vrijednosti te pomoću ternarnog operatora detektirajte je li učitana vrijednost parna ili neparna te ovisno o parnosti pozovite jednu od dviju funkcija koje ste prethodno kreirali.
7. Implementirajte funkciju prototipa rezultat(var1, var2), funkcija može primiti dvije vrijednosti. Tako kreirana funkcija neka u pozivajući dio programa vraća tri vrijednosti dobivene sljedećim aritmetičkim operacijama: var1\*var2, var1+var2, 10\*var1+var2. U glavnom dijelu programa pozovite prethodno implementiranu funkciju te njene povratne vrijednosti spremite u tri zasebne varijable proizvoljnog imena i ispišite ih.
8. \* Napišite program unutar kojeg će biti inicijalizirane dvije varijable proizvoljnog imena. U prvu varijablu spremite niz znakova: "Hello World!", a u drugu varijablu spremite listu sadržaja [1, 2, 3, 4]. Ispišite na ekran niz znakova i listu u obrnutom poretku na dva načina. Najprije vrijednosti ispišite pomoću petlje, a nakon toga vrijednosti ispišite na način koji je objašnjen u poglavlju 1.1. i 1.2.
9. Napredno korištenje funkcija

Po završetku ovoga poglavlja polaznik će moći koristiti složenije tipove funkcija:

rekurzivne funkcije

bezimene funkcije

ugniježđene funkcije

generatorske funkcije.

* 1. Rekurzivne funkcije

Rekurzivne funkcije su funkcije koje pozivaju same sebe. Programski kôd koji je napisan pomoću rekurzivnih funkcija je kraći, no rekurzivne funkcije imaju za posljedicu da njihovo izvođenje traje dulje. U nastavku slijedi primjer programskoga kôda koji prikazuje rekurzivni poziv funkcije. U gotovo svoj literaturi gdje se opisuju rekurzivne funkcije za početni primjer se uzima opis matematičke operacije faktorijeli – **n!**.

Uzmimo za primjer da naša rekurzivna funkcija mora izračunati koliko iznosi 4 faktorijela. Matematički gledano rezultat 4! se dobiva na sljedeći način 4\*3\*2\*1 = 24.

|  |
| --- |
| **def** faktorijel(x):  **if** x <= 1:  **return** 1  **else**:  **return** x \* faktorijel(x - 1)   rez = faktorijel(4) print(rez)  ***Napomena***  **Prototip** je zaglavlje funkcije.  **Implementacija** je tijelo funkcije. |
| Izlaz:  24 |

U gornjem primjeru vidimo da prototip naše funkcije izgleda faktorijel(x), dok u samoj implementaciji funkcije možemo raspoznati dva osnovna slučaja.

**Dva osnovna slučaja:**

* osnovni (bazni) slučaj – na temelju ovog uvjeta rekurzivno pozivanje funkcija završava,
* rekurzivni poziv – na temelju ovog uvjeta funkcija se ponovo rekurzivno poziva. Svakim novim rekurzivnim pozivom funkcija je sve bliže osnovnom slučaju.

Gornja funkcija faktorijel(x) je pozvana sve skupa 4 puta, jednom je pozvana iz glavnog programa, dok je 3 puta ta funkcija rekurzivno pozvala sama sebe.

U nastavku je detaljniji opis rekurzivnoga pozivanja funkcija gore prikazanoga programskog kôda. Rekurzivno pozivanje funkcija možemo podijeliti na 2 dijela, rekurzivno pozivanje do osnovnog koraka (koraci 1 – 4) i rekurzivno vraćanje natrag (koraci 5 – 8).

rez = faktorijel(4)

= 4 \* faktorijel(3)

= 3 \* faktorijel(2)

= 2 \* faktorijel(1)

= 1

= 2 \* 1

= 3 \* 2

= 4 \* 6

rez = 24

rez = faktorijel(4)

1

x = 4

4

8

|  |
| --- |
| **def** faktorijel(x):  **if** x <= 1:  **return** 1  **else**:  4 \* 6  **return** x \* faktorijel(x - 1)  7  x = 3  2  **def** faktorijel(x):  **if** x <= 1:  **return** 1  **else**:  3 \* 2  **return** x \* faktorijel(x - 1)  x = 2  3  **def** faktorijel(x):  6  **if** x <= 1:  **return** 1  **else**:  2 \* 1  **return** x \* faktorijel(x - 1)  x = 1  4  **def** faktorijel(x):  5  **if** x <= 1:  **return** 1  **else**:  **return** x \* faktorijel(x - 1) |

**Opis koraka**

1. Poziva se funkcija faktorijel(x) (u nastavku ovoga detaljnog opisa samo *funkcija*). Parametar x poprima vrijednost 4, kod ispitivanja uvjeta, uvjet if nije zadovoljen, te se izvađa else dio, tj. rekurzivno se poziva funkcija s vrijednošću argumenta x-1 tj. 3.
2. Ponavlja se postupak iz prethodnog koraka (korak 1.), no u ovom slučaju parametar x poprima vrijednost 3.
3. Ponavlja se postupak iz prethodnih koraka (korak 1. i 2.), no u ovom slučaju parametar x poprima vrijednost 2.
4. U ovom koraku parametar x poprima vrijednost 1, no za razliku od prethodnih koraka u ovom koraku uvjet if je zadovoljen.
5. Ovaj korak je ustvari osnovni (bazni) slučaj i u ovom koraku kreće rekurzivno vraćanje vrijednosti natrag, u ovom slučaju u pozivajuću funkciju vraća se vrijednost 1.
6. Rekurzivno se vraća vrijednost parametra x, tj. 2 iz tog koraka rekurzivne funkcije pomnožen s povratnom vrijednošću osnovnoga (prethodnog) koraka, tj. vrijednošću 1. Rezultat koji se vraća jest 2.
7. Rekurzivno se vraća vrijednost parametra x, tj. 3 iz tog koraka rekurzivne funkcije pomnožen s povratnom vrijednošću prethodnog koraka, tj. vrijednošću 2. Rezultat koji se vraća jest 6.
8. Rekurzivno se vraća vrijednost parametra x, tj. 4 iz tog koraka rekurzivne funkcije pomnožen s povratnom vrijednošću prethodnog koraka, tj. vrijednošću 6. Rezultat koji se vraća jest 24.

***Stog***

Stog je apstraktna struktura podataka kod koje se svaki novi element dodaje na vrh te strukture. Također, kada se neki element uzima sa stoga, uzima se onaj koji se nalazi na vrhu stoga, tj. element koji je zadnji stavljen na stog. Iz tog razloga ova struktura podataka zove se još i LIFO struktura (engl. Last In First Out).

Stog je detaljnije objašnjen u poglavlju 7.1.

**Rekurzivne funkcije ili iterativni način rješavanja problema**

Rekurzivne funckije moguće je primijeniti na mnogobrojnim primjerima, no potrebno je razmisliti je li to zaista potrebno (osim ako je rješavanje nekoga problema rekurzijom neophodno). Razlog ovoga promišljanja leži u tome da je rekurzivni poziv funkcije puno "skuplji" od iterativnog rješenja – brzina izvođenja i količina zauzete memorije, tim više ako će dani parametri uzrokovati preveliku dubinu gniježđenja. Funkcionalnost izračuna matematičke operacije faktorijeli, ali u slučaju kada ju napišemo iterativnim postupkom (na primjer, pomoću petlje for) puno je efikasnija od rekurzivnog pozivanja funkcije. Razlog u tome leži jer svaki poziv funkcije uključuje operacije kao što su: adresiranje, postavljanje argumenata i povratnih vrijednosti na stog. Struktura podataka stog koristi se za pohranjivanje rezultata i povratak iz rekurzije.

**Osnovni (bazni) slučaj**

***Stack overflow***

*Stack overflow* će se dogoditi ako rekurzivni poziv bude previše dubok ili pak beskonačan.

U tom slučaju prostor za spremanje varijabli i informacija koje su povezane sa svakim rekurzivnim pozivom više ne može stati na stog.

U slučaju da programer izostavi osnovni (bazni) slučaj kod rekurzivnih funkcija, funkcija bi sama sebe rekurzivno pozivala beskonačno puta, a kao posljedica toga u pozivajući dio programa nikada se ne bi vratila vrijednost koju bi takav rekurzivni poziv funkcije morao izračunati.

rez = faktorijel (4)

= 4 \* faktorijel (3)

= 3 \* faktorijel (2)

= 2 \* faktorijel (1)

= 1 \* faktorijel (0)

= 0 \* faktorijel (-1)

= -1 \* faktorijel (-2)

= -2 \* faktorijel (-3)

= -3 \* faktorijel (-4)

= -4 \* faktorijel (-5)

....

U programskom kôdu koji je prikazan u nastavku vidimo implementaciju funkcije faktorijel(x) koja je slična prethodno prikazanom programskom kôdu. Razlika između gornjeg i donjeg primjera je ta što u donjem primjeru ne postoji osnovni (bazni slučaj), koji bi prouzrokovao rekurzivno vraćanje natrag.

|  |
| --- |
| **def** faktorijel(x):  **return** x \* faktorijel(x - 1)   rez = faktorijel(4) print(rez) |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 4, in <module>  rez = faktorijel(4)  File "test.py", line 2, in faktorijel  return x \* faktorijel(x-1)  File "test.py", line 2, in faktorijel  return x \* faktorijel(x-1)  File "test.py", line 2, in faktorijel  return x \* faktorijel(x-1)  [Previous line repeated 990 more times]  RecursionError: maximum recursion depth exceeded |

* 1. Bezimene funkcije

U programskom jeziku *Python*, bezimena funkcija je funkcija koja je definirana bez imena. Bezimene funkcije (engl. *anonymous function*) se još nazivaju i lambda funkcije (engl. *lambda function*). Klasične funkcije definiraju se pozivom ključne riječi def, dok se bezimene funkcije definiraju korištenjem ključne riječi lambda. U kontekstu objektno orijentiranoga programiranja, bezimena funkcija je objekt.

Sintaksa kreiranja bezimene funkcije:

lambda [argumenti] : [izraz]

Bezimene funkcije mogu imati neograničenu količinu argumenata, svi argumenti se navode ispred dvotočke i međusobno su odijeljeni zarezom. Ovaj tip funkcije sintaksno je ograničen na samo jedan izraz. Takav izraz ima ograničenje da ne može sadržavati petlje: for, while, a niti pak uvjetne naredbe kao što je if. Ovo ograničenje moguće je izbjeći koristeći, na primjer, ternarni operator. Bezimena funkcija će na temelju argumenata izračunati rezultat izraza, te će u pozivajući dio programa uvijek vratiti izračunatu vrijednost (nije potrebno eksplicitno navoditi ključnu riječ return za vraćanje vrijednosti).

U nastavku slijedi primjer korištenja bezimene funkcije:

|  |
| --- |
| uvecajZa10 = **lambda** x: x + 10  print(uvecajZa10(5)) |
| Izlaz:  15 |

U gornjem primjeru bezimena funkcija je: lambda x : x + 10. Argument je varijabla x, dok je izraz bezimene funkcije: x + 10. Na temelju tog izraza izračunava se rezultat te se on vraća u pozivajući dio programa, tj. u gornjem primjeru ispisuje se na zaslon. Bezimene funkcije nemaju ime, no kreiranu bezimenu funkciju (lambda izraz) moguće je spremiti u varijablu, u gornjem primjeru u varijablu uvecajZa10. Ovako kreirana i spremljena bezimena funkcija poziva se na identičan način kao i klasične funkcije.

U nastavku slijedi primjer u kojem je funkcionalnost napisana u prethodnom programskom kôdu pomoću bezimene funkcije ostvarena implementacijom klasične funkcije:

|  |
| --- |
| **def** uvecaj(x):  **return** x + 10   print(uvecaj(5)) |
| Izlaz:  15 |

Primjer bezimene funkcije koja zbraja tri primljena argumenta:

|  |
| --- |
| zbroji = **lambda** a, b, c: a + b + c  print(zbroji(1, 2, 3)) |
| Izlaz:  6 |

**Složenije korištenje bezimenih funkcija**

Bezimene funkcije možemo koristiti i kod ugrađenih funkcija gdje se kao argument očekuje funkcija (bilo klasična bilo bezimena, tj. lambda funkcija). Ugrađene funkcije: filter(), map(), reduce() kao prvi argument očekuju upravo funkciju. U nastavku slijedi primjer s ugrađenom funkcijom filter() i slanjem bezimene funkcije kao prvi argument. Prototip funkcije filter() je:

filter(funkcija, iterable)

Kao prvi argument funkcija filter() očekuje ime klasične funkcije ili pak bezimenu funkciju koja mora vraćati jednu od dvije potencijalne vrijednosti, a to su *True* ili *False*, čiji se rezultat izračunava za svaki pojedini element iz objekta *iterable*. Kao drugi argument očekuje se lista, n-torka, skup ili bilo koja druga struktura po kojoj je moguće napraviti iteraciju. Funkcija filter() uklanja elemente s obzirom na zadani uvjet.

|  |
| --- |
| lista = [1, 3, 4, 7, 11, 55, 100, 102, 104] rez = list(filter(**lambda** e: (e % 2 == 0), lista)) print(rez) |
| Izlaz:  [4, 100, 102, 104] |

Znači, funkcija filter() iterira po primljenoj listi. Za svaku vrijednost pomoću bezimene funkcije, tj. lambda funkcije zaključuje se je li ta vrijednost parna ili neparna. Na temelju izraza bezimene funkcije: e%2 == 0 lambda izraz vraća vrijednost *True* ili *False*.

U nastavku je prikazan programski kôd s istom funkcionalnošću kao i prethodni primjer, no umjesto korištenja bezimene funkcije korištena je klasična funkcija. Unutar prikazanog programskog kôda korišten je poziv funkcije list(), toj funkciji predaje se rezultat funkcije filter() iz razloga što funkcija filter() vraća iterator.

|  |
| --- |
| **def** parnost(x):  **return** x % 2 == 0  lista = [1, 3, 4, 7, 11, 55, 100, 102, 104] rezultat = list(filter(parnost, lista)) print(rezultat) |
| Izlaz:  [4, 100, 102, 104] |

Klasična funkcija kao objekt može vratiti bezimenu funkciju. Takav primjer programskoga kôda nalazi se u nastavku. Funkcija prototipa multipliciranje(n) prima jedan argument n. Vrijednost parametra n se nakon toga ugrađuje u bezimenu, tj. lambda funkciju. Pretpostavimo da je vrijednost parametra n u nekom pozivu 3, tada će se preko povratne vrijednosti klasične funkcije multipliciranje(n) vratiti bezimena funkcija:

lambda a : a \* 3

U donjem programskom kôdu tako vraćena bezimena funkcija sprema se u objekt imena triPuta. Pozivom bezimene funkcije na način:

triPuta(3)

varijabla a u bezimenoj funkciji mijenja se s vrijednošću 3, te se izračunava rezultat izraza a\*3 i vraća se vrijednosti 9.

|  |
| --- |
| **def** multipliciranje(n):  **return lambda** a: a \* n  dvaPuta = multipliciranje(2) triPuta = multipliciranje(3) print(dvaPuta(3)) print(triPuta(3)) |
| Izlaz:  6  9 |

* 1. Ugniježđene funkcije

Ovo poglavlje bavi se ugniježđenim funkcijama (engl. *nested function* ili *inner function*). Ovaj koncept može se definirati kao funkcija unutar funkcije. Pravilo za stvaranje ovakvoga tipa funkcija identično je kao i za klasične funkcije (globalno dohvatljive funkcije) s jedinom razlikom da se ugniježđena funkcija nalazi unutar klasično kreirane funkcije (lokalni doseg globalno kreirane funkcije). Primjer ugniježđene funkcije prikazan je u nastavku.

|  |
| --- |
| **def** vanjska():  print(**"Vanjska funkcija."**)   **def** unutarnja():  print(**"Unutarnja funkcija."**)   vanjska() |
| Izlaz:  Vanjska funkcija. |

U gornjem primjeru možemo vidjeti implementaciju dviju funkcija vanjska() i unutarnja(). U glavnom programu poziva se funkcija vanjska() te se unutar te funkcije izvršava poziv ugrađene funkcije print(), unutar funkcije vanjska() implementirana je ugniježđena funkcija unutarnja(). Ta funkcija nigdje nije pozvana te se iz tog razloga nije ni izvršila. U nastavku je prikazan primjer programskoga kôda kod kojeg se poziva ugniježđena funkcija unutarnja().

|  |
| --- |
| **def** vanjska():  print(**"Vanjska funkcija."**)   **def** unutarnja():  print(**"Unutarnja funkcija."**)   unutarnja()   vanjska() |
| Izlaz:  Vanjska funkcija.  Unutarnja funkcija. |

**Poziv ugniježđene funkcije unutarnja() iz glavnog programa.**

Primijetimo da se u glavnom programu poziva funkcija unutarnja(). To je lokalna funkcija globalne funkcije vanjska() te je programski jezik *Python* iz tog razloga javio grešku da ona nije definirana. Vrlo je bitno imati na umu da se ugniježđena funkcija može dohvatiti samo unutar globalne (vanjske) funkcije.

|  |
| --- |
| **def** vanjska():  print(**"Vanjska funkcija."**)   **def** unutarnja():  print(**"Unutarnja funkcija."**)   unutarnja() |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 7, in <module>  unutarnja()  NameError: name 'unutarnja' is not defined |

**Primjer sa slanjem parametara ugniježđenoj funkciji**

|  |
| --- |
| **def** vanjska(x):  print(**"Vanjska funkcija, x ="**, x)   **def** unutarnja(y):  print(**"Unutarnja funkcija, y ="**, y)   unutarnja(x \* 2)   vanjska(10) |
| Izlaz:  Vanjska funkcija, x = 10  Unutarnja funkcija, y = 20 |

Ugniježđene funkcije najčešće se koriste kako bi se postigla enkapsulacija (engl. *encapsulation*), tj. skrivanje ponašanja nekoga dijela programskoga kôda od ostatka programa. Uzmimo za primjer gornji programski kôd. Implementacija funkcije unutarnja() vidi se samo iz funkcije vanjska(), što znači da je programski kôd napisan u funkciji unutarnja() sakriven od globalnog dosega.

**Memoizacija**

U nastavku se nalazi primjer programskoga kôda koji pomoću rekurzije ispisuje *Fibonacci* brojeve. Problem donjega primjera leži u činjenici da se s povećanjem ulaznog parametra n u funkciju fibonacci(n)povećava i broj koraka rekurzivnoga spusta koji računaju vrijednost koja se u nekom prethodnom koraku programskoga kôda već možda i izračunala.

|  |
| --- |
| **def** fibonacci(n):  **if** n <= 1:  **return** n  **else**:  **return** (fibonacci(n - 1) + fibonacci(n - 2))  print(**"Fibonacci sekvenca:"**) **for** i **in** range(9):  print(fibonacci(i)) |
| Izlaz:  Fibonacci sekvenca:  0  1  1  2  3  5  8  13  21 |

Tablica u nastavku prikazuje kako se za svaku pojedinu danu vrijednost funkcija fibonacci(n) rekurzivno poziva. Pregled poziva napravljen je na temelju gornjega programskog kôda u kojem se ispisuje prvih 9 *Fibonacci* brojeva. Radi preglednosti u tablici se umjesto imena fibonacci(n) koristi ime f(n). Zelenom bojom označene su vrijednosti koje se ″izračunavaju″ prvi put, dok su žutom bojom označeni pozivi izračuna koji su u nekom od prethodnih koraka već izračunati. Iz tablice je vidljivo da se najviše vremena izvođenja programskoga kôda troši na izračun vrijednosti koje su u nekom od prethodnih koraka već izračunate. Kako bi se spriječilo izračunavanje već izračunatih vrijednosti mogu se koristiti raznorazne programske strukture unutar kojih će se spremiti izračunate vrijednosti koje će se u budućnosti u slučaju potrebe koristiti. Takav postupak naziva se memoizacija.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pozivi funkcije fibonacci() | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| f(0) | f(1) | f(2) | f(3) | f(4) | f(5) | f(6) | f(7) | f(8) |
|  | f(0) | f(1) | f(2) | f(3) | f(4) | f(5) | f(6) | f(7) |
|  |  | f(0) | f(1) | f(2) | f(3) | f(4) | f(5) | f(6) |
|  |  |  | f(0) | f(1) | f(2) | f(3) | f(4) | f(5) |
|  |  |  |  | f(0) | f(1) | f(2) | f(3) | f(4) |
|  |  |  |  |  | f(0) | f(1) | f(2) | f(3) |
|  |  |  |  |  |  | f(0) | f(1) | f(2) |
|  |  |  |  |  |  |  | f(0) | f(1) |
|  |  |  |  |  |  |  |  | f(0) |

Memoizacija je tehnika optimizacije koja se koristi za ubrzavanje računalnih programa pohranjivanjem rezultata skupih poziva funkcija i vraćanjem rezultata iz predmemoriranih podataka kada se isti ulazi ponovno pojave. U ovom slučaju, memoizacija će ublažiti potrebu za ponovnim rješavanjem već riješenih *Fibonaccijevih* brojeva.

U nastavku slijedi primjer programskoga kôda kod kojeg je implementirana memoizacija. Unutar funkcije fib(n) implementirana je ugnježđena funkcija fibMemo(n, m). Unutar funkcije fib(n), kreiran je rječnik u kojem je pohranjena prva i druga vrijednost *Fibonacci* niza. Taj se riječnik šalje funkciji fibMemo(n, memo) koja pohranjene vrijednosti koristi za izračun ostalih *Fibonacci* brojeva čime se postiže optimizacija brzine izračuna. Svaki put kada se nova vrijednost *Fibonacci* niza izračuna, tako izračunata vrijednost stavlja se u riječnik te se koristi u sljedećem koraku rekurzije.

|  |
| --- |
| **def** fib(n):  **def** fibMemo(n, memo):  **if** n **in** memo:  **return** memo[n]   fib = fibMemo(n-1, memo)+fibMemo(n-2, memo)  memo[n] = fib  print(memo)  **return** fib   memo = {1: 1, 2: 1}  **return** fibMemo(n, memo)   print(fib(7)) print(**"---"**) print(fib(7)) |
| Izlaz:  {1: 1, 2: 1, 3: 2}  {1: 1, 2: 1, 3: 2, 4: 3}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}  13  ---  {1: 1, 2: 1, 3: 2}  {1: 1, 2: 1, 3: 2, 4: 3}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}  {1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}  13 |

* 1. Generatorske funkcije

Generatorske funkcije (engl. *generator function*) kao rezultat vraćaju generator, tj. objekt po kojem je moguće vršiti iteraciju. U nastavku slijedi usporedba normalnih funkcija i generatorskih funkcija.

|  |  |
| --- | --- |
| **Normalne funkcije** | **Generatorske funkcije** |
| koriste ključnu riječ return | koriste ključnu riječ yield |
| vraćaju povratnu vrijednost jednom | otpuštaju vrijednosti više puta |
| vraćaju vrijednost | vraćaju generator |

Generatorske funkcije su u načelu obične funkcije, no za razliku od normalnih funkcija koje za povratnu vrijednosti koriste ključnu riječ return, generatorske funkcije koriste ključnu riječ yield. Normalne funkcije neku izračunatu vrijednost vraćaju samo jednom, dok generatorske funkcije vraćaju generator objekt (što je zapravo pojednostavljeni iterator). Kroz tako vraćeni generator možemo iterirati po njegovim elementima. Ove funkcije moguće je pauzirati i nastavljati iz glavnog programa ovisno o potrebama (engl. *on the fly*). Generator (kao i iterator) predstavljaju pogled (engl. *view*) na kolekciju, tj. kolekcija se ne kopira već se samo pristupa njenim elementima.

U nastavku slijedi jednostavan primjer na temelju kojega će se objasniti ponašanje generatorskih funkcija.

|  |
| --- |
| **def** generatorska(x):  **while** x > 0:  **yield** x  x -= 1   g = generatorska(3)  print(next(g)) print(next(g)) print(next(g)) print(next(g)) |
| Izlaz:  3  2  1  Traceback (most recent call last):  File "test.py", line 11, in <module>  print( next(g) )  StopIteration |

U gore prikazanom programskom kôdu vidimo poziv funkcije generatorska(3), taj poziv nije izvršio tu funkciju već je u varijablu g vratio generator. Elemente generatora dobivamo pozivom funkcije next(). Kada prvi put pozovemo funkciju next(g), programski kôd implementirane generatorske funkcije krenut će se izvršavati do prvog pojavljivanja ključne riječi yield. Ključna riječ yield u gornjem primjeru vraća vrijednost varijable x. Nakon što je vrijednost varijable x vraćena, generatorska funkcija ne završava već se pamti cjelokupno stanje iste do sljedećeg poziva. Kod sljedećeg poziva generatorske funkcije, tj. naredbom next(g) programski kôd u funkciji će se nastaviti izvršavati od zadnjeg poziva naredbe yield, tako dugo dok se ponovo ne pojavi ključna riječ yield ili pak generatorska funkcija završi. Onoga trenutka kada izvođenje generatorske funkcije dođe do kraja ili pak do naredbe return, generator završetak iteracije signalizira pogreškom StopIteration. Ovu pogrešku možemo vidjeti kod ispisa greške četvrtoga poziva funkcije next(g). Ova pogreška dogodila se zato što je generator došao do kraja, tj. vrijednost varijable x je postala 0 te je while petlja unutar koje se poziva naredba yield završila.

U nastavku se nalazi primjer kôda u kojem se u varijablu g vratio generator, a po elementima toga generatora iterira se pomoću for petlje.

|  |
| --- |
| **def** generatorska(vrijednost):  i = 0  **while** i < vrijednost:  **yield** i  i += 1   g = generatorska(4) **for** e **in** g:  print(e) |
| Izlaz:  0  1  2  3 |

Sljedeći primjer prikazuje slučaj kod kojeg se u varijablu g vraća generator. Pomoću funkcije list() elementi danoga generatora konvertiraju se u listu.

|  |
| --- |
| **def** generatorska(vrijednost):  i = 0  **while** i < vrijednost:  **yield** i  i += 1   g = generatorska(4)  print(list(g)) |
| Izlaz:  [0, 1, 2, 3] |

**Napomena:**

Povratna vrijednost generatorske funkcije, a to je generator omogućava samo jedan prolaz kroz svoje elemente.

U nastavku je prikazan primjer kod kojeg se najprije putem petlje for iterira po svim elementima generatora, a nakon toga se ispisuje koliko elemenata ima u generatoru, jer je petlja for prošla po svim elementima generatora taj je rezultat 0.

|  |
| --- |
| **def** generatorska(vrijednost):  i = 0  **while** i < vrijednost:  **yield** i  i += 1   g = generatorska(4)  **for** e **in** g:  print(e)  duljinaNakonIteriranja = len(list(g)) print(**"Broj elemenata: "**, duljinaNakonIteriranja) |
| Izlaz:  0  1  2  3  Broj elemenata: 0 |

* 1. Vježba: Napredno korištenje funkcija

1. Napišite funkciju proizvoljnog imena koja prima proizvoljnu pozitivnu cjelobrojnu vrijednost. Rezultat je zbroj svih brojeva od 1 do primljene vrijednosti. Na primjer, ako funkcija primi vrijednost 5, u pozivajući dio programa mora se vratiti vrijednost 1+2+3+4+5 = *15*. Ovo je potrebno implementirati pomoću rekurzivne funkcije.
2. Napišite program koji će s tipkovnice učitati jednu proizvoljnu cjelobrojnu vrijednost. Za tako učitani cijeli broj prebrojite i ispišite koliko se parnih znamenki nalazi u njemu. Prebrojavanje parnih znamenki napravite pomoću rekurzivne funkcije. Na primjer, za broj 12345, potrebno je na zaslon ispisati vrijednost 2.
3. Napišite bezimenu funkciju koja prima 3 vrijednosti (pretpostavimo da su sve tri vrijednosti pozitivne i strogo veće od 0). Ta bezimena funkcija izračunava i vraća rezultat matematičke formule: (a\*b + c) / c. U glavnom programu ispišite rezultat matematičke formule za sljedeće vrijednosti: a = 5, b = 10; c = 2.
4. Nadogradite prethodni zadatak tako da prethodno implementirana bezimena funkcija može primiti 3 vrijednosti u rasponu od <-∞, ∞>. Ako izračun nije moguće provesti, ispišite poruku: "Err!", u suprotnom ispišite rezultat matematičke formule. Napomena: obradite slučaj dijeljenja s 0.
5. Isprobajte korištenje ugniježđenih funkcija. Vanjska funkcija imena izracun() neka prima dva parametra. Unutar vanjske funkcije kreirajte dvije unutarnje funkcije imena parniZbroj() i neparniZbroj(). Funkcija parniZbroj() neka vraća rezultat matematičke formule *2\*a+5\*b*, a funkcija neparniZbroj() neka vraća rezultat matematičke formule *a\*b-10*. Koja će se od dviju kreiranih unutarnjih funkcija pozvati neka se odredi na temelju parnosti zbroja primljenih parametara *a+b*. U glavnom programu na zaslon ispišite rezultat.
6. Implementirajte generatorsku funkciju imena danUTjednu(), funkcija mora vratiti generator s danima u tjednu. Nakon što svi dani u tjednu budu otpušteni, ne kreće se ponovo s otpuštanjem vrijednosti od ponedjeljka. U glavnom programu pozivom implementirane generatorske funkcije ispišite sve dane u tjednu.
7. Implementirajte generatorsku funkciju imena jedanDvaTri() koja vraća generator koji pak vraća vrijednosti: "Jedan", "Dva", "Tri", "Jedan", "Dva", "Tri", "Jedan", "Dva", "Tri" i tako u beskonačnost. U glavnom programu ispišite proizvoljan broj puta sekvencu: "Jedan", "Dva", "Tri".
8. \* Napišite program koji će od korisnika tražiti dvije vrijednosti. Prva vrijednost neka bude neki proizvoljan cijeli broj, a druga vrijednost neka bude znamenka od 0 do 9. Kreirajte rekurzivnu funkciju koja će odrediti broj pojavljivanja zadane znamenke u zadanom broju. Prototip funkcije:

prebroji(broj, znamenka)

1. \* Napišite program koji će s tipkovnice učitati jednu proizvoljnu cjelobrojnu vrijednosti. Za tako učitani broj, odredite pomoću rekurzivne funkcije je li on prost broj. Ako je učitani broj prost, ispišite na zaslon poruku: "Prost je!", u suprotnom ispišite "Nije prost!".
2. \*\* Pronađite na Internetu u službenoj *Python 3* dokumentaciji funkcionalnost i način korištenja ugrađene funkcije map().

Službenu *Python 3* dokumentaciju možete pronaći na URL-u: [https://docs.python.org/3/](https://docs.python.org/3).

Kreirajte listu od 5 proizvoljnih vrijednosti koje će predstavljati temperaturu izraženu u stupnjevima Celzijevim. Napišite bezimenu funkciju koja će uz pomoć funkcije map() generirati vrijednost temperature izražene u Farenhajtima. Formula: , x je vrijednost izražena u stupnjevima Celzijevim. Tako dobivenu listu ispišite na ekran.

1. \*\* Napišite generatorsku funkciju imena bottles(brojBoca, nazivNapitka). Ova funkcija mora vratiti generator koji otpušta stihove pjesme: "99 Bottles of Beer". Svaki stih ove pjesme ima za jedan napitak manje od prethodnog stiha, tako dugo dok se ne istroše svi napitci. Predefinirana vrijednost za broj boca je *99*, dok je predefinirana vrijednost za naziv napitka "*beer*". U nastavku slijedi primjer ispisa za dani parametar *brojBoca = 4*.

4 bottles of beer on the wall, 4 bottles of beer.

Take one down and pass it around, 3 bottles of beer on the wall.

3 bottles of beer on the wall, 3 bottles of beer.

Take one down and pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall, 2 bottles of beer.

Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.

Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.

Go to the store and buy some more, 99 bottles of beer on the wall.

* 1. Pitanja za ponavljanje: Napredno korištenje funkcija

1. Što su to rekurzivne funkcije?
2. Je li rekurzivni poziv funkcije efikasniji od iterativnog rješenja nekoga problema?
3. S obzirom na prethodno pitanje, zašto je tomu tako?
4. Što će se dogoditi ako je rekurzivna funkcija napisana tako da nema osnovnoga (baznog) slučaja?
5. Što je to bezimena funkcija?
6. Koji je drugi naziv za bezimenu funkciju?
7. Kako izgleda sintaksa bezimenih funkcija?
8. Koliko argumenata može primiti bezimena funkcija?
9. Je li bezimenu funkciju moguće spremiti u objekt (varijablu)?
10. Što su to ugniježđene funkcije?
11. Kada se ugniježđene funkcije najčešće koriste?
12. Koja je razlika između normalnih funkcija i generatorskih funkcija?
13. Iznimke

Po završetku ovoga poglavlja polaznik će moći:

obrađivati iznimke

podizati iznimke

istovremeno obrađivati više tipova iznimaka

prosljeđivati iznimke

koristiti else i finally programske blokove.

S povećanjem kompleksnosti programskoga kôda povećava se i vjerojatnost da se dogodi neki iznimni slučaj ili greška. Takvi iznimni slučajevi i greške koje se mogu dogoditi prilikom izvršavanja programa zovu se iznimke (engl. *exceptions*). U modernim programskim jezicima pa tako i u *Pythonu* iznimke je moguće obraditi na jednostavan način pomoću ugrađenih mehanizama čime se omogućava nesmetano izvršavanje programskoga kôda.

Greške koje se mogu dogoditi moguće je podijeliti u tri skupine grešaka:

* Sintaksne greške – ovaj tip greške događa se u slučaju kada programer napravi grešku koja nije u skladu s pravilima korištenoga programskog jezika, u našem slučaju *Pythona*. Zbog ovakvih se grešaka programski kôd ne može interpretirati i programeru se vraća informacija o grešci. Većinu sintaksnih grešaka relativno je lako ispraviti iz razloga što prilikom pokretanja programa interpreter vraća poziciju i opis greške.

|  |
| --- |
| prin(**"Test"**) |
| Izlaz:  Traceback (most recent call last):  File "C:/test.py", line 1, in <module>  prin("Test")  NameError: name 'prin' is not defined |

|  |
| --- |
| niz = **"abcde"** duljina = niz.len() print(duljina) |
| Izlaz:  Traceback (most recent call last):  File "C:/test.py", line 2, in <module>  duljina = niz.len()  AttributeError: 'str' object has no attribute 'len' |

* Semantičke greške – ovaj tip greške događa se kada je programski kôd napisan prema pravilima odabranoga programskog jezika te se uredno izvršava. Također, kod ovog tipa greške odabrana je ispravna logika, no program ne daje ispravan rezultat. Ovu grešku programer mora pronaći sam.

|  |
| --- |
| a = 10 b = **"abcd"** rez = a + b |
| Izlaz:  Traceback (most recent call last):  File "C:/test.py", line 4, in <module>  rez = a+b  TypeError: unsupported operand type(s) for +: 'int' and 'str' |

|  |
| --- |
| a += 1 |
| Izlaz:  Traceback (most recent call last):  File "C:/test.py", line 1, in <module>  a += 1  NameError: name 'a' is not defined |

* Logičke greške – kod ovog tipa greške programski kôd se uredno izvršava, no korištena logika za rješavanje nekoga problema nije ispravna te se dobivaju pogrešni rezultati. Ovu grešku programer mora pronaći sam.

|  |
| --- |
| a = 10  **if** a != 100:  print(**"Varijabla a je vrijednosti 100!"**) **else**:  print(**"Varijabla a nije vrijednosti 100!"**) |
| Izlaz:  Varijabla a je vrijednosti 100! |

Pogreške otkrivene tijekom izvršavanja programa nazivaju se iznimkama (engl. *exceptions*). Takve pogreške ne moraju nužno voditi do terminiranja izvršavanja programa, već ih je moguće obraditi o čemu će biti više riječi u ovom poglavlju.

U *Pythonu* sve iznimke nasljeđuju ugrađeni osnovni razred BaseException. BaseException je bazni razred za sve iznimke dok je Exception bazni razred za iznimke koje ne uzrokuju izlaz iz programa. Sve korisničke iznimke trebale bi nasljeđivati razred Exception.

U nastavku je prikazana hijerarhija iznimaka razreda BaseException. Kao što je iz hijerarhije moguće vidjeti, postoji mnogo različitih iznimaka koje su složene u smislene cjeline, čime se programerima omogućava da iznimke obrađuju na različite načine. Na primjer, podrazredi IndexError, KeyError pripadaju razredu LookupError. Ako se u programskom kôdu hvata iznimka IndexError, tada se neće uhvatiti iznimka KeyError, no ako se hvata iznimka LookupError tada će se uhvatiti bilo koja od dviju iznimaka koje pripadaju razredu LookupError.

|  |
| --- |
| BaseException  +-- SystemExit  +-- KeyboardInterrupt  +-- GeneratorExit  +-- Exception  +-- StopIteration  +-- StopAsyncIteration  +-- ArithmeticError  | +-- FloatingPointError  | +-- OverflowError  | +-- ZeroDivisionError  +-- AssertionError  +-- AttributeError  +-- BufferError  +-- EOFError  +-- ImportError  | +-- ModuleNotFoundError  +-- LookupError  | +-- IndexError  | +-- KeyError  +-- MemoryError  +-- NameError  | +-- UnboundLocalError  +-- OSError  | +-- BlockingIOError  | +-- ChildProcessError  | +-- ConnectionError  | | +-- BrokenPipeError  | | +-- ConnectionAbortedError  | | +-- ConnectionRefusedError  | | +-- ConnectionResetError  | +-- FileExistsError  | +-- FileNotFoundError  | +-- InterruptedError  | +-- IsADirectoryError  | +-- NotADirectoryError  | +-- PermissionError  | +-- ProcessLookupError  | +-- TimeoutError  +-- ReferenceError  +-- RuntimeError  | +-- NotImplementedError  | +-- RecursionError  +-- SyntaxError  | +-- IndentationError  | +-- TabError  +-- SystemError  +-- TypeError  +-- ValueError  | +-- UnicodeError  | +-- UnicodeDecodeError  | +-- UnicodeEncodeError  | +-- UnicodeTranslateError  +-- Warning  +-- DeprecationWarning  +-- PendingDeprecationWarning  +-- RuntimeWarning  +-- SyntaxWarning  +-- UserWarning  +-- FutureWarning  +-- ImportWarning  +-- UnicodeWarning  +-- BytesWarning  +-- ResourceWarning |

* 1. Primjeri iznimaka

U ovom poglavlju obrađene su iznimke koje se najčešće pojavljuju. Detaljniji opis ostalih iznimaka koje se mogu pojaviti moguće je pronaći u službenoj *Python 3* dokumentaciji. Također, treba napomenuti da nije dobra praksa hvatati određene tipove iznimaka poput: SyntaxError, NameError.

***SyntaxError***

Ovaj tip greške javlja se kada programer napiše programski kôd koji nije u skladu s pravilima (sintaksom) *Pythona*. Zbog ovog tipa pogreške interpreter ne zna kako interpretirati napisani programski kôd.

|  |
| --- |
| >>> return  SyntaxError: 'return' outside function  >>>  >>> None = 1  SyntaxError: can't assign to keyword  >>>  >>> 10 20  SyntaxError: invalid syntax  >>>  >>> iff x==0:    SyntaxError: invalid syntax  >>> |

*Python* nije 100 % interpreterski programski jezik već se svaka skripta prije izvršavanja kompajlira u bajt-kôd (engl. *bytecode*). Već prilikom toga procesa otkrivaju se sintaksne pogreške, što znači da je u trenutku izvršavanja programa skripta već prevedena i da nisu pronađene nikakve sintaktičke pogreške. Što opet znači da se iste ne mogu uhvatiti unutar skripte osim u nekoliko posebnih slučajeva. Tip greške SyntaxError moguće je uhvatiti ako se ona dogodi u programskom kôdu koji se izvršava pozivom funkcija: exec() ili eval() te uključivanjem modula u program pomoću import.

***NameError***

Ovaj tip greške javlja se kada korištena varijabla nije definirana, tj. kada joj nije pridružena neka konkretna vrijednost. Uz informaciju o tipu greške ispisuje se i ime varijable koja nije definirana.

|  |
| --- |
| >>> novaVarijabla  Traceback (most recent call last):  File "<pyshell#9>", line 1, in <module>  novaVarijabla  NameError: name 'novaVarijabla' is not defined  >>>  >>>  >>> print(novaVarijabla)  Traceback (most recent call last):  File "<pyshell#10>", line 1, in <module>  print(novaVarijabla)  NameError: name 'novaVarijabla' is not defined  >>>  >>> 10 + novaVarijabla  Traceback (most recent call last):  File "<pyshell#11>", line 1, in <module>  10 + novaVarijabla  NameError: name 'novaVarijabla' is not defined  >>> |

NameError je jedna od iznimaka koje se u praksi ne hvataju često već je potrebno napisati programski kôd na način da se ova iznimka ne dogodi.

***TypeError***

Ovaj tip greške javlja se kada operaciju ili funkciju primijenjujemo nad varijablom pogrešnoga tipa podataka. Uz informaciju o tipu greške ispisuje se i njen opis.

|  |
| --- |
| >>> 10 + "niz znakova"  Traceback (most recent call last):  File "<pyshell#12>", line 1, in <module>  10 + "niz znakova"  TypeError: unsupported operand type(s) for +: 'int' and 'str'  >>>  >>> len(5)  Traceback (most recent call last):  File "<pyshell#13>", line 1, in <module>  len(5)  TypeError: object of type 'int' has no len() |

***IndexError***

Ovaj tip greške javlja se kada operacija ili funkcija prima argument iz, na primjer, liste, niza znakova i slično, a vrijednost na željenom indeksu nije definirana (nepostojeći indeks). Uz informaciju o tipu greške ispisuje se i njen opis.

|  |
| --- |
| >>> lista = [1, 2, 3, 4]  >>> lista[1]  2  >>> lista[4]  Traceback (most recent call last):  File "<pyshell#23>", line 1, in <module>  lista[4]  IndexError: list index out of range  >>>  >>> nizZnakova = "SRCE"  >>> nizZnakova[4]  Traceback (most recent call last):  File "<pyshell#25>", line 1, in <module>  nizZnakova[4]  IndexError: string index out of range  >>> |

***ValueError***

Ovaj tip greške javlja se kada operacija ili funkcija prima argument ispravnoga tipa no neispravne vrijednosti. Uz informaciju o tipu greške ispisuje se i njen opis.

|  |
| --- |
| >>> int("55")  55  >>> int("test")  Traceback (most recent call last):  File "<pyshell#34>", line 1, in <module>  int("test")  ValueError: invalid literal for int() with base 10: 'test'  >>>  >>> a, b = [1, 2, 3]  Traceback (most recent call last):  File "<pyshell#37>", line 1, in <module>  a, b = [1, 2, 3]  ValueError: too many values to unpack (expected 2)  >>> |

***KeyError***

Ovaj tip greške javlja se kada se u rječniku ne nalazi traženi ključ. Uz informaciju o tipu greške ispisuje se naziv traženoga ključa koji ne postoji u rječniku.

|  |
| --- |
| >>> rjecnik = {"jedan" : "one", "dva" : "two"}  >>> rjecnik["jedan"]  'one'  >>> rjecnik["jeeeedan"]  Traceback (most recent call last):  File "<pyshell#42>", line 1, in <module>  rjecnik["jeeeedan"]  KeyError: 'jeeeedan'  >>> |

***AttributeError***

Ovaj tip greške javlja se kada se nad nekim objektom poziva atribut koji ne postoji. U donjem primjeru nad objektom nizZnakova poziva se metoda velika(), no kako ta metoda nije implementirana i *Python* ju ne može pronaći, vraća se ovaj tip greške. Uz informaciju o tipu greške ispisuje se i njen opis.

|  |
| --- |
| >>> nizZnakova = "zagreb"  >>> nizZnakova.upper()  'ZAGREB'  >>> nizZnakova.velika()  Traceback (most recent call last):  File "<pyshell#48>", line 1, in <module>  nizZnakova.velika()  AttributeError: 'str' object has no attribute 'velika'  >>> |

* 1. Obrada iznimaka

U dosadašnjem dijelu ovoga poglavlja obrađene su najčešće iznimke koje se pojavljuju. No sve te iznimke blokiraju, tj. zaustavljaju daljnje izvršavanje programa te same po sebi, ako ih se ne obradi, nisu korisne. Iz tog razloga uvedena je naredba try … except koja omogućava hvatanje i obradu iznimaka (engl. *exception haldling*). Unutar tijela naredbe try, izvodi se neki programski kôd. Ako se unutar tijela naredbe try pojavi iznimka (u ovo je uključen i programski kôd koji se poziva iz ovog tijela), daljnje izvođenje tijela naredbe try se prekida te se kreće izvoditi tijelo naredbe except. Sintaksa:

try:  
 <tijeloTryNaredbe>  
except:  
 <tijeloExceptNaredbe>

U nastavku slijedi pokazni primjer kod kojeg je uzrokovana iznimka IndexError. Definirana je lista od 4 elementa, a unutar bloka try pokušava se ispisati vrijednost na indeksu 4 koja nije definirana. Iz tog razloga izvođenje tijela naredbe try se prekida, a pokreće se izvođenje tijela naredbe except. U slučaju da se programski kôd napisan unutar naredbe try izvršio bez ikakve podignute iznimke, tada se tijelo naredbe except preskače.

|  |
| --- |
| lista = [1, 2, 3, 4] **try**:  print(lista[4]) **except**:  print(**"Podignuta je iznimka!"**) |
| Izlaz:  Podignuta je iznimka! |

Kod ovakvog načina obrade iznimaka naredba except će pokupiti sve podignute iznimke te se sve one obrađuju na isti način. Što znači da se na isti način obrađuje iznimka IndexError i iznimka ValueError i svaka druga podignuta iznimka. Ako se želi napisati programski kôd koji selektivno obrađuje iznimke, sintaksa je sljedeća:

try:  
 <tijeloTryNaredbe>   
except <iznimka1>:  
 <tijeloExceptNaredbe1>  
except <iznimka2>:  
 <tijeloExceptNaredbe2>  
except:  
 <tijeloExceptNaredbe>

Kao što se može vidjeti iz gornjeg primjera sintakse, nakon ključne riječi except navodi se ime iznimke i ovakvim načinom možemo uhvatiti i obraditi neki zadatak na način specifičan za tu iznimku. Ako ne želimo selektivno navoditi sve potencijalne iznimke koje se mogu dogoditi, na kraju možemo napisati samo ključnu riječ except, koja će uhvatiti sve iznimke koje nisu selektivno navedene.

**VAŽNO**: Dobra praksa obrade iznimaka jest da se iznimke selektivno obrađuju, tj. da se ne koristi ključna riječ except koja hvata sve iznimke koje nisu selektivno navedene. Razlog tome je taj da u slučaju da u tijeku izvršavanja programa nešto pođe po zlu, tada program u tišini nastavlja daljnje izvođenje te je na taj način otežano pronalaženje problema koji se dogodio, a nije prikladno obrađen. Sveobuhvatni blok except obično se koristi u slučaju kada se iznimke žele prosljeđivati dalje.

U nastavku slijede dva primjera kod kojih se u try bloku podižu iznimke (žutom bojom označen je except blok koji je uhvatio iznimku).

***Primjer 1:***

|  |
| --- |
| **try**:  lista = [1, 2, 3, 4]  print(lista[4]) **except** ValueError:  print(**"Podigla se ValueError iznimka!"**) **except** IndexError:  print(**"Podigla se IndexError iznimka!"**) |
| Izlaz:  Podigla se IndexError iznimka! |

Kod prvog primjera podiže se iznimka IndexError, jer se pokušava dohvatiti indeks 4 koji ne postoji u varijabli lista.

***Primjer 2:***

|  |
| --- |
| **try**:  print(**"niz znakova"** + 10) **except** ValueError:  print(**"Podigla se ValueError iznimka!"**) **except** IndexError:  print(**"Podigla se IndexError iznimka!"**) **except**:  print(**"Nešto je pošlo po zlu!"**) |
| Izlaz:  Nešto je pošlo po zlu! |

Kod drugog primjera podiže se iznimka TypeError, no kako tu iznimku nismo selektivno obuhvatili, nju hvata zadnji sveobuhvatni blok except. Kao što je gore u tekstu navedeno, sveobuhvatni blok except obično se koristi da se iznimke prosljeđuju dalje. Ovo je primjer loše prakse. Ako se želi ova iznimka obraditi prema pravilima struke, tada bi primjer programskoga kôda izgledao ovako:

***Primjer 2:***

|  |
| --- |
| **try**:  print(**"niz znakova"** + 10) **except** ValueError:  print(**"Podigla se ValueError iznimka!"**)  **except** IndexError:  print(**"Podigla se IndexError iznimka!"**)  **except** TypeError:  print(**"Podigla se TypeError iznimka!"**) |
| Izlaz:  Podigla se TypeError iznimka! |

**Napomena – objekt iznimke**

**Objekt iznimke**

Objekt iznimke priprada razredu iznimke koja se dogodila, u ovom primjeru to je iznimka ZeroDivisionError.  
On u sebi sadrži poruku o grešci, stacktrace, itd. Pozivom naredbe print(e) objekt e implicitno se pretvara u string čime se dobiva poruka o grešci.

Također, moguće je pristupiti detaljima (opisu) iznimke. U nastavku je primjer sintakse.

try:  
 <tijeloTryNaredbe>   
except <iznimka1> as <objektIznimke>:  
 <tijeloExceptNaredbe1>  
except:  
 <tijeloExceptNaredbe>

Sljedeći primjer prikazuje ispis detalja iznimke iz objekta imena e. Ovaj objekt je opcionalan, no ako ga ne navedemo tada opisu iznimke nije moguće pristupati. Ovi podaci nam omogućavaju da na ekran (ili negdje drugdje, na primjer u datoteku) zapišemo što se točno dogodilo i time omogućimo lakši pronalazak i ispravljanje greške.

|  |
| --- |
| **try**:  print(1/0) **except** ZeroDivisionError **as** e:  print(e) |
| Izlaz:  division by zero |

* 1. Podizanje iznimaka

Do ovog poglavlja sve iznimke podizane su od strane *Pythona*. No ne mora uvijek *Python* biti taj koji će podići iznimku, već iznimku može podići (engl. *raise*) i sam programer. Iznimke se od strane programera podižu kada on sam uvidi da će se za neke ulazne vrijednosti u nastavku programskoga kôda potencijalno dogoditi greška. Za podizanje iznimaka koristi se ključna riječ raise. Sintaksom prikazanom u nastavku kreira se objekt koji je tipa ImeIznimke. Argument <opcionalniArgument> predstavlja poruku greške i ona je tipa niz znakova, tj. String.

raise ImeIznimke(<opcionalniArgument>)

U nastavku slijedi primjer programskoga kôda kod kojeg se podiže iznimka ValueError bez opcionalnog argumenta.

|  |
| --- |
| >>> raise ValueError *#ili 'raise ValueError()'*  Traceback (most recent call last):  File "<pyshell#51>", line 1, in <module>  raise ValueError  ValueError  >>> |

U drugom primjeru programskoga kôda također se podiže iznimka imena ValueError, no ovdje se prenosi i opcionalni argument koji detaljnije može opisivati neki događaj. U oba slučaja kreira se objekt tipa ValueError.

|  |
| --- |
| >>> raise ValueError("Ručno se podigla iznimka!")  Traceback (most recent call last):  File "<pyshell#52>", line 1, in <module>  raise ValueError("Ručno se podigla iznimka!")  ValueError: Ručno se podigla iznimka!  >>> |

U nastavku slijedi primjer u kojem se podiže i hvata ručno bačena iznimka.

|  |
| --- |
| **try**:  **raise** ValueError(**"Iznimka je podignuta!"**) **except** ValueError **as** e:  print(e) |
| Izlaz:  Iznimka je podignuta! |

* 1. Istovremena obrada više tipova iznimaka

U dosadašnjem dijelu teksta ovoga priručnika, kod obrade iznimaka one su se hvatale u svakom except bloku, jedna po jedna. No nekad je više različitih tipova iznimaka potrebno obraditi na identičan način. U *Pythonu* je to omogućeno tako da se u zaglavlje bloka except navede   
n-torka unutar koje su popisane sve iznimke koje se žele hvatati unutar nekog except bloka. U nastavku je prikazan programski kôd u kojem se najlakše mogu prouzročiti dvije iznimke, iznimka ValueError i iznimka ZeroDivisionError. U donjem primjeru, iznimke ValueError moguće je prouzrokovati tako da se preko tipkovnice unese neki niz znakova unutar kojeg se nalazi znak koji nije broj. Iznimka ZeroDivisionError prouzrokuje se tako da se za vrijednost varijable nazivnik unese 0. U zaglavlju except bloka moguće je vidjeti n-torku unutar koje su navedene obje iznimke i kao takve obrađuju se s istim programskim kôdom.

|  |
| --- |
| **try**:  brojnik = int(input(**"Brojnik: "**))  nazivnik = int(input(**"Nazivnik: "**))  rezultat = brojnik / nazivnik  print(rezultat) **except** (ValueError, ZeroDivisionError) **as** e:  print(**"Dogodila se iznimka, opis:"**, e) **except**:  print(**"Ostale iznimke!"**) |
| Izlaz 1:  Brojnik: ***broj***  Dogodila se iznimka, opis: invalid literal for int() with base 10: 'broj'  ***Ctrl+C***  Kombinacijom tipki Ctrl+C terminira se izvršavanje programa. Ovu kombinaciju tipki moguće je koristiti i kada se želi prekinuti, na primjer, beskonačna petlja. |
| Izlaz 2:  Brojnik: ***10***  Nazivnik: ***0***  Dogodila se iznimka, opis: division by zero |
| Izlaz 3:  Brojnik: ***^C #(Ctrl+C)***  Ostale iznimke! |

* 1. Prosljeđivanje iznimaka

Ako se u pozivajući dio programa želi vratiti iznimka koja je uhvaćena unutar, na primjer, funkcije ili metode, to se može napraviti pomoću ključne riječi raise.

U funkciji kolicnik() koja je implementirana u programskom kôdu prikazanom u nastavku, prilikom dijeljenja s nulom *Python* podiže iznimku ZeroDivisionError. Iznimke koje će se dogoditi unutar funkcije kolicnik() hvataju se u sveobuhvatnom except bloku (blok koji hvata sve iznimke koje se dogode). Unutar except bloka u funkciji pomoću ključne riječi raise svaka iznimka koja se dogodi prosljeđuje se u glavni program. Tako proslijeđena iznimka podignuta u funkciji kolicnik() hvata se unutar glavnog programa. Poziv funkcije kolicnik() u glavnom programu nalazi se unutar try bloka, dok blok except hvata iznimku ZeroDivisionError. Sve ostale iznimke koje će se dogoditi bit će proslijeđene u pozivajući dio programa, no neće biti obrađene.

|  |
| --- |
| **def** kolicnik(a, b):  **try**:  rez = a / b  **return** rez  **except**:  **raise**   **try**:  brojnik = int(input(**"Brojnik: "**))  nazivnik = int(input(**"Nazivnik: "**))  rezultat = kolicnik(brojnik, nazivnik)  print(rezultat)  **except** ZeroDivisionError **as** e:  print(**"Dogodila se iznimka, opis:"**, e) |
| Izlaz 1:  Brojnik: ***1***  Nazivnik: ***0***  Dogodila se iznimka, opis: division by zero |
| Izlaz 2:  Brojnik: ***6***  Nazivnik: ***2***  3.0 |

U slučaju da se dogodi iznimka, *Python* će tražiti najbliži except blok. Ako se takav blok ne nađe (programer ga nije napisao), *Python* interpreter ima "sveobuhvatni" except blok koji će zastaviti program i ispisati poruku o grešci.

Uvijek se ulazi se u prvi except blok kod kojeg je uvjet zadovoljen. Na primjer, napisana su dva except bloka. Najprije je napisan blok: except ZeroDivisionError te je nakon njega napisan blok: except ArithmeticError. Prilikom dijeljenja s nulom oba uvjeta su zadovoljena, no kako je ZeroDivisionError napisan prvi, on će se i izvršiti.

* 1. Završne akcije: else i finally

**Blok else**

Blok else izvodi se u slučaju kada se unutar bloka try ne dogodi iznimka ili ako se on ne završi pozivom naredbe return, break ili continue.

**Blok finally**

Programski kôd koji se nalazi unutar bloka finally uvijek će se izvršavati prije završetka obrade iznimke (bilo da se iznimka dogodila ili ne). Blok finally obično se koristi za otpuštanje zauzetih resursa (na primjer, zatvaranje toka podataka prilikom rada s datotekama, ovaj primjer prikazan je na kraju ovoga potpoglavlja).

U programskom kôdu koji se nalazi u nastavku kod primjera 1 i 2, vidljivo je da se blok finally izvršava neovisno o tome je li se iznimka unutar funkcije kolicnik() podigla ili nije.

|  |
| --- |
| **def** kolicnik(a, b):  **try**:  rezultat = a / b  **except** ZeroDivisionError:  print(**"Dijeljenje s nulom!"**)  **else**:  print(**"Rezultat je:"**, rezultat)  **finally**:  print(**"Izvršavanje finally bloka!"**)   brojnik = int(input(**"Brojnik: "**)) nazivnik = int(input(**"Nazivnik: "**)) kolicnik(brojnik, nazivnik) |
| Izlaz 1:  Brojnik: ***10***  Nazivnik: ***2***  Rezultat je: 5.0  Izvršavanje finally bloka! |
| Izlaz 2:  Brojnik: ***10***  Nazivnik: ***0***  Dijeljenje s nulom!  Izvršavanje finally bloka! |
| Izlaz 3:  Brojnik: ***10***  Nazivnik: ***nazivnik***  Traceback (most recent call last):  File "test.py", line 13, in <module>  nazivnik = int(input("Nazivnik: "))  ValueError: invalid literal for int() with base 10: 'nazivnik' |

1. **primjer:**

U nastavku slijedi primjer iz prakse koji prikazuje korištenje ključne riječi with u korelaciji s try/finally programskim blokom.

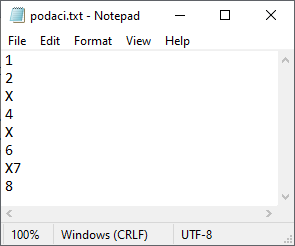
|  |
| --- |
| **with** open(**"datoteka.txt"**, **"wb"**) **as** f:  f.write(**"Hello Python!"**) |

Pomoću naredbe with, a nakon izvršavanja pripadajućega programskog kôda, osigurava se otpuštanje zauzetoga resursa, spremljenog u objektu f. Prikaz funkcionalnosti naredbe with, moguće je prikazati pomoću try/finally programskoga bloka. Unutar try bloka otvara se tok podataka prema datoteci te se izvršavaju željene akcije nad datotekom. Nakon što su sve akcije završene tada se uvijek izvršava finally blok kod kojeg se otpuštaju resursi koji su bili zauzeti.

|  |
| --- |
| **try**:  f = open(**"datoteka.txt"**, **"w"**)  f.write(**"Hello Python!"**) **except** IOError:  print(**"IOError!"**) **finally**:  f.close() |

1. **primjer:**

U nastavku slijedi primjer programa kod kojeg se čitaju brojevi (linija po linija) iz tekstualne datoteke te se kao rezultat dobiva suma tako pročitanih brojeva. Datoteka podaci.txt je sljedećeg sadržaja:



Programski kôd koji čita zapise iz tekstualne datoteke te izračunava sumu pročitanih brojeva izgleda ovako:

|  |
| --- |
| suma = 0  **with** open(**"podaci.txt"**, **"r"**) **as** f:  **for** linija **in** f.readlines():  broj = int(linija)  suma += broj  print(linija, end=**""**)  print(**"\n\nSuma: "**, suma) |
| Izlaz:  1  2  Traceback (most recent call last):  File "C:/test.py", line 5, in <module>  broj = int(linija)  ValueError: invalid literal for int() with base 10: 'X\n' |

Kao što je moguće vidjeti, došlo je do iznimke. Iznimka se podigla zato što se u datoteci nalaze zapisi koji unutar sebe ne sadrže samo brojeve već i neke ostale znakove. Prilikom pretvaranja takvih zapisa u broj korištenjem funkcije int() podiže se iznimka ValueError.

Kako bi se izračunala suma brojeva iz datoteke, neovisno sadržava li neki zapis samo brojeve ili pak i nedopuštene znakove (zapisi s nedopuštenim znakovima se ignoriraju), potrebno je napraviti obradu iznimnog slučaja. U nastavku se nalazi primjer programskog kôda koji to omogućava.

|  |
| --- |
| suma = 0 brojGresaka = 0  **with** open(**"podaci.txt"**, **"r"**) **as** f:  **for** linija **in** f.readlines():  **try**:  broj = int(linija)  **except** ValueError:  brojGresaka += 1  **else**:  suma += broj  **finally**:  print(linija, end=**""**)  print(**"\n\nSuma: "**, suma) print(**"Broj grešaka: "**, brojGresaka) |
| Izlaz:  1  2  X  4  X  6  X7  8  Suma: 21  Broj grešaka: 3 |

* 1. Vježba: Iznimke

1. Napišite programski kôd unutar kojeg postoji vjerojatnost da se podigne iznimka ValueError. Pokrenite program i namjestite ulazne vrijednosti koje će podići iznimku ValueError.
2. Programski kôd iz prethodnog zadatka nadogradite tako da uhvatite iznimku koja se dogodila (napravite da se hvataju sve iznimke, a ne selektivno) te na ekran ispišite poruku "Iznimka je uhvaćena!".
3. Napišite proizvoljni programski kôd unutar kojeg se mogu dogoditi bar dvije vrste iznimaka. Selektivno uhvatite iznimke i ispišite koji tip iznimke je uhvaćen.
4. Nadogradite prethodni zadatak tako da uz ispis tipa uhvaćene iznimke ispišete i detalje (opis) iznimke generirane od strane *Pythona*.
5. Napišite funkciju imena ucitajPaZbroji(), koja učitava 3 broja s tipkovnice i ispisuje njihov zbroj. U toj funkciji uhvatite potencijalne iznimke te tako uhvaćene iznimke proslijedite u glavni dio programa (dio programa unutar kojeg je funkcija pozvana) te tamo ispišite da se dogodila iznimka.
6. Prethodni zadatak nadogradite tako da rezultat zbroja ispisujete unutar else bloka funkcije imena ucitajPaZbroji(). Također, napišite i blok finally koji će ispisivati generičku poruku poput: "Finally blok!".
   1. Pitanja za ponavljanje: Iznimke
7. U koja tri skupa je moguće podijeliti pogreške unutar programskog kôda?
8. Koji razred nasljeđuju sve iznimke u *Pythonu*?
9. Koji se tip iznimke podiže u slučaju da korištena varijabla nije definirana?
10. Koje dvije ključne riječi tvore osnovni blok za obradu iznimaka?
11. Koje dvije dodatne ključne riječi tvore prošireni blok iznimaka?
12. Za što služi except blok?
13. Kada se izvodi else blok?
14. Kada se izvodi finally blok?
15. Objektno orijentirano programiranje

Po završetku ovoga poglavlja polaznik će moći:

kreirati razrede i objekte

razlikovati metode:

* + specijalne metode
  + metode objekata
  + statičke metode

definirati vlasništvo i vidljivost varijable

shvatiti kako funkcionira vidljivost varijable: public, protected i private

koristiti svojstva

objasniti i upotrijebiti:

* + nasljeđivanje
  + preopterećenje
  + nadjačavanje u objektno orijentiranom programiranju.

Način razvoja programskoga kôda koji je prikazan u dosadašnjem dijelu tečaja naziva se proceduralno programiranje. To je način programiranja čija se logika zasniva na temelju funkcija, tj. blokova programskoga kôda.

Izrada programskoga kôda koja će biti objašnjena u ovom poglavlju zove se objektno orijentirano programiranje (kraće OOP). Većinu programa moguće je kvalitetno napisati pomoću proceduralnoga načina razvoja programskoga kôda, no kod velikih projekata preporučljivo je koristiti objektno orijentirano programiranje. Objektno orijentirano programiranje služi smanjenju kompleksnosti razvoja i održavanja programskih rješenja. OOP ne sprječava programere da pišu loš programski kôd već ih usmjerava da razvijaju programski kôd u okvirima standarda ove paradigme.

Kod objektno orijentiranoga programiranja, ideja je da problem koji se rješava bude razdijeljen na što manje logičke cjeline. Na primjer, ako program treba obrađivati podatke o osobama, tada je osnovni tip objekta u tom programu **razred** Osoba. Svakoj osobi možemo pridružiti razne parametre ili pak u duhu objektno orijentiranoga pristupa **atribute**. Atributi mogu biti, na primjer, ime, prezime, visina, težina, OIB. Također, svakom razredu (instanca razreda naziva se objekt) pridružene su i pripadajuće akcije, tj. **metode**. Na primjer, osoba može hodati, sjesti, trčati itd., pa tako imamo metode: hodaj(), sjedni(), trci(), koje objektu Osoba daju "život". Niže se nalazi ilustrativni prikaz razreda Osoba.

|  |  |
| --- | --- |
| Ime razreda: | Osoba |
| Atributi | ime |
| prezime |
| visina |
| tezina |
| ... |
| Metode: | hodaj() |
| sjedni() |
| trci() |
| ... |

Temeljni pojmovi (detaljnije su objašnjeni u narednim poglavljima) objektno orijentiranoga programiranja su:

* razred (engl. *class*)
* objekt (engl. *object*)
* nasljeđivanje (engl. *inheritance*)
* enkapsulacija ili učahurivanje (engl. *encapsulation*)
* apstraktcija (engl. *abstraction*)
* polimorfizam ili višeobličje (engl. *polymorphism*).
  1. Razredi i objekti

U nastavku slijedi tablica sinonima koji se često pojavljuju prilikom čitanja literature, a imaju isto značenje.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hrvatski nazivi (sinonimi)** | | | **Engleski nazivi** |
| Razred | Klasa | - | engl. (*class*) |
| Atribut | Svojstvo | - | engl. (*attribute*) |
| Objekt | Instanca | Jedinka | engl. (*instance*) |

**Razred** je osnovna programska cjelina kod objektno orijentiranoga načina programiranja. Unutar razreda zapisan je shematski plan (engl. *blueprint*) te se na temelju njega kreiraju **objekti**. Razred sadrži **atribute** i **metode**. Atributi opisuju objekt raznim podacima, svaki objekt ima kopiju svih atributa. Metode su ustvari funkcije, no one se pozivaju nad objektom kojem je neka metoda pridružena. Pridruživanje metode instanci objekta radi se pomoću parametra self. Parametar self je objašnjen u nastavku tečaja.

U nastavku se nalazi osnovni primjer kreiranja razreda.

|  |
| --- |
| **class** MojRazred:  *# Varijable* **def** imeMetode(self):  *# Implementacija metode* |

Svaki razred započinje ključnom riječju class, a nakon ključne riječi navodi se proizvoljno ime razreda. Prema *Python* preporuci, riječi naziva razreda pišu se velikim početnim slovom, na primjer: Osoba, KoordinatniSustav i slično. Unutar tako kreiranog razreda (klase) pišu se:

* atributi – svojstva razreda
* metode – funkcionalnosti razreda.

U nastavku je prikazan programski kôd koji ostvaruje neke funkcionalnosti od samih ljudi. Unutar razreda imena Osoba implementirane su tri metode hodaj(), sjedni(), trci(). U glavnom programu kreiran je jedan objekt (instanca) razreda Osoba imena o. Iz donjeg primjera moguće je vidjeti da se objekt nekoga razreda može kreirati na sljedeći način:

imeObjekta = ImeRazreda()

Metode nekog objekta pozivaju se pomoću sljedeće sintakse:

imeObjekta.imeMetode()

*Napomena: unutar donjeg primjera vidljivo je da se koristi parametar self, taj parametar bit će detaljnije objašnjen u nastavku tečaja.*

|  |
| --- |
| **class** Osoba:  **def** hodaj(self):  print(**"Hodaj!"**)   **def** sjedni(self):  print(**"Sjedni!"**)   **def** trci(self):  print(**"Trci!"**)   o = Osoba()  o.hodaj() o.sjedni() o.trci() |
| Izlaz:  Hodaj!  Sjedni!  Trci! |

U slučaju da je potrebno kreirati "beskoristan" razred koji ne sadrži programski kôd, to je moguće postići na način da se u tijelo kreiranoga razreda napiše naredba pass.

U donjem primjeru vidljivo je da je razred imena Osoba, razred bez atributa i metoda. Unutar glavnog programa kreirana su tri objekta (instance) razreda Osoba, a to su: o1, o2, o3. Iz ispisa je moguće zaključiti da sva tri objekta imaju različitu memorijsku lokaciju, što znači da su sva tri objekta međusobno neovisna.

|  |
| --- |
| **class** Osoba:  **pass** o1 = Osoba() o2 = Osoba() o3 = Osoba()  print(o1) print(o2) print(o3) |
| Izlaz:  <\_\_main\_\_.Osoba object at 0x030EEF30>  <\_\_main\_\_.Osoba object at 0x035567F0>  <\_\_main\_\_.Osoba object at 0x03556790> |

* 1. Vrste metoda

Unutar pojedinog razreda moguće je implementirati nekoliko tipova metoda, a to su:

* specijalne metode
* metode objekta (engl. *instance methods*)
* statičke metode (engl. *static methods*)
* metode razreda (engl. *class methods*) (*ne obrađuje se*).
  + 1. Specijalne metode

Iako postoji mnoštvo predefiniranih specijalnih metoda, na primjer: \_\_repr\_\_, \_\_str\_\_, \_\_format\_\_ i tako dalje, u nastavku će se obraditi samo jedna specijalna metoda, a to je metoda: \_\_init\_\_. Specijalne metode su predefinirane metode koje se nalaze unutar svakog razreda. U praksi ova metoda naziva se **konstruktor** (engl. *constructor*). Konstruktor se poziva prilikom instanciranja objekta i njegova je prvenstvena namjena da se atributi inicijaliziraju na željene vrijednosti prije početka korištenja kreiranog objekta.

Sintaksa kreiranja konstruktora **bez parametara** je:

def \_\_init\_\_(self):

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self):  print(**"Kreiran je objekt!"**)   o1 = Osoba() o2 = Osoba() |
| Izlaz:  Kreiran je objekt!  Kreiran je objekt! |

Kao što je vidljivo iz gornjeg primjera, u programu se kreiraju dva objekta: o1 i o2. Nigdje se ne poziva metoda \_\_init\_\_, no tijelo te metode se svejedno izvršava te se ispisuje niz znakova "Kreiran je objekt!". Razlog tog izvršavanja je jer se ta metoda, tj. konstruktor automatski pokreće prilikom kreiranja objekta razreda u kojem se on nalazi.

Sintaksa kreiranja konstruktora **s parametrima** je:

def \_\_init\_\_(self, [arg1, …]):

Konstruktor osim inicijalizacije objekta kreira i atribute te ih postavlja na željene vrijednosti. Kod drugih programskih jezika svi se atributi moraju definirati posebno izvan konstruktora, a unutar konstruktora se onda eventualno (ako programska logika to zahtjeva) postavljaju na inicijalne vrijednosti.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, vrijednost):  self.ime = vrijednost   o1 = Osoba(**"Marko"**) o2 = Osoba(**"Marija"**)  print(o1.ime) print(o2.ime) |
| Izlaz:  Marko  Marija |

U gornjem primjeru programskoga kôda vidljivo je da se prilikom instanciranja objekta, razredu Osoba prenosi ime, kod prvog poziva ime: "Marko", a kod drugog poziva ime: "Marija". Konstruktor će za svaki pojedini objekt (o1 i o2), atribut ime postaviti na predane vrijednosti.

* + 1. Metode objekta

Ako se metoda nekog razreda želi povezati s nekim objektom, to je moguće napraviti pomoću parametra self. Tako povezane metode zovu se metode objekata. Povezivanje metode objekta, uključujući i konstruktor, radi se na način da takva metoda kao prvi parametar prima referencu na objekt. Prema *Python* konvenciji za ime toga parametra uzima se riječ self, iako riječ self nije ključna riječ.

Sintaksa za definiranje takve metode je:

def imeMetode(self, [arg1, …])

Sintaksa za pristupanje atributima unutar takve metode je:

self.imeAtributa

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispisPodataka(self):  print(**"Podaci su: "**, self.ime, self.prezime)   **def** hodaj(self):  print(self.ime, **"hoda!"**)   o1 = Osoba(**"Ivan"**, **"Horvat"**) o2 = Osoba(**"Ana"**, **"Hrvat"**)  o1.ispisPodataka() o2.ispisPodataka()  o1.hodaj() o2.hodaj() |
| Izlaz:  Podaci su: Ivan Horvat  Podaci su: Ana Hrvat  Ivan hoda!  Ana hoda! |

U gornjem primjeru kreiran je razred imena Osoba. Kreirani razred sastoji se od konstruktora, dva atributa i dvije metode. U glavnom dijelu programa kreirana su dva objekta razreda Osoba. Prilikom kreiranja objekata o1 i o2 automatikom se poziva konstruktor koji atributima svakog objekta postavlja predane vrijednosti, ime i prezime. Nakon kreiranja objekata razreda, nad oba objekta pozvana je metoda ispisPodataka(). Ta metoda ispisuje vrijednosti argumenata ime i prezime pojedinog objekta. Sintaksa poziva metode nad objektom s dodatnim argumentima je:

imeObjekta.imeMetode([arg1, …])

Na kraju je pozvana metoda hodaj() koja ispisuje ime osobe koja hoda. Potrebno je obratiti pažnju na parametar self. Ako se taj parametar slučajno izostavi, program neće raditi ispravno te će se vratiti greška.

**Primjeri potencijalnih grešaka – parametar self**

**1. primjer** – izostavljen parametar self kod dohvaćanja vrijednosti atributa unutar metode objekta. U primjeru programskoga kôda izostavljeni kôd je precrtan i markiran žutom bojom.

Uvidom u ispis greške, može se primijetiti da *Python* ne može pronaći atribut naziva prezime. To se dogodilo zato što *Python* nije znao iz kojeg objekta uzeti vrijednost atributa prezime.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispisPodataka(self):  print(**"Podaci su: "**, self.ime, ~~self.~~prezime)  o = Osoba(**"Ivan"**, **"Horvat"**) o.ispisPodataka() |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 13, in <module>  o.ispisPodataka()  File "test.py", line 8, in ispisPodataka  print("Podaci su: ", self.ime, prezime)  NameError: name 'prezime' is not defined |

**2. primjer** – parametar self izostavljen je iz popisa parametara metode. U primjeru programskoga kôda izostavljeni kôd je precrtan i markiran žutom bojom.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispisPodataka(~~self~~):  print(**"Podaci su: "**, self.ime, self.prezime)  o = Osoba(**"Ivan"**, **"Horvat"**) o.ispisPodataka() |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 13, in <module>  o.ispisPodataka()  TypeError: ispisPodataka() takes 0 positional arguments but 1 was given |

Iz greške koju je *Python* vratio može se iščitati da metoda imena ispisPodataka() u svojoj implementaciji prima 0 parametara, ali jedan argument je predan. No, ako se pogleda poziv te metode koji se nalazi u glavnom programu, vidljivo je da u zagradama nema nikakve predane vrijednosti.

Kada se poziva metoda objekta na način:

imeObjekta.imeMetode([arg1, arg2, …])

*Python* će takav poziv automatski konvertirati u poziv oblika:

ImeRazreda.imeMetode(imeObjetka,[arg1, arg2, …])

Prototip metode objekta:

imeMetode(self, [arg1, arg2, …])

Ako se pogleda konvertiran poziv metode objekta i prototip metode objekta, može se primijetiti poveznica između njih preko parametra self.

* + 1. Statičke metode

Ako se želi omogućiti da se metoda unutar razreda može pozvati, a da poziv nije vezan za instanciran objekt, to je moguće postići pomoću statičkih metoda. Što znači da su statičke metode vezane za razred, a ne za objekt razreda. Za razliku od metoda objekata kod kojih se prilikom poziva potajno šalje prvi parametar self, kod statičkih metoda, taj se parametar ne šalje. Ovaj tip metode ne zna ništa o razredu ni o instanciranim objektima toga razreda (ne može pristupati varijablama ni ostalim metodama).

Sintaksa definiranja statičke metode:

def imeMetode([arg1, …])  
 pass

Sintaksa poziva statičke metode:

ImeRazreda.imeMetode([arg1, …])

Iz donjeg primjera programskoga kôda vidljivo je da unutar glavnog programa nigdje nije kreiran objekt razreda Osoba, no metoda imena punoljetnaOsoba() uspješno se poziva. Ono što je važno naglasiti jest da se kao prvi parametar statičke metode ne navodi parametar self, jer se u statičku metodu ne želi prenositi objekt razreda. Ako se navede parametar self tada to više neće biti statička metoda, nego metoda objekta.

|  |
| --- |
| **class** Osoba:   **def** punoljetnaOsoba(godina):  **return** godina >= 18   print(Osoba.punoljetnaOsoba(17)) print(Osoba.punoljetnaOsoba(18)) |
| Izlaz:  False  True |

U nastavku slijedi primjer kod kojeg se statička metoda poziva nad objektom razreda.

Iz ispisane pogreške moguće je vidjeti da se prilikom poziva statičke metode preko objekta razreda uz broj godina potajno prenosi i ime self. Kako bi se izbjegao ovaj problem, uveden je dekorator @staticmethod. Ovaj dekorator omogućava da se statička metoda poziva preko objekta razreda.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime):  self.ime = ime   **def** punoljetnaOsoba(godina):  **return** godina >= 18   o = Osoba(**"Marko"**) print(o.punoljetnaOsoba(17)) |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 10, in <module>  print(o.punoljetnaOsoba(17))  TypeError: punoljetnaOsoba() takes 1 positional argument but 2 were given |

Sintaksa definiranja statičke metode uz dekorator @staticmethod:

***Dekorator***

Dekoratori omogućavaju dodatne fukcionalnosti nad postojećim objektom bez izmjene strukture.

@staticmethod  
def imeMetode([arg1, …])  
 pass

Sintaksa poziva statičke metode uz dekorator @staticmethod:

1. način: ImeRazreda.imeMetode([arg1, …])  
2. način: imeObjekta.imeMetode([arg1, …])

Primjer korištenja statičke metode uz dekorator @staticmethod prikazan je u nastavku. Unutar glavnog programa donjega primjera najprije se kreira objekt imena o razreda Osoba. Nakon što je objekt kreiran, prikazana su dva načina kako je moguće pozvati statičku metodu. U ovom slučaju zbog korištenja dekoratora @staticmethod prilikom interpretiranja programskoga kôda ne izaziva se pogreška.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime):  self.ime = ime   @staticmethod  **def** punoljetnaOsoba(godina):  **return** godina >= 18   o = Osoba(**"Marko"**) print(o.punoljetnaOsoba(17)) print(Osoba.punoljetnaOsoba(18)) |
| Izlaz:  False  True |

* 1. Vlasništvo varijabli

Varijable kreirane u razredu pripadaju tom razredu i nije im moguće pristupati, a da se ne navede naziv objekta ili razreda, ovisno o implementaciji. Varijable unutar razreda možemo podijeliti na dvije vrste, ovisno o tome pripada li varijabla razredu ili pak objektu:

* varijable razreda
* varijable objekta.

**Varijable razreda** (engl. *class variables*) ili statičke varijable (engl. *static variables*) su dijeljene varijable kroz sve objekte nekoga razreda. Postoji samo jedna vrijednost ove varijable nevezano za to koliko je instanci kreirano što povlači za sobom činjenicu da ako se vrijednost promijeni u bilo kojoj instanci razreda (objektu), sve ostale instance razreda vidjet će promijenjenu vrijednost.

Sintaksa definiranja varijable razreda:

class ImeRazreda:  
 imeVarijable = <vrijednost>

Postavljanje i dohvaćanje vrijednosti unutar razreda:

Sintaksa za postavljanje vrijednosti je:

ImeRazreda.imeVarijable = <vrijednost>

Sintaksa za dohvaćanje vrijednosti je:

ImeRazreda.imeVarijable

Postavljanje i dohvaćanje vrijednosti iz glavnog programa:

Sintaksa za postavljanje vrijednosti je:

ImeRazreda.imeVarijable = <vrijednost>

imeObjekta.imeVarijable = <vrijednost>

Sintaksa za dohvaćanje vrijednosti je:

ImeRazreda.imeVarijable

imeObjekta.imeVarijable

**Varijable objekta** (engl. *instance variables*) ili ne-statičke varijable (engl. *non-static variables*) su varijable koje su u vlasništvu objekta razreda (instance). U ovom slučaju svaki objekt ima svoju varijablu, tj. ovakva varijabla nije dijeljena i ni na koji način povezana između različitih instanci jednog te istog razreda. Varijabla objekta vezana je za svaku pojedinu instancu. U programskom kôdu ta se veza ostvaruje korištenjem imena self, to ime označava vezu instance razreda (objekta) i same varijable.

Postavljanje i dohvaćanje vrijednosti unutar razreda:

Sintaksa za postavljanje vrijednosti je:

self.imeVarijable = <vrijednost>

Sintaksa za dohvaćanje vrijednosti je:

self.imeVarijable

Postavljanje i dohvaćanje vrijednosti iz glavnog programa:

Sintaksa za postavljanje vrijednosti je:

imeObjekta.imeVarijable = <vrijednost>

Sintaksa za dohvaćanje vrijednosti je:

imeObjekta.imeVarijable

U nastavku slijedi primjer u kojem je prikazano korištenje obaju tipova varijabli (varijable razreda i varijable objekta).

|  |
| --- |
| **class** Demo:  *# Varijabla razreda* broj = 0   **def** \_\_init\_\_(self, naziv):  *# Varijabla objekta* self.naziv = naziv  Demo.broj += 1   **def** ispis(self):  print(**"Kreirano instanci: "**, Demo.broj)  print(**"Naziv instance: "**, self.naziv)   d1 = Demo(**"PRVA"**) d2 = Demo(**"DRUGA"**) d3 = Demo(**"TRECA "**)  d1.ispis() d2.ispis() d3.ispis() |
| Izlaz:  Kreirano instanci: 3  Naziv instance: PRVA  Kreirano instanci: 3  Naziv instance: DRUGA  Kreirano instanci: 3  Naziv instance: TRECA |

U gore napisanom programskom kôdu nalaze se dvije varijable, a to su varijabla broj i varijabla naziv. Varijabla broj je varijabla razreda, dok je varijabla naziv varijabla objekta. Varijabla broj je definirana izvan svih metoda i njezina vrijednost je postavljena na 0. Vrijednost 0 bit će pridružena varijabli broj samo jednom, tj. prilikom prvoga kreiranja razreda Demo. Prilikom svakoga sljedećeg kreiranja razreda Demo pridruživanje vrijednosti 0 varijabli broj neće se izvršiti. Ovu varijablu možemo smatrati globalnom varijablom kroz sve instance razreda Demo. Ta globalna vrijednost izražena je prilikom ispisa sadržaja te varijable. Kao što se vidi iz priloženog, za svaku instancu razreda Demo, pozvana je metoda ispis() i kod svakog poziva ispisana je identična vrijednost, broj 3.

* 1. Vježba: Objektno orijentirano programiranje - I

1. Napišite jednostavni razred imena Vozilo. Tako kreirani razred neka bude potpuno prazan, bez atributa i metoda. U glavnom programu kreirajte tri objekta razreda Vozilo i pozivom funkcije print() ispišite podatke o tim trima objektima.
2. Nadogradite razred imena Vozilo iz prethodnog zadatka na način da unutar njega implementirate dvije metode. Metodu Vozi i metodu Koci. Metoda Vozi neka ispisuje na ekran poruku „Vozi!“, a metoda Koci neka ispisuje na ekran poruku „Koci!“. Iz glavnog programa pozovite tako napisane metode nad prethodno kreiranim objektima.
3. Nadogradite razred imena Vozilo iz prethodnog zadatka tako da mu pridružite tri atributa (pomoću konstruktora): naziv proizvođača, naziv modela, godište modela.
4. Nastavno na prethodni zadatak implementirajte metodu objekta imena ispis() koja ispisuje vrijednosti postavljenih atributa. Kreirajte dva objekta razreda Vozilo kojima ćete prilikom kreiranja pridružiti proizvoljne vrijednosti te pozivom metode ispis() ispišite pridružene vrijednosti.
5. Napišite objektno orijentirani program. Razred programa neka bude Student i neka ima sljedeće metode:
   * metoda koja omogućava dodavanje nove ocjene u listu ocjena (metoda objekta),
   * metoda koja vraća aritmetičku sredinu svih ocjena (metoda objekta),
   * metoda koja vraća, ovisno o zaprimljenom argumentu, broj takvih ocjena u listi ocjena (na primjer, ako je primljena vrijednost 5, metoda mora vratiti broj sakupljenih 5-tica), (metoda objekta),
   * metoda koja vraća ukupan broj dodijeljenih ocjena svim studentima zajedno (statička varijabla i statička metoda), ovaj zadatak napravite na dva načina, s dekoratorom @staticmethod i bez njega.

U glavnom programu kreirajte dva objekta razreda Student, pomoću metode za upis ocjena unesite minimalno 8 proizvoljnih ocjena te ispitajte ispravnost metode koja vraća aritmetičku sredinu ocjena, metode koja vraća koliko ocjena određene vrijednosti je student sakupio i zatim isprobajte metodu koja vraća koliko je ukupno ocjena dodijeljeno svim studentima zajedno.

* 1. Vidljivost varijable – public, protected, private

Za razliku od drugih programskih jezika kao što su *C++* ili *Java*, *Python* nema efektivan mehanizam koji bi mogao ograničiti programeru pristup varijablama koje se nalaze unutar razreda. No, ipak postoji konvencija pomoću koje se varijablama dodjeljuje jedno od triju svojstava: public, protected, private. Učinkovitost korištenja ove konvencije objašnjena je u nastavku.

Vidljivosti nam omogućavaju da se ostvari enkapsulacija ili učahurivanje (engl. *encapsulation*). Pomoću ovih triju svojstava (public, protected, private) moguće je ograničiti naovlašten pristup do metoda i varijabli. U narednom poglavlju imena *"Svojstva"* opisan je način kako se pristupa varijablama koje su označene kao privatne varijable nekoga razreda (pomoću postavljajućih i dobavljajućih metoda). Enkapsulacija je skrivanje atributa i ponašanja od ostalih razreda. Učahurivanjem se postiže slaba povezanost objekta čime objekti postaju neovisniji i interne promjene jednog objekta ne utječu na rad drugog objekta.

* + 1. Javne varijable (engl. *Public*)

Sve varijable u *Pythonu*, ako se ne navede drugačije, automatikom postaju javne, što znači da su vidljive i dohvatljive iz svih dijelova programskoga kôda.

U nastavku slijedi primjer programskoga kôda u kojem se prikazuje korištenje javnih varijabli.

|  |
| --- |
| **class** Demo:   **def** \_\_init\_\_(self, naziv):  self.naziv = naziv   **def** postavi(self, vrijednost):  self.naziv = vrijednost   **def** ispis(self):  print(self.naziv)  d = Demo(**"TEST"**)  d.naziv = **"Nova vrijednost"** d.ispis()  d.postavi(**"Novo"**) d.ispis() |
| Izlaz:  Nova vrijednost  Novo |

Gore prikazani programski kôd sastoji se od jedne varijable imena naziv. Korištena varijabla je javna, tj. dohvatljiva je iz svih dijelova programskoga kôda. Osim što je moguće dohvatiti njenu vrijednost, tu javnu varijablu moguće je i mijenjati iz svih dijelova programskoga kôda.

* + 1. Zaštićena varijabla (engl. *Protected*)

Ideja ovoga tipa varijable jest da je varijablama moguće pristupiti samo iz razreda i naslijeđenih razreda (naslijeđeni razredi su objašnjeni u nastavku tečaja). No, u *Pythonu* ne postoji mehanizam koji će zaista varijablu zamišljenu da bude tipa zaštićena, zaštititi od neovlaštenog pristupa, već se takvu varijablu samo označava zaštićenom. Ovdje se koristi ustaljena konvencija i programer sâm mora paziti da programski kôd implementira tako da se toj varijabli pristupa samo iz razreda u kojoj je definirana ili iz naslijeđenih razreda. Sintaksa definiranja takve varijable jest da se ispred imena varijable navede znak '\_'.

|  |
| --- |
| **class** Demo:   **def** \_\_init\_\_(self, naziv):  self.\_naziv = naziv   **def** postavi(self, vrijednost):  self.\_naziv = vrijednost   **def** ispis(self):  print(self.\_naziv)   d = Demo(**"TEST"**) d.\_naziv = **"Nova vrijednost"** d.ispis() d.postavi(**"Novo"**) d.ispis()  **VAŽNO!**  Vanjsko pristupanje privatnim i zaštićenim atributima loša je praksa koju valja izbjegavati! |
| Izlaz:  Nova vrijednost  Novo |

Kao što se može primijetiti, gornja dva primjera programskoga kôda napisana za zaštićenu varijablu i za javnu varijablu ne razlikuju se ni u čemu drugom već samo u imenovanju varijable: \_naziv. Kod primjera zaštićene varijable, iako je zabranjeno pristupati toj varijabli izvan razreda Demo, varijabli \_naziv i dalje je moguće pristupiti na način kao da je ona javna, što znači da programer sam mora brinuti da programski kôd bude napisan tako da se varijablama koje su zaštićene pristupa samo unutar razreda ili naslijeđenih razreda, jer *Python* ne brine o sigurnosti varijable.

* + 1. Privatna varijabla (engl. *Private*)

Ovome tipu varijable moguće je pristupiti samo unutar razreda u kojoj je ta varijabla definirana. Sintaksa označavanja privatne varijable je:

\_\_imeVarijable

Ispred imena varijable potrebno je dva puta napisati znak '\_'. U nastavku slijedi primjer programskoga kôda u kojem se iz glavnog dijela programa želi pristupiti privatnoj varijabli.

|  |
| --- |
| **class** Demo:  **def** \_\_init\_\_(self, naziv):  self.\_\_naziv = naziv   d = Demo(**"Test"**) print(d.\_\_naziv)  **Prividna "privatnost"**  Ostvarivanje privatnosti atributa u programskom jeziku *Python* ostvaruje se na način da *Python* jednostavno mijenja ime atributa i na taj način postiže prividnu "privatnost" koju je moguće zaobići.  **VAŽNO!**  Vanjsko pristupanje privatnim i zaštićenim atributima loša je praksa koju valja izbjegavati! |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 8, in <module>  print(d1.\_\_naziv)  AttributeError: 'Demo' object has no attribute '\_\_naziv' |

Kao što se iz gore prikazanoga primjera vidi, privatnoj varijabli naziva \_\_naziv nije moguće pristupiti iz glavnog dijela programa.

Ako je programeru potrebno pristupiti privatnoj varijabli, to je moguće napraviti na sljedeći način:

|  |
| --- |
| **class** Demo:  **def** \_\_init\_\_(self, naziv):  self.\_\_naziv = naziv   d = Demo(**"Test"**) print(d.\_Demo\_\_naziv) |
| Izlaz:  Test |

Varijabla \_\_naziv je i dalje privatna, no na gore prikazan način moguće joj je pristupiti i iz glavnog dijela programa. Sintaksa je:

imeObjekta.\_ImeRazreda\_\_imeVarijable

* 1. Svojstva

Iz glavnog dijela programskoga kôda moguće je pristupiti svim varijablama, bilo da su varijable javne, zaštićene ili pak privatne. Poanta zaštićenih i privatnih varijabli jest da im nije moguće pristupati, a ni mijenjati ih iz glavnog programa. U nastavku slijedi primjer koji zaštićenoj varijabli pristupa iz glavnog dijela programskoga kôda.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime):  self.\_ime = ime  o = Osoba(**"Ivica"**) print(o.\_ime) |
| Izlaz:  Ivica |

Ovakav način pristupanja varijabli imena \_ime funkcionira, no ta varijabla je zaštićena varijabla i njoj se ne bi smjelo pristupati iz glavnog programa. Jedan od načina kako ispraviti ovakvo pristupanje zaštićenim i privatnim varijablama jest pomoću dobavljajućih metoda (engl. *getter metod*) i postavljajućih metoda (engl. *setter metod*).

**VAŽNO!**

Glavni razlog za postojanje dobavljajućih i postavljajućih metoda jest taj da u slučaju promjene tipa atributa ili logike kojom se dobavljaju/postavljaju vrijednosti tu promjenu je potrebno napraviti samo na jednom mjestu (unutar dobavljajuće ili postavljajuće metode), a ne na svakom mjestu gdje se u programskom kôdu pristupa tom nekom atributu.

Ovaj način omogućava bolju kontrolu programskoga kôda i lakše upravljanje istim.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime):  self.\_ime = ime   *# Dobavljujuća metoda* **def** getIme(self):  **return** self.\_ime   *# Postavljujuća metoda* **def** setIme(self, ime):  self.\_ime = ime   o = Osoba(**"Ivica"**) print(o.getIme()) o.setIme(**"Perica"**) print(o.getIme()) |
| Izlaz:  Ivica  Perica |

Iz gornjeg primjera je vidljivo da metoda imena getIme() vraća vrijednost koja se nalazi u zaštićenoj varijabli \_ime, a metoda setIme() postavlja zaštićenu varijablu \_ime na vrijednost primljenoga parametra ime.

Programski jezik *Python* u svojoj implementaciji ima takozvana "svojstva" koja implementiraju funkcionalnost dobavljajućih i postavljajućih metoda.

**Dobavljajuća metoda – @property**

Dekorator @property ustvari implementira dobavljajuću metodu, tj. metodu koja vraća vrijednost pohranjenu u nekoj varijabli. Sintaksa kreiranja ovoga dekoratora je:

@property  
def imeMetode(self):

return self.\_imeVarijable

Sintaksa poziva je dobavljajuće metode (iako je ovo metoda, prilikom poziva dobavljajuće metode nije potrebno navoditi zagrade):

imeObjekta.imeMetode

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, ime):  self.\_ime = ime   @property  **def** ime(self):  **return** self.\_ime   o = Osoba(**"Ivica"**) print(o.ime) |
| Izlaz:  Ivica |

**Postavljajuća metoda – @imeMetode.setter**

Dekorator @imeMetode.setter ustvari implementira postavljajuću metodu, tj. metodu koja mijenja vrijednost neke varijable. Sintaksa kreiranja ovoga dekoratora je:

@imeMetode.setter  
def imeMetode(self, vrijednost):

self.\_imeVarijable = vrijednost

Sintaksa poziva je (iako je ovo metoda, prilikom poziva postavljajuće metode nije potrebno navoditi zagrade):

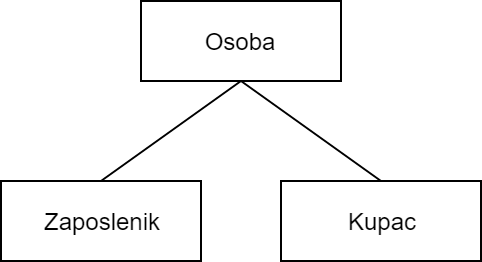
imeObjekta.imeMetode = <vrijednost>

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, ime):  self.\_ime = ime   @property  **def** ime(self):  **return** self.\_ime   @ime.setter  **def** ime(self, value):  self.\_ime = value   o = Osoba(**"Ivica"**) print(o.ime) o.ime = **"Perica"** print(o.ime) |
| Izlaz:  Ivica  Perica |

* 1. Vježba: Objektno orijentirano programiranje - II

1. Napišite razred imena Osoba te pomoću konstruktora tome razredu pridružite atribut imeOsobe. Tako kreirani atribut imeOsobe neka bude **javna** varijabla. U glavnom programu kreirajte jedan objekt razreda Osoba te iz njega ispišite vrijednost zapisanu u javnoj varijabli imeOsobe (bez korištenja dobavljajuće metode). Nakon što ste ispisali vrijednost spremljenu u varijabli imeOsobe, unutar glavnog programa promijenite tu vrijednost u neku drugu proizvoljnu vrijednost te ponovo ispišite vrijednost spremljenu u varijabli imeOsobe.
2. Prethodni zadatak prepravite na način da varijabla imeOsobe bude **zaštićena** varijabla. Unutar glavnog programa ispišite njenu vrijednost, nakon toga ju promijenite u neku drugu proizvoljnu vrijednost te ju ponovo ispišite. Postoji li razlika u ponašanju *Python* programa naspram 1. zadatka?
3. Prethodni zadatak prepravite na način da varijabla imeOsobe bude **privatna** varijabla. Unutar glavnog programa ispišite njenu vrijednost, nakon toga ju promijenite u neku drugu proizvoljnu vrijednost te ju ponovo ispišite. Postoji li razlika u ponašanju *Python* programa naspram 1. tj. 2. zadatka?
4. Prethodni zadatk prepravite na način da vrijednost **privatne** varijable imeOsobe ispišete unutar glavnog programa iako to „nije“ moguće (zaobilaznim putem).
5. Prethodni zadatak prepravite na način da za **privatnu** varijablu imeOsobe napišete postavljajuću i dobavljajuću metodu bez korištenja dekoratora. Unutar glavnog programa kreirajte jedan objekt razreda Osoba. Pomoću postavljajuće i dobavljajuće metode pristupite varijabli imeOsobe te najprije ispišite njenu vrijednost, zatim ju promijenite te ju opet ispišite.
6. Prethodni zadatak prepravite na način da ostvarite identičnu funkcionalnost, no ovog puta koristite dekoratore za postavljajuću i dobavljajuću metodu.
   1. Nasljeđivanje (engl. *Inheritance*)

Jedno od najvažnijih svojstava objektno orijentiranoga programiranja jest ponovna iskoristivost već napisanoga programskog kôda. Jedan od načina kojim se već napisani programski kôd može iskoristiti na efektivan način jest nasljeđivanje. Nasljeđivanje je osnovna funkcionalnost u objektno orijentiranim programskim jezicima. Nasljeđivanje omogućava da razred specifičnih funkcionalnosti može preuzeti svojstva osnovnoga razreda. Nasljeđivanje je najlakše moguće zamisliti kao vezu između glavnog razreda i njemu podređenoga razreda. Radi lakšeg razumijevanja, u nastavku se nalazi grafički prikaz nasljeđivanja.



Neke od prednosti nasljeđivanja jesu:

* Ako je potrebno promijeniti ili dodati neki novi atribut ili metodu kojom su zahvaćeni svi izvedeni razredi, umjesto da se taj atribut, tj. metoda dodaje na nekoliko mjesta u programskom kôdu (po svim izvedenim razredima), potrebno ga je dodati samo u bazni razred.
* Promjene u jednom izvedenom razredu ne utječu ni na jedan drugi izvedeni razred ili pak osnovni razred.
* Zbog mogućnosti korištenja programskoga kôda iz osnovnog razreda, nije potrebno taj isti kôd pisati (ili pak kopirati) još jednom kao što bi bilo potrebno ako bismo imali neovisne razrede.

**Primjer:**

U primjeru koji slijedi u nastavku, Osoba je **osnovni razred** (engl. *Base class*, *superclass*), dok su Zaposlenik i Kupac **izvedeni razredi** (engl. *Derived* *classes*, *subclasses*).

**Razred Osoba:** ovom razredu pridruženi su atributi ime i prezime te metode ispis() i hodaj(). Razredi koji opisuju zaposlenika i kupca imaju dodatne atribute i metode koji rade nešto specifično za njih, no istovremeno mogu pristupati i metodama iz osnovnog razreda.

**Razred Zaposlenik:** ovaj razred nasljeđuje razred Osoba. Uz atribute i metode iz razreda Osoba, ovom razredu dodijeljen je dodatni atribut placa i dodatne metode infoZaposlenik() i naplati(). Ovaj razred može pristupati metodama iz osnovnog razreda.

**Razred Kupac:** ovaj razred također nasljeđuje razred Osoba. Uz atribute i metode iz razreda Osoba, ovom razredu dodijeljen je dodatni atribut tipKupca i dodatna metoda infoKupac() i kupi(). Ovaj razred može pristupati metodama iz osnovnog razreda.

Sintaksa kojom izvedeni razred nasljeđuje bazni razred je:

class IzvedeniRazred(BazniRazred):

Izvedeni razred nadjačava bazni konstruktor pa ga je potrebno eksplicitno pozvati. Kod nekih drugih jezika to je često napravljeno "skriveno, automatski". Sintaksa za eksplicitno pizivanje baznoga konstruktora je:

BazniRazred.\_\_init\_\_(self, [param1], …)

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispis(self):  print(**"Ime:"**, self.ime)  print(**"Prezime:"**, self.prezime)   **def** hodaj(self):  print(**"Hodam!"**)   **class** Zaposlenik(Osoba):   **def** \_\_init\_\_(self, ime, prezime, placa):  Osoba.\_\_init\_\_(self, ime, prezime)  self.placa = placa   **def** infoZaposlenik(self):  print(**"Ime:"**, self.ime)  print(**"Prezime:"**, self.prezime)  print(**"Plaća:"**, self.placa)   **def** naplati(self):  print(**"Naplaćujem!"**)   **class** Kupac(Osoba):   **def** \_\_init\_\_(self, ime, prezime, tipKupca):  Osoba.\_\_init\_\_(self, ime, prezime)  self.tipKupca = tipKupca   **def** infoKupac(self):  print(**"Ime:"**, self.ime)  print(**"Prezime:"**, self.prezime)  print(**"Tip kupca:"**, self.tipKupca)   **def** kupi(self):  print(**"Kupujem!"**)   o = Osoba(**"Pero"**, **"Perić"**) z = Zaposlenik(**"Ivo"**, **"Ivić"**, 10000) k = Kupac(**"Lovro"**, **"Lovrić"**, **"Fizički kupac"**)  print(**">>> Klasa Osoba:"**) o.ispis() o.hodaj() print(**"\n>>> Klasa Zaposlenik:"**) z.infoZaposlenik() z.ispis() z.hodaj() z.naplati() print(**"\n>>> Klasa Kupac:"**) k.infoKupac() k.ispis() k.hodaj() k.kupi()  **Napomena**  Poziv baznoga konstruktora najbolje je staviti odmah na početak konstruktora izvedenoga razreda. |
| Izlaz:  >>> Klasa Osoba:  Ime: Pero  Prezime: Perić  Hodam!  >>> Klasa Zaposlenik:  Ime: Ivo  Prezime: Ivić  Plaća: 10000  Ime: Ivo  Prezime: Ivić  Hodam!  Naplaćujem!  >>> Klasa Kupac:  Ime: Lovro  Prezime: Lovrić  Tip kupca: Fizički kupac  Ime: Lovro  Prezime: Lovrić  Hodam!  Kupujem! |

**Python super()**

super() vraća objekt koji omogućava referenciranje na bazni razred. Također, omogućava rad s višestrukim nasljeđivanjem (višestruko nasljeđivanje nije obrađeno u ovom priručniku). U gore opisanom primjeru, prilikom nasljeđivanja baznoga razreda, sintaksa pomoću koje se unutar konstruktora izvedenoga razreda postavljaju atributi je:

**Napomena**

Nije dobro miješati super().\_\_init\_\_()  
i  
BazniRazred.\_\_init\_\_()  
tipove poziva jer mogu uzrokovati neočekivane pogreške.

def \_\_init\_\_(self, [param1], …)

BazniRazred.\_\_init\_\_(self, [param1], …)

Elegantniji način za postavljanje atributa jest korištenjem sljedeće sintakse:

def \_\_init\_\_(self, [param1], …)

super().\_\_init\_\_([param1], …)

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **class** Zaposlenik(Osoba):   **def** \_\_init\_\_(self, ime, prezime, placa):  super().\_\_init\_\_(ime, prezime)  self.placa = placa   **def** infoZaposlenik(self):  print(**"Ime:"**, self.ime)  print(**"Prezime:"**, self.prezime)  print(**"Plaća:"**, self.placa)   z = Zaposlenik(**"Pero"**, **"Perić"**, 10000) z.infoZaposlenik()  **Razlika: "is" i "=="**  *Python* ima dvije vrste operatora za uspoređivanje, a to su "is" i "==".  Operator "==" uspoređuje jednakost. Rezultat će biti True ako su vrijednosti dviju varijabli iste, nevažno kriju li se te dvije varijable iza istog ili iza različitih objekata.  Operator "is" uspoređuje identitete. Rezultat će biti True ako su vrijednosti dviju varijabli iste te ako obje varijable pokazuju na isti objekt.  *>>> a = [1, 2]*  *>>> b = a*  *>>> a == b*  *True*  *>>> a is b*  *True*  *>>> b = a.copy()*  *>>> a == b*  *True*  *>>> a is b*  *False*  *>>>*  *>>>*  *>>>*  *>>> a = 5*  *>>> b = 5*  *>>> a == b*  *True*  *>>> a is b*  *True*  *>>> b = a*  *>>> a == b*  *True*  *>>> a is b*  *True* |
| Izlaz:  Ime: Pero  Prezime: Perić  Plaća: 10000 |

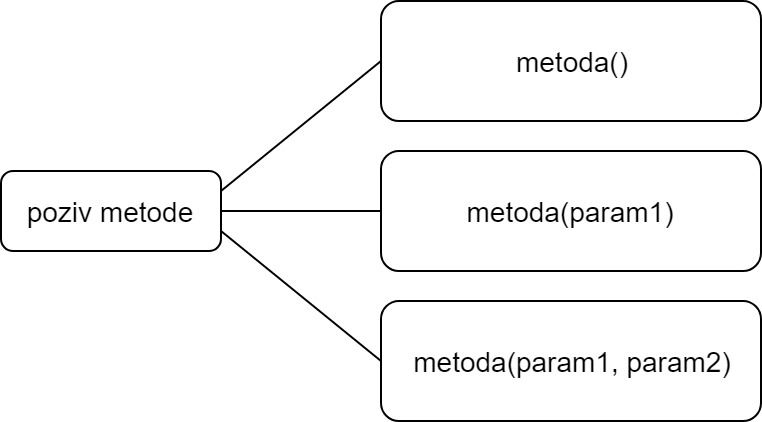
Prednost ovakvoga načina pisanja programskoga kôda jest ta da prilikom slanja atributa u bazni razred nije potrebno pisati parametar self. Također, kada se koristi super() nije potrebno navoditi ime baznoga razreda, čime se olakšava potencijalna promjena baznoga razreda. Promjenu baznoga razreda u tom slučaju potrebno je navesti samo na jednom mjestu, a ne na dvama.

* 1. Preopterećenje (engl. *Overloading*)

Kada dvije i više metoda imaju identično ime, a različit broj parametara, to se zove preopterećenje (engl. *Overloading*). Preopterećene metode se ustvari tretiraju kao različite metode. Ovdje se može primijetiti jedan od osnovnih pojmova OOP-a, a to je polimorfizam u kojem je ista metoda implementirana i djeluje na različite načine. U nastavku slijedi primjer klase s nekoliko preopterećenih metoda.

|  |
| --- |
| **class** Demo:   **def** metoda(self, param1=**None**, param2=**None**):   **if** param1 **is None and** param2 **is None**:  print(**"Metoda bez parametara!"**)   **elif** param2 **is None**:  print(**"Parametar:"**, param1)   **else**:  print(**"Parametri:"**, param1, **"i"**, param2)  d = Demo() d.metoda() d.metoda(**"A"**) d.metoda(**"A"**, **"B"**)  **Napomena**  Za razliku od nekih drugih programskih jezika, preopterećenje u *Pythonu* nije preopterećenje u punom smislu toga termina.  Preopterećenje u *Pythonu* nije sasvim fleksibilno jer koristi opcionalne parametre koji definiraju tijek izvršavanja programa.  Ako bi se preopterećenje u *Pythonu* napisalo kao, na primjer, u *Javi*, tako da se eksplicitno definiraju dvije metode s istim imenom i različitim potpisima, to bi rezultiralo korištenjem samo zadnje definirane metode. |
| Izlaz:  Metoda bez parametara!  Parametar: A  Parametri: A i B |

Preopterećenje se koristi kada se želi koristiti metoda jednog imena, no s mogućnošću prenošenja različitoga broja argumenata.



**Napomena - None**

None se koristi kako bi se nekoj varijabli pridružila null vrijednost ili nikakva vrijednost.

None je objekt razreda NoneType.

* None podržava operatore == i is.
* None nije jednako kao i False.
* None nije 0.
* None nije prazni niz znakova.
* Uspoređujući None sa bilo čime, vratit će se False, osim u slučaju ako je u objektu koji se uspoređuje sa None pohranjeno None.

Gore navedeni programski kôd ilustriran je gornjom slikom iz koje se vidi da imamo jednu metodu naziva metoda() i nju možemo pozvati na više različitih načina (u gornjem slučaju na 3 različita načina).

* Poziv metode bez argumenata:

Ako tu metodu pozovemo, a da joj ne prenesemo ijedan argument, tada će varijable param1 i param2 poprimiti preddefiniranu vrijednost, tj. None.

* Poziv metode s jednim argumentom:

U ovom slučaju, varijabla param1 će poprimiti vrijednost prenesenog argumenta, dok će varijabla param2 poprimiti preddefiniranu vrijednost, tj. None.

* Poziv metode s dvama argumentima:

U ovom slučaju, obje varijable param1 i param2 će poprimiti vrijednosti prenesenih argumenata.

* 1. Nadjačavanje (engl. *Overriding*)

Metoda istog imena u izvedenom razredu uvijek nadjačava metodu iz baznog razreda. Razlog za nadjačavanje metode baznoga razreda jest taj što se želi promijeniti funkcionalnost metode baznoga razreda. Također, i ovdje se primijenjuje jedan od osnovnih pojmova OOP-a, a to je polimorfizam u kojem je ista metoda implementirana i djeluje na različite načine u dvama različitim razredima.

U donjem programu kreiran je objekt imena z koji je instanca razreda Zaposlenik. Metoda imena ispis() implementirana je unutar razreda imena Osoba. Pozivom metode ispis() nad objektom imena z koji pripada razredu Zaposlenik, pokreće se metoda ispis() koju je razred Zaposlenik naslijedio iz bazne klase.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispis(self):  print(**"Ja sam:"**, self.ime, self.prezime)   **class** Zaposlenik(Osoba):   **def** \_\_init\_\_(self, ime, prezime, placa):  super().\_\_init\_\_(ime, prezime)  self.placa = placa   z = Zaposlenik(**"Pero"**, **"Perić"**, 10000) z.ispis() |
| Izlaz:  Ja sam: Pero Perić |

Ako se nad objektom imena z želi pozvati metoda identičnog imena kao metoda ispis() koja je implementirana unutar razreda Osoba, no drugačije funkcionalnosti, tada se primjenjuje princip nadjačavanja metode (engl. *Overriding*). Ovaj princip se primjenjuje tako da se unutar izvedenog razreda, u gornjem slučaju razreda Zaposlenik implementira metoda identičnog imena kao metoda koju se želi nadjačati, ali drugačije funkcionalnosti.

|  |
| --- |
| **class** Osoba:   **def** \_\_init\_\_(self, ime, prezime):  self.ime = ime  self.prezime = prezime   **def** ispis(self):  print(**"Ja sam:"**, self.ime, self.prezime)  **class** Zaposlenik(Osoba):   **def** \_\_init\_\_(self, ime, prezime, placa):  super().\_\_init\_\_(ime, prezime)  self.placa = placa   **def** ispis(self):  print(**"Ime:"**, self.ime)  print(**"Prezime:"**, self.prezime)  print(**"Plaća:"**, self.placa) z = Zaposlenik(**"Pero"**, **"Perić"**, 10000) z.ispis() |
| Izlaz:  Ime: Pero  Prezime: Perić  Plaća: 10000 |

* 1. Vježba: Objektno orijentirano programiranje - III

1. U ovom zadatku potrebno je simulirati nasljeđivanje. Kreirajte bazni razred imena Stavka i dva razreda koji nasljeđuju bazni razred, imena Sok i Hrana.  
     
   Bazni razred Stavka mora sadržavati konstruktor i metodu cijenaPdv(). Konstruktor prima dva parametra, naziv i cijenu bez PDV-a. Metoda cijenaPdv() mora vratiti cijenu s PDV-om (25%).  
     
   Razred Sok mora naslijediti bazni razred te mora sadržavati konstruktor i metodu ispis(). Konstruktor prima tri parametara, naziv, cijenu i volumen pića. Primijetite da se naziv i cijena nalaze u baznom konstruktoru. Metoda ispis() mora ispisati naziv, cijenu (bez PDV-a) i volumen pića.   
     
   Razred Hrana mora naslijediti bazni razred te mora sadržavati konstruktor i metodu ispis(). Konstruktor prima tri parametara, naziv, cijenu i broj kalorija hrane. Primijetite da se naziv i cijena nalaze u baznom konstruktoru. Metoda ispis() mora ispisati naziv, cijenu (bez PDV-a) i broj kalorija.  
     
   U glavnom programu kreirajte dva objekta, jedan razreda Sok, drugi razreda Hrana. Nad svakim objektom pozovite njegovu pripadajuću metodu ispis() te pomoću metode cijenaPdv() iz baznog razreda ispišite cijenu pića ili hrane sa PDV-om.
2. U ovom zadatku potrebno je simulirati nadjačavanje. Nadogradite prethodni zadatak ove vježbe na način da u izvedenim razredima implementirate metodu cijenaPdv() koja će nadjačati tu istu metodu iz baznog razreda. Tako implementirana metoda za piće mora vratiti cijenu s porezom od 15%, dok za hranu mora vratiti cijenu s porezom od 5%.
3. U ovom zadatku potrebno je simulirati preopterečenje metode. Kreirajte razred imena GeometrijskiLik, on neka implementira jednu metodu razreda imena opseg() koja može primiti 0 parametara ili pak maksimalno 3 parametara. Ovisno o broju primljenih parametera ova metoda ispisuje pripadajući rezultat.   
     
   0 parametara – Ispišite poruku o greški (Dogodila se greška!)  
   1 parametar – ispišite opseg kruga (2 \* r \* PI)  
   2 parametra – ispišite opseg pravokutinika (2\*a + 2\*b)  
   3 parametra – ispišite opseg raznostraničnog trokuta (a+b+c)  
     
   U glavnom programu pozovite tako implementiranu metodu razreda imena opseg() na sva 4 moguća načina.
   1. Apstraktni razred (engl. *Abtract Base Classes*)

Apstraktni razred (apstrakcija) se obično koristi u sljedećim slučajevima:

* Kada se želi spriječiti da se neki razred može instancirati (nekada nema smisla da se bazni razredi instanciraju, već se želi osigurati da se može instancirati samo izvedeni razred).
* Kada se želi osigurati da razredi koji nasljeđuju bazni razred implementiraju sve metode navedene u apstraktnom razredu identičnih prototipa (one metode koje su apstraktne, tj. bez implementacije).
* Kada se želi postići poveznica između izvedenih razreda (zajednička točka je bazna klasa).

U nastavku se nalazi primjer apstraktnoga razreda. Kako bi se neki razred mogao proglasiti apstraktnim razredom, potrebno je koristiti modul imena abc, to ime dolazi od engleskog izraza: *Abstract Base Class*. Sintaksa kojom se neki razred proglašava apstraktnim razredom je:

class ImeRazreda(ABC):

Neka metoda proglašava se apstraktnom navođenjem dekoratora @abstractmethod, primjer sintakse:

@abstractmethod

def imeMetode():  
 pass

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Oblik(ABC):   @abstractmethod  **def** brojStranica():  **pass** o = Oblik() |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 9, in <module>  o = Oblik()  TypeError: Can't instantiate abstract class Oblik with abstract methods brojStranica |

U ovom primjeru razred imena Oblik() nije mogao biti instanciran jer je on apstraktni razred te se iz tog razloga on ne može instancirati. Potrebno je napisati izvedeni razred koji će naslijediti ovaj apstraktni razred.

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Oblik(ABC):   @abstractmethod  **def** brojStranica():  **pass   class** Pravokutnik(Oblik):   **def** brojStranica(self):  print(**"Pravokutnik ima 4 stranice!"**)   **class** Trokut(Oblik):   **def** brojStranica(self):  print(**"Trokut ima 3 stranice!"**)   p = Pravokutnik() t = Trokut() p.brojStranica() t.brojStranica() |
| Izlaz:  Pravokutnik ima 4 stranice!  Trokut ima 3 stranice! |

Izvedeni razred mora implementirati sve apstraktne metode iz baznog razreda. U slučaju da unutar razreda Trokut ne bi bila implementirana metoda imena brojStranica() *Python* interpreter vratio bi sljedeću grešku: TypeError: Can't instantiate abstract class Trokut with abstract methods brojStranica.

**Primjer dinamičkoga polimorfizma objekata pri nasljeđivanju:**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** GeometrijskiLik(ABC):   @abstractmethod  **def** ime():  **pass** @abstractmethod  **def** opseg():  **pass**  @abstractmethod  **def** povrsina():  **pass**  **class** Kvadrat(GeometrijskiLik):   **def** \_\_init\_\_(self, a):  self.a = a   **def** ime(self):  **return "Kvadrat"   def** opseg(self):  **return** 4 \* self.a   **def** povrsina(self):  **return** self.a \* self.a   **class** Krug(GeometrijskiLik):   **def** \_\_init\_\_(self, r):  self.r = r   **def** ime(self):  **return "Krug"   def** opseg(self):  **return** self.r \* self.r \* 3.1415   **def** povrsina(self):  **return** 2 \* self.r \* 3.1415   **def** ispisi(lik):  print(lik.ime())  print(**"O ="**, lik.opseg())  print(**"P ="**, lik.povrsina())   likKvadrat = Kvadrat(5) likKrug = Krug(10)  ispisi(likKvadrat) ispisi(likKrug) |
| Izlaz:  Kvadrat  O = 20  P = 25  Krug  O = 314.15000000000003  P = 62.830000000000005 |

U ovom primjeru prikazan je dinamički polimorfizam objekata pri nasljeđivanju. Razred GeometrijskiLik je apstraktni razred s tri apstraktne metode: opseg(), povrsina(), ime(). Razredi Kvadrat i Krug nadjačavaju i implementiraju metode: opseg(), povrsina() i ime(). Metoda ispisi(lik) prima instancu razreda Kvadrat ili Krug te poziva pripadajuće metode za ispis opsega i površine.

* 1. Ulančavanje metoda

Ulančavanje metoda je tehnika koja se koristi za ulančano pozivanje više metoda nad istim objektom, koristeći referencu objekta samo jednom. Ulančavanje metoda postiže se na način da sve metode koje se žele ulančati moraju vratiti self.

U nastavku slijedi primjer programskoga kôda kod kojeg je implementirana mogućnost korištenja ulančavanja metoda te pripadajući pozivi razreda Gradovi s i bez korištenja ulančavanja.

Unutar implementacije razreda Gradovi nalazi se konstruktor, metoda ispis() koja ispisuje vrijednost koja je postavljena unutar atributa naziv te dvije metode zagreb() i split() koje postavljaju vrijednost atributa naziv. Ove tri metode (bez konstruktora) kao povratnu vrijednost vraćaju self čime se omogućava ulančavanje.

Primjer programskoga kôda kod kojeg se metode pozivaju bez ulančavanja:

|  |
| --- |
| **class** Gradovi():  **def** \_\_init\_\_(self, n = **None**):  self.naziv = n    **def** ispis(self):  print(self.naziv)     **def** zagreb(self):  self.naziv = **'Zagreb'**  **def** split(self):  self.naziv = **'Split'**  g = Gradovi() g.zagreb() g.ispis() g.split() g.ispis() |
| Izlaz:  Zagreb  Split |

Kod ovog primjera svaka od metoda – zagreb(), split(), ispisi() poziva se nad objektom g na način da se povratne vrijednosti tih metoda ignoriraju. Primjer koji slijedi u nastavku jest primjer kod kojeg se metode pozivaju pomoću ulančavanja.

|  |
| --- |
| **class** Gradovi():  **def** \_\_init\_\_(self, n = **None**):  self.naziv = n    **def** ispis(self):  print(self.naziv)  **return** self    **def** zagreb(self):  self.naziv = **'Zagreb'  return** self    **def** split(self):  self.naziv = **'Split'  return** self   g = Gradovi() g.zagreb().ispis().split().ispis() |
| Izlaz:  Zagreb  Split |

Opis izvršavanja po koracima:

1. korak:

g.zagreb().ispisi().split().ispisi()

Izvršava se g.zagreb(), unutar metode zagreb() najprije se atribut ispis postavlja na vrijednost ″Zagreb″ te se vraća self.

1. korak:

obj.ispisi().split().ispisi()

Izvršava se obj.ispisi(), unutar metode ispisi() ispisuje se atribut ispis te se vraća self. Ime obj predstavlja objekt koji je vraćen pomoću naredbe return iz prethodne metode.

1. korak:

obj.split().ispisi()

Izvršava se self.split(), unutar metode zagreb() najprije se atribut ispis postavlja na vrijednost ″Zagreb″ te se vraća self.

1. korak:

obj.ispisi()

Izvršava se obj.ispisi(), unutar metode ispisi() ispisuje se atribut ispis te se vraća self.

* 1. Vježba: Objektno orijentirano programiranje - IV

1. Napišite apstraktni razred imena Zivotinja. Tako kreirani apstraktni razred neka sadrži apstraktne metode imena glasanje() i pokret(). Napišite još dva razreda imena Pas i Puz, ti razredi neka nasljeđuju apstraktni razred Zivotinja. U glavnom programu kreirajte po jedan objekt razreda Pas i Puz te pozovite implementirane metode koje ispisuju kako se određena životinja glasa i kako se kreće.
2. Proučite poglavlje 4.13. i shvatite kako funkcionira prikazani primjer (preporučuje se da prikazani primjer napišete na računalu te ga pokrenete).
   1. Pitanja za ponavljanje: Objektno orijentirano programiranje
3. Za kompleksnije programe, je li bolji pristup korištenje proceduralnoga programiranja ili objektno orijentiranoga programiranja?
4. Kako se zove osnovna cjelina u objektno orijentiranom programiranju?
5. Kako se naziva specijalna metoda koja se pokreće automatski prilikom instanciranja objekta?
6. Mora li obavezno u svakoj klasi postojati konstruktor?
7. Čemu služi parametar self?
8. Koja je razlika između varijable objekta i varijable razreda?
9. Kada je riječ o vidljivosti varijabli, koja tri tipa vidljivosti postoje?
10. Koja je razlika između javne varijable i privatne varijable?
11. Što je nasljeđivanje?
12. Što je preopterećenje?
13. Što je nadjačavanje?
14. Algoritmi sortiranja

Po završetku ovoga poglavlja polaznik će moći:

objasniti što su to algoritmi sortiranja

navesti najpopularnije algoritme sortiranja i njihovu složenost

objasniti kada se koristi ugrađena funkcija sort(), a kada je potrebno implementirati algoritam za sortiranje podataka

implementirati algoritme: sortiranje biranjem i sortiranjem zamjenom susjednih elemenata.

Algoritmi sortiranja omogućavaju sortiranje elemenata uzlaznim ili silaznim redoslijedom. Postoji mnogo različitih algoritama koji su razvijeni u svrhu sortiranja elemenata, a u ovom poglavlju opisana je implementacija sljedećih algoritama: sortiranje biranjem, sortiranje zamjenom susjednih elemenata te poboljšani algoritam sortiranja zamjenom susjednih elemenata. Algoritmi koji se najčešće pojavljuju u programskim rješenjima su (s desne strane svakog algoritma nalazi se njegova složenost):

* sortiranje biranjem (engl. *Selection sort*) – O(n2)
* sortiranje zamjenom susjeda (engl. *Bubble sort*) – O(n2)
* sortiranje zamjenom elemenata (engl. *Exchange sort*) – O(n2)
* sortiranje umetanjem (engl. *Insertion sort*) – O(n2)
* sortiranje sjedinjavanjem (engl. *Merge sort*) – O(n log n)
* brzo sortiranje po Hoareu (engl. *Quick sort*) – O(n log n)
* sortiranje razvrstavanjem u pretince (engl. *Bucket sort*) – O(n)

Svaki od ovih algoritama izveden je na drugačiji način (drugačija implementacija) i ima drugačiju složenost, tj. brzinu sortiranja podataka. Razvijen je velik broj algoritama za sortiranje podataka, no izrazito je teško izdvojiti algoritam koji bi bio najbolji. Svaki od razvijenih algoritama ima svoje prednosti i mane te se, ovisno o problemu, odabire algoritam koji će se primijeniti. U *Pythonu* postoji ugrađena metoda sort() koja služi za sortiranje podataka. Ova ugrađena metoda izrazito efikasno sortira elemente liste te ju je preporučljivo koristiti gdje god je to moguće. Algoritmi za sortiranje obično se pišu u situacijama kada nije moguće koristiti ugrađenu metodu sort(). Na primjer, ako postoje dvije liste, u jednoj listi nalaze se imena osoba, a u drugoj listi nalaze se rezultati natjecanja (podaci su povezani preko zajedničkog indeksa). Ako je potrebno poredati imena osoba uzlazno ovisno o ostvarenim bodovima, tada nije moguće koristiti ugrađenu metodu sort(), već je potrebno ručno implementirati algoritam koji će istovremeno sortirati obje liste.

Na poveznici: <https://www.toptal.com/developers/sorting-algorithms> moguće je vizualno vidjeti razliku u brzini sortiranja pojedinih algoritama.

**Složenost algoritma**

Složenost algoritma opisana je vremenom koje je potrebno za izvršavanje algoritma.

* O(1) – vrijeme koje je potrebno da se algoritam izvrši **ne ovisi** o količini elemenata u kolekciji
* O(n) – algoritam čije vrijeme izvršavanja linearno ovisi o broju elemenata u kolekciji
* O(n2) – algoritam čije vrijeme izvršavanja linearno ovisi o kvadratu broja elemenata u kolekciji
* neke od ostalih složenosti: O(n log n), O(n!), O(nc)...

**Objašnjenje složenosti na konkretnim primjerima**

Neka se za primjer uzme lista koju je potrebno sortirati algoritmima složenosti: , , , . U toj listi neka se nalazi elemenata (1.000.000), također, uzeta je u obzir najčešća pretpostavka da računalo u sekundi može obraditi instrukcija (1.000.000.000). U nastavku se nalazi matematički izračun koliko vremena je potrebno kako bi se izvršila operacija sortiranja liste ovisno o složenosti odabranog algoritma.

* 1. Sortiranje biranjem (engl. *Selection sort*)

Ovaj algoritam zasniva se na traženju vrijednosti, od najmanje prema najvećoj vrijednosti ili pak od najveće prema najmanjoj vrijednosti. U daljnjem tekstu svi algoritmi opisivat će sortiranje elemenata od najmanje vrijednosti prema najvećoj vrijednosti, tj. uzlaznim redoslijedom.

Na početku ovog algoritma pronalazi se najmanja vrijednost, tako pronađena najmanja vrijednost stavlja se na prvo mjesto, nakon toga traži se sljedeća najmanja vrijednost koja se stavlja na drugo mjesto i tako dalje sve do trenutka kada niz bude sortiran.

Niz nad kojim ovaj algoritam radi moguće je podijeliti u dva dijela:

* Unutar prvog dijela niza nalaze se vrijednosti koje su sortirane prema uzlaznom ili silaznom redoslijedu. U vizualnom prikazu ovog algoritma, ovaj dio niza prikazan je zelenom bojom.
* Drugi dio niza sadrži elemente koji nisu sortirani. Unutar ovog dijela niza traži se sljedeći element koji će se staviti na kraj sortiranoga dijela niza. U vizualnom prikazu ovog algoritma, ovaj dio niza prikazan je narančastom bojom.

U nastavku slijedi vizualni prikaz funkcioniranja ovog algoritma.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 8 | 6 | **1** | 3 | 5 |
| 1 | 4 | **2** | 8 | 6 | 7 | 3 | 5 |
| 1 | 2 | 4 | 8 | 6 | 7 | **3** | 5 |
| 1 | 2 | 3 | 8 | 6 | 7 | **4** | 5 |
| 1 | 2 | 3 | 4 | 6 | 7 | 8 | **5** |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | **6** |
| 1 | 2 | 3 | 4 | 5 | 6 | 8 | **7** |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | **8** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

**Koraci algoritma:**

* U nesortiranom dijelu niza (narančasti dio) traži se najmanja (ili najveća) vrijednost.
* Pronađena najmanja (ili najveća) vrijednost mijenja se s prvim članom nesortiranoga dijela niza.
* Prethodna dva koraka ponavljaju se tako dugo dok cijeli niz ne bude sortiran.

Primjer programskoga kôda ovog algoritma (uzlazno sortiranje):

|  |
| --- |
| niz = [7, 4, 2, 8, 6, 1, 3, 5]  print(**"Nesortiran niz:"**, niz) brojElemenata = len(niz)  **for** i **in** range(brojElemenata):   minIndex = i  **for** j **in** range(i + 1, brojElemenata):  **if** niz[j] < niz[minIndex]:  minIndex = j   niz[i], niz[minIndex] = niz[minIndex], niz[i]   print(**"Iteracija:"**, i + 1, niz)  print(**"Sortiran niz:"**, niz) |
| Izlaz:  Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]  Iteracija: 1 [**1**, 4, 2, 8, 6, 7, 3, 5]  Iteracija: 2 [1, **2**, 4, 8, 6, 7, 3, 5]  Iteracija: 3 [1, 2, **3**, 8, 6, 7, 4, 5]  Iteracija: 4 [1, 2, 3, **4**, 6, 7, 8, 5]  Iteracija: 5 [1, 2, 3, 4, **5**, 7, 8, 6]  Iteracija: 6 [1, 2, 3, 4, 5, **6**, 8, 7]  Iteracija: 7 [1, 2, 3, 4, 5, 6, **7**, 8]  Iteracija: 8 [1, 2, 3, 4, 5, 6, 7, **8**]  Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8] |

* 1. Sortiranje zamjenom susjednih elemenata (engl. *Bubble sort*)

Još jedan od algoritama sortiranja jest "sortiranje zamjenom susjednih elemenata". Ovaj algoritam prolazi kroz niz te uspoređuje parove vrijednosti, tj. susjedne elemente. U slučaju da su ti elementi (susjedi) u neispravnom poretku, tada se njihove vrijednosti mijenjaju. Algoritam kroz nesortirani dio niza prolazi tako dugo dok niz ne postane sortiran.

Niz nad kojim ovaj algoritam radi moguće je podijeliti u dva dijela:

* Unutar prvog dijela niza nalaze se vrijednosti koje nisu sortirane. U vizualnom prikazu ovog algoritma, ovaj dio niza prikazan je narančastom bojom. Vrijednosti koje se uspoređuju i eventualno mijenjaju (u slučaju neispravnoga poretka) označene su sivom bojom.
* Drugi dio niza sadrži elemente koji su sortirani. U vizualnom prikazu ovog algoritma, ovaj dio niza prikazan je zelenom bojom.

Ovaj algoritam ima složenost O(n2).

**Koraci algoritma:**

* Algoritam prolazi kroz nesortirani dio niza (narančasti dio).
* Ako dvije susjedne vrijednosti koje se uspoređuju nisu u ispravnom poretku (ovisno o uzlaznom ili silaznom sortiranju), tako pronađene vrijednosti se mijenjaju. Algoritam nakon zamjene nastavlja dalje od mjesta gdje je zadnji put zamijenio dvije vrijednosti. Onog trenutka kada algoritam dođe do kraja niza, taj niz nije sortiran, već je na zadnje mjesto unutar niza dovedena najveća vrijednost za uzlazno sortirani niz ili pak najmanja vrijednost za silazno sortirani niz.
* Nakon što algoritam dođe unutar jednog prolaza do kraja niza, u sljedećem prolazu on kreće od početka te se u sortirani dio niza dovodi sljedeća najveća vrijednost.

**1. prolaz (7 usporedbi)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 8 | 6 | 1 | 3 | 5 |
| 4 | 7 | 2 | 8 | 6 | 1 | 3 | 5 |
| 4 | 2 | 7 | 8 | 6 | 1 | 3 | 5 |
| 4 | 2 | 7 | 8 | 6 | 1 | 3 | 5 |
| 4 | 2 | 7 | 6 | 8 | 1 | 3 | 5 |
| 4 | 2 | 7 | 6 | 1 | 8 | 3 | 5 |
| 4 | 2 | 7 | 6 | 1 | 3 | 8 | 5 |
| 4 | 2 | 7 | 6 | 1 | 3 | 5 | 8 |

**2. prolaz (6 usporedbi)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 7 | 2 | 6 | 1 | 3 | 5 | 8 |
| 4 | 7 | 2 | 6 | 1 | 3 | 5 | 8 |
| 4 | 2 | 7 | 6 | 1 | 3 | 5 | 8 |
| 4 | 2 | 6 | 7 | 1 | 3 | 5 | 8 |
| 4 | 2 | 6 | 1 | 7 | 3 | 5 | 8 |
| 4 | 2 | 6 | 1 | 3 | 7 | 5 | 8 |
| 4 | 2 | 6 | 1 | 3 | 5 | 7 | 8 |

**3. prolaz (5 usporedbi)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 2 | 6 | 1 | 3 | 5 | 7 | 8 |
| 2 | 4 | 6 | 1 | 3 | 5 | 7 | 8 |
| 2 | 4 | 6 | 1 | 3 | 5 | 7 | 8 |
| 2 | 4 | 1 | 6 | 3 | 5 | 7 | 8 |
| 2 | 4 | 1 | 3 | 6 | 5 | 7 | 8 |
| 2 | 4 | 1 | 3 | 5 | 6 | 7 | 8 |

**4. prolaz (4 usporedbe)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 1 | 3 | 5 | 6 | 7 | 8 |
| 2 | 4 | 1 | 3 | 5 | 6 | 7 | 8 |
| 2 | 1 | 4 | 3 | 5 | 6 | 7 | 8 |
| 2 | 1 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 | 1 | 3 | 4 | 5 | 6 | 7 | 8 |

**5. prolaz (3 usporedbe)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**6. prolaz (2 usporedbe)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**7. prolaz (1 usporedba)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Primjer programskoga kôda ovog algoritma (uzlazno sortiranje):

|  |
| --- |
| niz = [7, 4, 2, 8, 6, 1, 3, 5]  print(**"Nesortiran niz:"**, niz) brojElemenata = len(niz)  **for** i **in** range(brojElemenata - 1):   **for** j **in** range(brojElemenata - 1 - i):  **if** niz[j + 1] < niz[j]:  niz[j], niz[j + 1] = niz[j + 1], niz[j]   print(**"Iteracija:"**, i + 1, niz)  print(**"Sortiran niz:"**, niz) |
| Izlaz:  Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]  Iteracija: 1 [4, 2, 7, 6, 1, 3, 5, 8]  Iteracija: 2 [2, 4, 6, 1, 3, 5, 7, 8]  Iteracija: 3 [2, 4, 1, 3, 5, 6, 7, 8]  Iteracija: 4 [2, 1, 3, 4, 5, 6, 7, 8]  Iteracija: 5 [1, 2, 3, 4, 5, 6, 7, 8]  Iteracija: 6 [1, 2, 3, 4, 5, 6, 7, 8]  Iteracija: 7 [1, 2, 3, 4, 5, 6, 7, 8]  Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8] |

* 1. Poboljšani algoritam zamjene susjednih elemenata

Prethodno objašnjeni algoritam moguće je poboljšati. Ako se pogleda vizualni prikaz algoritma "sortiranje zamjenom susjednih algoritama", vidljivo je da se u zadnjim dvama koracima nije dogodila nijedna zamjena (6. i 7. prolaz). Algoritam zamjene susjednih elemenata moguće je poboljšati tako da se uvede dodatna varijabla (u donjem primjeru programskoga kôda dodatna varijabla nazvana je imenom zamjena). U tu varijablu bit će zapisano je li se unutar nekog prolaza dogodila zamjena ili ne. Ako unutar nekog prolaza po nesortiranom dijelu niza nema zamjene, niz se proglašava sortiranim te se pomoću naredbe break prekida daljnje uspoređivanje susjednih elemenata, tj. sljedeći prolazak po nizu.

Ako se pogleda vizualni prikaz funkcioniranja algoritma koji je prikazan u prethodnom poglavlju, kod ovog poboljšanog algoritma taj bi prikaz izgledao ovako:

* Od 1. do 5. prolaza sve ostaje identično.
* U 6. prolazu nema zamjene susjednih elemenata te se preddefinirana vrijednost varijable zamjena ne postavlja na True. Na kraju ovoga prolaza, poziva se naredba break koja sprječava algoritam u daljnjim prolascima po nizu.
* 7. prolaz se neće dogoditi.

U nastavku slijedi prikaz programskoga kôda poboljšanog algoritama.

|  |
| --- |
| niz = [7, 4, 2, 8, 6, 1, 3, 5] print(**"Nesortiran niz:"**, niz) brojElemenata = len(niz)  **for** i **in** range(brojElemenata - 1):  zamjena = **False  for** j **in** range(brojElemenata - 1 - i):  **if** niz[j + 1] < niz[j]:  niz[j], niz[j + 1] = niz[j + 1], niz[j]  zamjena = **True** print(**"Prolaz:"**, i + 1, niz)   **if** zamjena == **False**:  **break** print(**"Sortiran niz:"**, niz) |
| Izlaz:  Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]  Prolaz: 1 [4, 2, 7, 6, 1, 3, 5, 8]  Prolaz: 2 [2, 4, 6, 1, 3, 5, 7, 8]  Prolaz: 3 [2, 4, 1, 3, 5, 6, 7, 8]  Prolaz: 4 [2, 1, 3, 4, 5, 6, 7, 8]  Prolaz: 5 [1, 2, 3, 4, 5, 6, 7, 8]  Prolaz: 6 [1, 2, 3, 4, 5, 6, 7, 8]  Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8] |

* 1. Usporedba funkcije sorted() i metode sort()

**Funkcija sorted()**

* Ugrađena funkcija
* Radi za bilo koji *iterable* (na primjer: niz znakova, lista, n-terac, rječnik (dobivaju se ključevi), generator, itd.)
* Vraća novu sortiranu listu
* Originalna lista ostaje netaknuta

**Metoda sort()**

* Metoda liste
* Radi samo za listu
* Vraća None vrijednost
* Sortira listu nad kojom je pozvana, tj. na mjestu (engl. *in-place*)
* Nakon poziva ove metode, gubi se originalni poredak elemenata objekta nad kojim je ova metoda pozvana

Ako se želi sortirati listu, metoda sort() brža je od funkcije sorted() zato što metoda sort() ne mora kreirati novu listu. Za bilo koji drugi *iterable* potrebno je koristiti funkciju sort().

**Primjer funkcije sorted():**

|  |
| --- |
| lista = [6, 4, 8, 5, 10, 7, 3, 1, 9] print(**"lista:"**, lista, **"\n"**)  sortiranaLista = sorted(lista)  print(**"Liste nakon korištenja funkcije sorted():"**) print(**"lista:"**, lista) print(**"sortiranaLista:"**, sortiranaLista) |
| Izlaz:  lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]  Liste nakon korištenja funkcije sorted():  lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]  sortiranaLista: [1, 3, 4, 5, 6, 7, 8, 9, 10] |

**Primjer metode sort():**

|  |
| --- |
| lista = [6, 4, 8, 5, 10, 7, 3, 1, 9] print(**"lista:"**, lista, **"\n"**)  lista.sort()  print(**"Lista nakon korištenja metode sort():"**)  print(**"lista:"**, lista) |
| Izlaz:  lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]  Lista nakon korištenja metode sort():  lista: [1, 3, 4, 5, 6, 7, 8, 9, 10] |

Funkcija sorted() i metoda sort() za sortiranje koriste algoritam imena **Timsort**. Ovaj algoritam kreirao je Tim Peters 2002. upravo za korištenje u programskom jeziku *Python*. On je mješavina dvaju algoritama: sortiranje sjedinjavanjem (engl. *Merge sort*) i sortiranje umetanjem (engl. *Insertion sort*). Njegova složenost je O(n log n).

* 1. Vježba: Algoritmi sortiranja

|  |
| --- |
| sudionici = [**"Ivica"**, **"Marica"**, **"Perica"**, **"Nika"**, **"Nikica"**, **"Tihana"**, **"Stjepan"**, **"Petra"**]  rezultati = [10.11, 11.58, 10.08, 10.06, 10.77, 11.22, 11.55, 10.05] |

Pogledajte prethodne dvije liste utrke na 100 metara, sudionici i rezultati. Vaš zadatak je da ih sortirate i ispišete uzlazno prema rezultatima (od najbržeg do najsporijeg).

1. Sortiranje napravite pomoću algoritma "sortiranje biranjem" (engl. *Selection sort*).
2. Sortiranje napravite pomoću poboljšanog algoritma zamjene susjednih elemenata (engl. *Bubble sort*).
   1. Pitanja za ponavljanje: Algoritmi sortiranja
3. Navedite primjer kada nije moguće koristiti ugrađenu metodu sort() već je potrebno implementirati algoritam za sortiranje podataka.
4. Kako funkcionira algoritam za sortiranje imena "sortiranje biranjem"?
5. Kako funkcionira algoritam za sortiranje imena "sortiranje zamjenom susjednih elemenata"?
6. Kako poboljšati algoritam imena "sortiranje zamjenom susjednih elemenata"?
7. Oblikovni obrasci

Po završetku ovoga poglavlja polaznik će moći:

upoznati se s poantom korištenja oblikovnih obrazaca

u praksi implementirati dva oblikovna obrasca, a to su: "jedinstveni objekt" i "promatrač".

Oblikovni obrasci u programiranju temelje se na objektno orijentiranom pristupu i oni rješavaju neke najčešće probleme s kojima se programeri susreću. Oblikovne obrasce u programiranju možemo smatrati ustaljenom praksom rješavanja problema i njihovim korištenjem ubrzava se razvoj programskih rješenja.

**Svojstva (simptomi) lošega programskog kôda:**

* krutost (engl. *rigidity*) – program je teško promijeniti čak i na jednostavne načine, jer svaka promjena zahtijeva nove promjene, zbog dugog lanca eksplicitne ovisnosti (domino-efekt). Rješenje je kraćenje lanca eksplicitne ovisnosti primjenom *apstrakcije* i *enkapsulacije*.
* krhkost (engl. *fragility*) – tendencija programa da puca uslijed promjena. Najčešće se događa zbog implicitne međuovisnosti uslijed ponavljanja programskoga kôda te se jedna konceptualna izmjena mora unositi na više različitih lokacija.
* nepokretnost (engl. *immobility*) – otežano višekratno korištenje već razvijenoga programskog kôda (engl. *reusability*). Lakše je napisati programski kôd ponovo nego koristiti već gotove programske komponente. Čest uzrok je pretjerana međuovisnost zbog neadekvatnih sučelja i neadekvatne raspodijele funkcionalnosti po komponentama. Nepokretnost potiče ponavljanje, odnosno krhkost i krutost.
* viskoznost (engl. *viscosity*) – programski sustav je viskozan kad ga je teško nadograđivati uz očuvanje konceptualnog integriteta programa. Postoje dvije vrste viskoznosti:
  + *viskoznost programske organizacije* – nadogradnje koje čuvaju integritet zahtijevaju puno manualnoga rada ili nisu očite (promjene je teško unijeti u skladu s originalnom zamisli, novu funkcionalnost je lakše dodati na dugoročno loš način)
  + *viskoznost razvojnoga procesa* – spora, neefikasna razvojna okolina (npr. komplicirani sustav za verziranje implicira rjeđe sinkronizacije kôda te kasnije otkrivanje problema u vezi s integracijom, sporo prevođenje pospješuje unošenje "zakrpa'' umjesto primjerenog održavanja organizacije).“ **[[1]](#endnote-1)**
  1. Upoznavanje s oblikovnim obrascima

Oblikovni obrasci opisani su tako da ih je moguće primijeniti na mnogo različitih primjera. Oblikovni obrasci često se miješaju s algoritmima, no u suštini to su dvije različite stvari. Algoritam definira korake koje je potrebno napraviti kako bi se postigao neki cilj, dok je oblikovni obrazac uputa prema kojoj je moguće riješiti neki problem na najbolji mogući način.

Oblikovne obrasce moguće je podijeliti u tri osnovna skupa obrazaca, a to su:

1. ponašajni obrasci – pomažu oko implementacije same komunikacije između objekata.
2. strukturni obrasci – objašnjavaju kako sastaviti objekte i razrede u veće strukture, a da se takva složenija struktura uspije zadržati fleksibilna i učinkovita.
3. obrasci stvaranja – ovaj tip obrazaca pruža razne mehanizme stvaranja novih objekata te istovremeno povećavaju fleksibilnost i ponovno iskorištavanje programskoga kôda.

U nastavku slijedi popis nekih najčešće korištenih oblikovnih obrazaca:

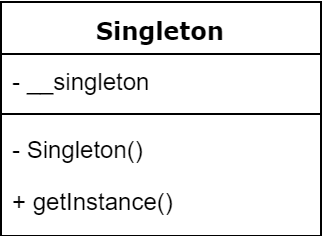
1. ponašajni obrasci:
   * iterator (engl. *Iterator Pattern*)
   * metoda predložak (engl. *Template Method Pattern*)
   * naredba (engl. *Command Pattern*)
   * posjetitelj (engl. *Visitor Pattern*)
   * promatrač (engl. *Observer Pattern*)
   * stanje (engl. *State Pattern*)
   * strategija (engl. *Strategy Pattern*)
2. strukturni obrasci:
   * adapter (engl. *Adapter Pattern*)
   * dekorator (engl. *Decorator Pattern*)
   * fasada (engl. *Facade Pattern*)
   * kompozit (engl. *Composite Pattern*)
   * most (engl. *Bridge Pattern*)
3. obrasci stvaranja:
   * apstraktna tvornica (engl. *Abstract Factory Pattern*)
   * graditelj (engl. *Builder Pattern*)
   * jedinstveni objekt (engl. *Singleton Pattern*)
   * metoda tvornica (engl. *Factory Method Pattern*)
   * prototip (engl. *Prototype Pattern*).

U ovom priručniku obradit će se dva oblikovna obrasca. Prvi oblikovni obrazac je "jedinstveni objekt" (engl. *Singleton Pattern*), a drugi oblikovni obrazac je "promatrač" (engl. *Observer Pattern*).

Uz oblikovne obrasce koji se preporučuju koristiti u razvoju programskoga kôda i koji su primjer dobre prakse razvoja programa, postoje i takozvani antiobrasci (engl. *anti-pattern*). Pojam antiobrazac označava obrazac koji je vrlo neučinkovit i/ili je vrlo kontraproduktivan te se preporučuje da se izbjegne korištenje takvoga načina razvoja programskoga kôda. Antiobrasci se u ovom priručniku neće obrađivati. U nastavku slijedi nekoliko primjera: "Reinvent the wheel", "Spaghetti code", "Swiss-Army Knife", "Lava Flow", "Dead End", "Boat Anchor".

* 1. Jedinstveni objekt (engl. *Singleton Pattern*)

Oblikovni obrazac "jedinstveni objekt" je jedan od najjednostavnijih obrazaca. To je moguće vidjeti i iz dijagrama razreda koji je prikazan u nastavku. Ovaj oblikovni obrazac omogućava samo jedno kreiranje objekta nekoga razreda, što znači da ne može postojati više različitih objekata jednoga te istog razreda.



Ovaj obrazac ima jednu privatnu statičku varijablu (varijabla koja pripada razredu, a ne objektu) imena \_\_singleton i unutar nje se sprema kreirani objekt razreda. Na početku se vrijednost te varijable postavlja na None, što omogućava da se ispita je li objekt kreiran ili nije. Postoji više mogućnosti za implementaciju ovog oblikovnog obrasca. U nastavku ovoga priručnika prikazan je način implementacije pomoću statičke metode.

**Implementacija pomoću statičke metode**

Prvi primjer prikazuje implementaciju oblikovnog obrasca "jedinstveni objekt" pomoću statičke metode. Kreirani objekt razreda Singleton moguće je dobiti na dva načina: pozivom metode getInstance() i klasičnim kreiranjem novog objekta preko konstruktora. Ako se objekt želi dohvatiti sljedećim pozivom: Singleton(), to je moguće napraviti samo u slučaju ako instanca toga jedinstvenog objekta nije kreirana, jer će u svakom drugom slučaju unutar konstruktora biti podignuta iznimka. Nevezano za to je li objekt razreda Singleton već kreiran ili ne, moguće je pozvati statičku metodu imena getInstance(), koja će provjeriti je li instanca kreirana ili ne. U slučaju da je instanca kreirana, vraća se taj objekt, a ako nije, poziva se konstruktor koji kreira novi objekt razreda te se novo kreirani objekt vraća.

|  |
| --- |
| **class** Singleton:  \_\_singleton = **None   def** \_\_init\_\_(self):  **if** Singleton.\_\_singleton == **None**:  Singleton.\_\_singleton = self  **else**:  **raise** Exception(**"Singleton!"**)   @staticmethod  **def** getInstance():  **if** Singleton.\_\_singleton == **None**:  Singleton()  **return** Singleton.\_\_singleton   s1 = Singleton() print(s1)  *# Iznimka >>> Exception: Singleton! # s2 = Singleton() # print(s2)* s3 = Singleton.getInstance() print(s3)  s4 = Singleton.getInstance() print(s4) |
| Izlaz:  <\_\_main\_\_.Singleton object at 0x033C6810>  <\_\_main\_\_.Singleton object at 0x033C6810>  <\_\_main\_\_.Singleton object at 0x033C6810> |

* 1. Promatrač (engl. *Observer Pattern*)

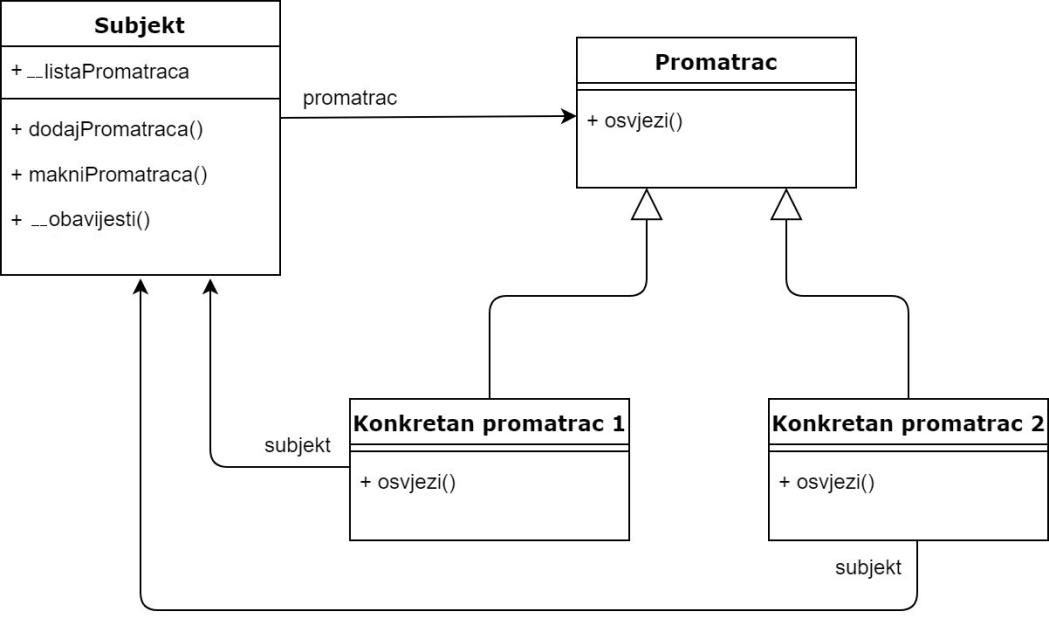
Oblikovni obrazac "promatrač" omogućava da se na jednostavan način obavijeste svi objekti koji su pretplaćeni na obavještavanje da je došlo do promjene podatka unutar glavnog objekta. Ovaj oblikovni obrazac stvara vezu od jednog prema više objekata (engl. *One to many*).

**Sudionici:**

* Subjekt (engl. *Subject*) – u ovom objektu nalaze se originalne vrijednosti svih varijabli koje se dalje šalju pretplatnicima. Unutar subjekta implementirane su metode koje omogućavaju prijavu i odjavu promatrača te se na temelju takvoga popisa obavještavaju "pretplaćeni" sudionici.
* Promatrač (engl. *Observer*) – ovaj razred pruža sučelje na temelju kojega ga nasljeđuju drugi razredi te nadjačavaju njegove metode. *Ovo je apstraktni razred*.
* Konkretan promatrač (engl. *Concrete Observer*) – nasljeđuje bazni razred "Promatrač". "Konkretnih promatrača" može biti neograničeno, jedino je potrebno da se registriraju kod "Subjekta" na osvježavanje informacija (naravno, ako imaju potrebu za tom pretplatom).

Subjekt može imati neograničenu količinu promatrača. Prednost ovog oblikovnog obrasca jest to da "Subjekt" ne mora unaprijed znati točan broj pretplatnika na njegove informacije te se pretplatnici mogu dinamički prijavljivati i odjavljivati kod "Subjekta". Negativna strana je to što se može dogoditi da se obavijesti neki "Konkretan promatrač" kojega se promjena ne tiče (ne tiču se sve promjene svih "Konkretnih promatrača"). Ovakav pristup omogućuje laku implementaciju vrlo bitnoga načela objektno orijentiranoga načina pristupa programiranju, a to je: **nadogradnja bez promjene**.

U nastavku se nalazi strukturni dijagram oblikovnog obrasca "Promatrač".



U niže prikazanom programskom kôdu nalazi se jednostavna implementacija oblikovnog obrasca "Promatrač". Razred MeteoStanica može se smatrati mjernom postajom (mjeri temperaturu i vlažnost zraka). Unutar njega nalazi se nekoliko metoda:

* \_\_init\_\_(self) – konstruktor koji kreira prazan skup (engl. *Set*) \_\_prijavljeniUredaji unutar kojeg će biti popis svih pretplatnika na informacije te dvije varijable \_\_temp i \_\_vlaga u koje će se zapisivati konkretne informacije.
* dodajUredaj(self, o) – metoda koja u skup dodaje novog pretplatnika, a također kreira poveznicu od pretplatnika do razreda MeteoStanica.
* makniUredaj(self, o) – metoda koja izbacuje pretplatnika iz skupa.
* \_\_obavijesti(self) – metoda koja obavještava sve pretplatnike o novim informacijama/podacima.
* također, unutar ovog razreda nalaze se 2 postavljajuće i 2 dobavljajuće metode.

Unutar apstraktnog razreda Uredaj nalazi se samo poziv konstruktora i prazna metoda imena osvjezi() koja će biti nadjačana implementacijom unutar "konkretnih promatrača".

Konkretni razredi ZaslonZaPrikaz i MobilnaAplikacija nasljeđuju apstraktni razred Uredaj i unutar njih je implementirana metoda osvjezi() koja će se pozvati onoga trenutka kada će razred MeteoStanica raspolagati s novim podacima o čijim će vrijednostima obavijestiti sve pretplaćene objekte. Ovaj razred možemo smatrati objektom koji prikazuje informacije koje je mjerna postaja izmjerila.

U glavnom programu kreiran je jedan objekt razreda MeteoStanica koji predstavlja mjernu postaju te nekoliko objekta razreda ZaslonZaPrikaz i MobilnaAplikacija koji predstavljaju medije na kojima će se temperatura i vlažnost prikazivati.

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   *# Subjekt* **class** MeteoStanica:   **def** \_\_init\_\_(self):  self.\_\_prijavljeniUredaji = set()  self.\_\_temp = **None** self.\_\_vlaga = **None   def** dodajUredaj(self, o):  self.\_\_prijavljeniUredaji.add(o)  o.izvorInformacija = self   **def** makniUredaj(self, o):  self.\_\_prijavljeniUredaji.discard(o)  o.izvorInformacija = **None   def** \_\_obavijesti(self):  print(**"\n> Novi podaci:"**)  **for** o **in** self.\_\_prijavljeniUredaji:  o.osvjezi(self.\_\_temp, self.\_\_vlaga)   @property  **def** temperatura(self):  **return** self.\_\_temp   @temperatura.setter  **def** temperatura(self, v):  self.\_\_temp = v  self.\_\_obavijesti()   @property  **def** vlaznost(self):  **return** self.\_\_vlaga   @vlaznost.setter  **def** vlaznost(self, v):  self.\_\_vlaznost = v  self.\_\_obavijesti()  *# Promatrac* *# Apstraktni razred* **class** Uredaj(ABC):  **def** \_\_init\_\_(self):  self.izvorInformacija = **None** self.\_\_temp = **None** self.\_\_vlaga = **None** @abstractmethod  **def** osvjezi(self, v1, v2):  **pass** *# Konkretan promatrac 1 # Nasljeđen apstraktni razred - Uredaj* **class** ZaslonZaPrikaz(Uredaj):  **def** \_\_init\_\_(self, naziv):  super().\_\_init\_\_()  self.\_\_naziv = naziv   **def** osvjezi(self, t, v):  self.\_\_temp = t  self.\_\_vlaga = v   print(**" Promatrac:"**, self.\_\_naziv)  print(**" Temperatura:"**, self.\_\_temp)  print(**" Vlaga zraka:"**, self.\_\_vlaga)   *# Konkretan promatrac 2 # Nasljeđen apstraktni razred - Uredaj* **class** MobilnaAplikacija(Uredaj):  **def** \_\_init\_\_(self, naziv):  super().\_\_init\_\_()  self.\_\_naziv = naziv   **def** osvjezi(self, t, v):  self.\_\_temp = t  self.\_\_vlaga = v   print(**" Promatrac:"**, self.\_\_naziv)  print(**" T:"**, self.\_\_temp)  print(**" V:"**, self.\_\_vlaga)    *# Glavni program* ms = MeteoStanica()  blagavaonica = ZaslonZaPrikaz(**"Blagavaonica"**) dnevna = ZaslonZaPrikaz(**"Dnevna soba"**) kupanica = ZaslonZaPrikaz(**"Kupaonica"**) spavacaSoba = ZaslonZaPrikaz(**"Spavaca soba"**) mobitel = MobilnaAplikacija(**"Mobitel"**) tablet = MobilnaAplikacija(**"Tablet"**)  ms.dodajUredaj(blagavaonica) ms.dodajUredaj(dnevna) ms.dodajUredaj(spavacaSoba) ms.dodajUredaj(mobitel)  ms.temperatura = 24 ms.vlaznost = 40  ms.makniUredaj(dnevna) ms.makniUredaj(spavacaSoba)  ms.temperatura = 23 ms.vlaznost = 41 |
| Izlaz:  >>> Novi podaci:  Promatrac: Mobitel  T: 24  V: None  Promatrac: Blagavaonica  Temperatura: 24  Vlaga zraka: None  Promatrac: Dnevna soba  Temperatura: 24  Vlaga zraka: None  Promatrac: Spavaca soba  Temperatura: 24  Vlaga zraka: None  >>> Novi podaci:  Promatrac: Mobitel  T: 24  V: None  Promatrac: Blagavaonica  Temperatura: 24  Vlaga zraka: None  Promatrac: Dnevna soba  Temperatura: 24  Vlaga zraka: None  Promatrac: Spavaca soba  Temperatura: 24  Vlaga zraka: None  >>> Novi podaci:  Promatrac: Mobitel  T: 23  V: None  Promatrac: Blagavaonica  Temperatura: 23  Vlaga zraka: None  >>> Novi podaci:  Promatrac: Mobitel  T: 23  V: None  Promatrac: Blagavaonica  Temperatura: 23  Vlaga zraka: None |

* 1. Vježba: Oblikovni obrasci

1. Napišite programski kôd koji će omogućiti generiranje nasumičnih brojeva na način opisan u nastavku. Početna, tj. prva vrijednost koja se mora vratiti jest 67, a svaka sljedeća vrijednost računa se na temelju sljedeće formule:  
   Pomoću oblikovnog obrasca "*jedinstveni objekt*" osigurajte samo jedno instanciranje objekta unutar kojeg će biti pohranjena prethodno generirana vrijednost. Unutar razreda imena NasumicniBroj implementirajte metodu imena sljedeci(), kod svakog njezinog poziva ta metoda vraća novu vrijednost dobivenu na temelju gore napisane formule. Primijetite da će generirane vrijedosti biti identične svaki put kada se programski kôd pokrene ponovo.
2. Pomoću oblikovnog obrasca "*promatrač*" napišite programski kôd koji omogućava pretplaćivanje na motivacijske citate. Razred imena Citat (subjekt) mora prihvaćati nove pretplatnike, obavještavati pretplatnike s informacijom o novom citatu te imati metode za postavljanje i dohvaćanje novoga citata. Apstraktni razred Pretplatnik (promatrač) sadrži apstraktnu metodu osvjezi() koja je implementirana u konkretnim promatračima. Razred KonkretanPretplatnik (konkretni promatrač) nasljeđuje razred Pretplatnik te je u njemu implementirana metoda osvjezi() koja ispisuje ime pretplatnika koji je zaprimio citat te citat.
3. \* Nadogradite prethodni zadatak na način da je razred imena Citat (subjekt) moguće instancirati samo jednom (ovo osigurajte pomoću oblikovnog obrasca "*jedinstveni objekt*"). U glavnom programu ispitajte je li osiguran princip jedinstvenog objekta.
   1. Pitanja za ponavljanje: Oblikovni obrasci
4. Koja je razlika između oblikovnih obrazaca i algoritma?
5. Za što služi oblikovni obrazac "Jedinstveni objekt"?
6. Za što služi oblikovni obrazac "Promatrač"?

Dodatak: Rješenja vježbi

1.12.1.

|  |
| --- |
| lista = [1, 2, 3, 4]  a, b, c, d = lista  print(a, b, c, d) |
| Izlaz:  1 2 3 4 |

1.12.2.

|  |
| --- |
| a = 5 b = 10  print(a, b)  a, b = b, a  print(a, b) |
| Izlaz:  5 10  10 5 |

1.12.3.

|  |
| --- |
| broj = input(**"Unesite broj: "**)  broj = int(broj)  **if** 10 < broj < 30:  print(**"Zadovoljava!"**) **else**:  print(**"Ne zadovoljava!"**) |
| Izlaz 1:  Unesite broj: ***55***  Ne zadovoljava! |
| Izlaz 2:  Unesite broj: ***22***  Zadovoljava! |

1.12.4.

|  |
| --- |
| lista = [1, 2, 3, 4, 5, 6, 7]  broj = input(**"Unesite broj: "**) broj = int(broj)  **for** e **in** lista:  **if** e == broj:  print(**"Postoji!"**)  **break else**:  print(**"Ne postoji!"**) |
| Izlaz 1:  Unesite broj: ***2***  Postoji! |
| Izlaz 2:  Unesite broj: ***11***  Ne postoji! |

1.12.5.

|  |
| --- |
| broj = input(**"Unesite broj: "**) broj = int(broj)  print(**"Paran!" if** broj % 2 == 0 **else "Neparan!"**) |
| Izlaz 1:  Unesite broj: ***1***  Neparan! |
| Izlaz 2:  Unesite broj: ***2***  Paran! |

1.12.6.

|  |
| --- |
| **def** paran():  print(**"Paran!"**)   **def** neparan():  print(**"Neparan"**)   broj = input(**"Unesite broj: "**) broj = int(broj)  (paran **if** broj % 2 == 0 **else** neparan)() |
| Izlaz 1:  Unesite broj: 1  Neparan |
| Izlaz 2:  Unesite broj: 2  Paran! |

1.12.7.

|  |
| --- |
| **def** rezultat(var1, var2):  a = var1 \* var2  b = var1 + var2  c = 10 \* var1 + var2   **return** a, b, c   var1, var2, var3 = rezultat(5, 10)  print(var1, var2, var3) |
| Izlaz:  50 15 60 |

1.12.8.

|  |
| --- |
| nizZnakova = **"Hello World!"** lista = [1, 2, 3, 4]  print(**"Niz znakova (petlja): "**, end=**""**) i = len(nizZnakova) - 1 **while** i >= 0:  print(nizZnakova[i], sep=**""**, end=**""**)  i -= 1  print(**"\nLista (petlja): "**, end=**""**) i = len(lista) - 1 **while** i >= 0:  print(lista[i], sep=**""**, end=**""**)  i -= 1  print(**"Niz znakova:"**, nizZnakova[::-1]) print(**"Lista:"**, lista[::-1]) |
| Izlaz:  Niz znakova (petlja): !dlroW olleH  Lista (petlja): 4321  Niz znakova: !dlroW olleH  Lista: [4, 3, 2, 1] |

2.5.1.

|  |
| --- |
| **def** zbrojiBrojeveDo(x):  **if** x == 1:  **return** 1  **else**:  **return** x + zbrojiBrojeveDo(x - 1)   zbroj = zbrojiBrojeveDo(5) print(zbroj) |
| Izlaz:  15 |

2.5.2.

|  |
| --- |
| **def** brojParnihZnamenki(x):  **if** x == 0:  **return** 0  **elif** x % 2 == 0:  **return** 1 + brojParnihZnamenki(x // 10)  **else**:  **return** brojParnihZnamenki(x // 10)   x = input(**"Unesite broj: "**) x = int(x)  broj = brojParnihZnamenki(x) print(broj) |
| Izlaz:  Unesite broj: ***123456***  3 |

2.5.3.

|  |
| --- |
| fun = **lambda** a, b, c: (a \* b + c) / c  rezultat = fun(5, 10, 2)  print(rezultat) |
| Izlaz:  26.0 |

2.5.4.

|  |
| --- |
| fun = **lambda** a, b, c: (a\*b+c) / c **if** (c != 0) **else "Err!"** rezultat = fun(5, 10, 0)  print(rezultat) |
| Izlaz:  Greška u predanim podacima! |

2.5.5.

|  |
| --- |
| **def** izracun(a, b):  **def** parniZbroj(a, b):  **return** 2 \* a + 5 \* b   **def** neparniZbroj(a, b):  **return** a \* b - 10   **if** (a + b) % 2 == 0:  **return** parniZbroj(a, b)  **else**:  **return** neparniZbroj(a, b)   rezultat = izracun(5, 15) print(rezultat) |
| Izlaz:  85 |

2.5.6.

|  |
| --- |
| **def** danUTjednu():  dani = [**"Ponedjeljak"**,  **"Utorak"**,  **"Srijeda"**,  **"Četvrtak"**,  **"Petak"**,  **"Subota"**,  **"Nedjelja"**]   **for** e **in** dani:  **yield** e   g = danUTjednu()  **for** e **in** g:  print(e) |
| Izlaz:  Ponedjeljak  Utorak  Srijeda  Četvrtak  Petak  Subota  Nedjelja |

2.5.7.

|  |
| --- |
| **def** jedanDvaTri():  brojac = [**"Jedan"**,  **"Dva"**,  **"Tri"**]   **while True**:  **for** e **in** brojac:  **yield** e   g = jedanDvaTri()  print(next(g)) print(next(g)) print(next(g)) print(next(g)) print(next(g)) print(next(g)) print(next(g)) print(next(g)) |
| Izlaz:  Jedan  Dva  Tri  Jedan  Dva  Tri  Jedan  Dva |

2.5.8.

|  |
| --- |
| **def** prebroji(broj, znamenka):  **if** broj == 0:  **return** 0  **elif** (broj % 10) == znamenka:  **return** 1 + prebroji(broj // 10, znamenka)  **else**:  **return** prebroji(broj // 10, znamenka)   rezultat = prebroji(1223455, 2) print(rezultat) |
| Izlaz:  2 |

2.5.9.

|  |
| --- |
| **def** jeProst(broj):  **def** jeProstInner(broj, i):  **if** broj == i:  **return** 1  **if** broj % i == 0:  **return** 0  **return** jeProstInner(broj, i + 1)   **return** jeProstInner(broj, 2)   **if** jeProst(7) == 1:  print(**"Prost je!"**) **else**:  print(**"Nije prost!"**) |
| Izlaz:  Prost je! |

2.5.10.

|  |
| --- |
| celsius = [0, 5, 10, 15.8]  fahrenheit = map(**lambda** x: (float(9) / 5) \* x + 32, celsius)  print(list(fahrenheit)) |
| Izlaz:  [32.0, 41.0, 50.0, 60.44] |

2.5.11.

|  |
| --- |
| **def** bottles(brojBoca=99, nazivNapitka=**"beer"**):  brojBocaNaPocetku = brojBoca  **while** brojBoca >= 0:   **if** brojBoca >= 2:  **yield** (str(brojBoca) + **" bottles of "** +  nazivNapitka + **" on the wall, "** +  str(brojBoca) + **" bottles of "** +  nazivNapitka + **".\n"** +  **"Take one down and pass it around, "** +  str(- 1) + **" bottles of "** +  nazivNapitka + **" on the wall.\n"**)   **elif** brojBoca == 1:  **yield** (str(brojBoca) + **" bottle of "** +  nazivNapitka + **" on the wall, "** +  str(brojBoca) + **" bottle of "** +  nazivNapitka + **".\n"** +  **"Take one down and pass it around, "** +  **"no more bottles of "** +  nazivNapitka + **" on the wall.\n"**)  **else**:  **yield** (**"No more bottles of "** + nazivNapitka +  **" on the wall, no more bottles of "** +  nazivNapitka + **".\n"** +  **"Go to the store and buy some more, "** + str(brojBocaNaPocetku) +  **" bottles of "** +  nazivNapitka + **" on the wall."**)   brojBoca -= 1  g = bottles(4)  **for** e **in** g:  print(e) |
| Izlaz:  4 bottles of beer on the wall, 4 bottles of beer.  Take one down and pass it around, 3 bottles of beer on the wall.  3 bottles of beer on the wall, 3 bottles of beer.  Take one down and pass it around, 2 bottles of beer on the wall.  2 bottles of beer on the wall, 2 bottles of beer.  Take one down and pass it around, 1 bottles of beer on the wall.  1 bottle of beer on the wall, 1 bottle of beer.  Take one down and pass it around, no more bottles of beer on the wall.  No more bottles of beer on the wall, no more bottles of beer.  Go to the store and buy some more, 4 bottles of beer on the wall. |

2.6.1.

Rekurzivne funkcije su funkcije koje pozivaju same sebe.

2.6.2.

Rekurzivni poziv funkcije NIJE efikasniji od iterativnog rješenja.

2.6.3.

Rekurzivni poziv funkcije nije efikasniji od iterativnog rješenja jer svaki poziv funkcije uključuje operacije kao što su: adresiranje, postavljanje argumenata i povratnih vrijednosti na stog.

2.6.4.

Funkcija će sama sebe rekurzivno pozivati beskonačno puta (u stvarnosti dubina rekurzije ograničena je veličinom memorije).

2.6.5.

Bezimena funkcija je funkcija koja je definirana bez imena.

2.6.6.

Drugi naziv za bezimenu funkciju je lambda funkcija.

2.6.7.

Sintaksa bezimena funkcije je:  
lambda [argumenti] : [izraz]

2.6.8.

Bezimena funkcija može primiti neograničenu količinu argumenata.

2.6.9.

Da, bezimenu funkciju je moguće spremiti u objekt, tj. varijablu.

2.6.10.

Ugniježđene funkcije su funkcije koje su definirane unutar drugih funkcija.

2.6.11.

Ugniježđene funkcije najčešće se koriste kako bi se postiglo skrivanje ponašanja nekoga dijela programskoga kôda od ostatka programa.

2.6.12.

|  |  |
| --- | --- |
| **Normalne funkcije** | **Generatorske funkcije** |
| koriste ključnu riječ return | koriste ključnu riječ yield |
| vraćaju povratnu vrijednost jednom | otpuštaju vrijednosti više puta |
| vraćaju vrijednost | vraćaju generator |

3.7.1.

|  |
| --- |
| unos = input(**"Unesite broj: "**) broj = int(unos) print(broj) |
| Izlaz 1:  Unesite broj: ***5***  5 |
| Izlaz 2:  Unesite broj: ***broj***  Traceback (most recent call last):  File "test.py", line 2, in <module>  broj = int(unos)  ValueError: invalid literal for int() with base 10: 'broj' |

3.7.2.

|  |
| --- |
| **try**:  unos = input(**"Unesite broj: "**)  broj = int(unos)  print(broj) **except**:  print(**"Iznimka je uhvaćena!"**) |
| Izlaz 1:  Unesite broj: ***5***  5 |
| Izlaz 2:  Unesite broj: ***broj***  Iznimka je uhvaćena! |

3.7.3.

|  |
| --- |
| **try**:  a = int(input(**"Unesite vrijednost a: "**))  b = int(input(**"Unesite vrijednost b: "**))  c = int(input(**"Unesite vrijednost c: "**))   rezultat = (a + b) / c  print(rezultat) **except** ValueError:  print(**"Iznimka - ValueError"**) **except** ZeroDivisionError:  print(**"Iznimka - ZeroDivisionError"**) **except**:  print(**"Ostale iznimke!"**) |
| Izlaz 1:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***broj***  Iznimka - ValueError |
| Izlaz 2:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***2***  Unesite vrijednost c: ***0***  Iznimka - ZeroDivisionError |
| Izlaz 3:  Unesite vrijednost a: 1  Unesite vrijednost b: ^C #(Ctrl+C)  Ostale iznimke! |

3.7.4.

|  |
| --- |
| **try**:  a = int(input(**"Unesite vrijednost a: "**))  b = int(input(**"Unesite vrijednost b: "**))  c = int(input(**"Unesite vrijednost c: "**))   rezultat = (a + b) / c  print(rezultat) **except** ValueError **as** e:  print(**"Iznimka - ValueError, opis:"**, e) **except** ZeroDivisionError **as** e:  print(**"Iznimka - ZeroDivisionError, opis:"**, e) **except**:  print(**"Ostale iznimke!"**) |
| Izlaz 1:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***broj***  Iznimka - ValueError, opis: invalid literal for int() with base 10: 'broj' |
| Izlaz 2:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***2***  Unesite vrijednost c: ***0***  Iznimka - ZeroDivisionError, opis: division by zero |

3.7.5.

|  |
| --- |
| **def** ucitajPaZbroji():  **try**:  a = int(input(**"Unesite vrijednost a: "**))  b = int(input(**"Unesite vrijednost b: "**))  c = int(input(**"Unesite vrijednost c: "**))  rezultat = a + b + c  print(**"Zbroj je:"**, rezultat)  **except**:  print(**"Iznimka je uhvaćena u funkciji!"**)  **raise   try**:  ucitajPaZbroji() **except**:  print(**"Iznimka je uhvaćena u glavnom programu!"**) |
| Izlaz 1:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***2***  Unesite vrijednost c: ***3***  Zbroj je: 6 |
| Izlaz 2:  Unesite vrijednost a: ***broj***  Iznimka je uhvaćena u funkciji!  Iznimka je uhvaćena u glavnom programu! |

3.7.6.

|  |
| --- |
| **def** ucitajPaZbroji():  **try**:  a = int(input(**"Unesite vrijednost a: "**))  b = int(input(**"Unesite vrijednost b: "**))  c = int(input(**"Unesite vrijednost c: "**))   rezultat = a + b + c  **except**:  print(**"Iznimka je uhvaćena u funkciji!"**)  **raise  else**:  print(rezultat)  **finally**:  print(**"Finally blok!"**)   **try**:  ucitajPaZbroji() **except**:  print(**"Iznimka je uhvaćena u glavnom programu!"**) |
| Izlaz 1:  Unesite vrijednost a: ***1***  Unesite vrijednost b: ***2***  Unesite vrijednost c: ***3***  6  Finally blok! |
| Izlaz 2:  Unesite vrijednost a: ***broj***  Iznimka je uhvaćena u funkciji!  Finally blok!  Iznimka je uhvaćena u glavnom programu! |

3.8.1.

Sintaksne, semantičke i logičke greške.

3.8.2.

U *Pythonu* sve iznimke nasljeđuju ugrađeni osnovni razred BaseException.

3.8.3.

U slučaju da korištena varijabla nije definirana podiže se iznimka NameError.

3.8.4.

Osnovni blok za obradu iznimaka tvore ključne riječi try i except.

3.8.5.

Prošireni blok za obradu iznimaka tvore ključne riječi else i/ili finally.

3.8.6.

Unutar except bloka, iznimke se hvataju i obrađuju.

3.8.7.

Blok else se izvodi u slučaju kada se unutar bloka try ne dogodi iznimka ili ako se on ne završi pozivom naredbe return, break ili continue.

3.8.8.

Blok finally se uvijek izvršava prije završetka obrade iznimke (bilo da se iznimka dogodila ili ne).

4.4.1.

|  |
| --- |
| **class** Vozilo:  **pass** v1 = Vozilo() v2 = Vozilo() v3 = Vozilo()  print(v1) print(v2) print(v3) |
| Izlaz:  <\_\_main\_\_.Vozilo object at 0x02960C50>  <\_\_main\_\_.Vozilo object at 0x02BBD9B0>  <\_\_main\_\_.Vozilo object at 0x02BBD990> |

4.4.2.

|  |
| --- |
| **class** Vozilo:  **def** vozi(self):  print(**"Vozi!"**)   **def** koci(self):  print(**"Koci!"**)   v1 = Vozilo() v2 = Vozilo() v3 = Vozilo()  v1.vozi() v1.koci()  v2.vozi() v2.koci()  v3.vozi() v3.koci() |
| Izlaz:  Vozi!  Koci!  Vozi!  Koci!  Vozi!  Koci! |

4.4.3.

|  |
| --- |
| **class** Vozilo:  **def** \_\_init\_\_(self, proizvodac, model, godiste):  self.proizvodac = proizvodac  self.model = model  self.godiste = godiste   **def** vozi(self):  print(**"Vozi!"**)   **def** koci(self):  print(**"Koci!"**) |

4.4.4.

|  |
| --- |
| **class** Vozilo:  **def** \_\_init\_\_(self, proizvodac, model, godiste):  self.proizvodac = proizvodac  self.model = model  self.godiste = godiste   **def** ispis(self):  print(**"Proizvodac:"**, self.proizvodac)  print(**"Model:"**, self.model)  print(**"Godiste:"**, self.godiste)   **def** vozi(self):  print(**"Vozi!"**)   **def** koci(self):  print(**"Koci!"**)   v1 = Vozilo(**"Proizvodac 1"**, **"Model X"**, **"2010"**) v2 = Vozilo(**"Proizvodac 2"**, **"Model Y"**, **"2018"**)  print(**"Vozilo 1:"**) v1.ispis() print(**"\nVozilo 2:"**) v2.ispis() |
| Izlaz:  Vozilo 1:  Proizvodac: Proizvodac 1  Model: Model X  Godiste: 2010  Vozilo 2:  Proizvodac: Proizvodac 2  Model: Model Y  Godiste: 2018 |

4.4.5.

|  |
| --- |
| **class** Student:  brojac = 0   **def** \_\_init\_\_(self):  self.ocjene = []   **def** novaOcjena(self, ocjena):  self.ocjene.append(ocjena)  Student.brojac += 1   **def** arSr(self):  suma = 0  broj = 0  **for** element **in** self.ocjene:  suma += element  broj += 1  **return** suma / broj   **def** brojOcjena(self, ocjena):  **return** self.ocjene.count(ocjena)   **def** brojSvihOcjena1():  **return** Student.brojac   @staticmethod  **def** brojSvihOcjena2():  **return** Student.brojac   s1 = Student() s2 = Student()  s1.novaOcjena(5) s1.novaOcjena(5) s1.novaOcjena(5) s1.novaOcjena(5) s1.novaOcjena(5)  s2.novaOcjena(4) s2.novaOcjena(3) s2.novaOcjena(2) s2.novaOcjena(2) s2.novaOcjena(2)  print(**"S1 - Ar. sredina je: "**, s1.arSr()) print(**"S1 - Broj ocjena (5) je:"**, s1.brojOcjena(5))  print(**"S2 - Ar. sredina je: "**, s2.arSr()) print(**"S2 - Broj ocjena (5) je:"**, s2.brojOcjena(5))  print(**"Ukupno ocjena:"**, Student.brojSvihOcjena1()) print(**"Ukupno ocjena:"**, s1.brojSvihOcjena2()) |
| Izlaz:  S1 - Ar. sredina je: 5.0  S1 - Broj ocjena (5) je: 5  S2 - Ar. sredina je: 2.6  S2 - Broj ocjena (5) je: 0  Ukupan broj ocjena je: 10  Ukupan broj ocjena je: 10 |

4.7.1.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.imeOsobe) o.imeOsobe = **"Perica"** print(o.imeOsobe) |
| Izlaz:  Ivan  Perica |

4.7.2.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.\_imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.\_imeOsobe) o.\_imeOsobe = **"Perica"** print(o.\_imeOsobe) |
| Izlaz:  Ivan  Perica |

4.7.3.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.\_\_imeOsobe) o.\_\_imeOsobe = **"Perica"** print(o.\_\_imeOsobe) |
| Izlaz:  Traceback (most recent call last):  File "test.py", line 7, in <module>  print(o.\_\_imeOsobe)  AttributeError: 'Osoba' object has no attribute '\_\_imeOsobe' |

4.7.4.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.\_Osoba\_\_imeOsobe) |
| Izlaz:  Ivan |

4.7.5.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   **def** getImeOsobe(self):  **return** self.\_\_imeOsobe   **def** setImeOsobe(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.getImeOsobe()) o.setImeOsobe(**"Perica"**) print(o.getImeOsobe()) |
| Izlaz:  Ivan  Perica |

4.7.6.

|  |
| --- |
| **class** Osoba:  **def** \_\_init\_\_(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   @property  **def** imeOsobe(self):  **return** self.\_\_imeOsobe   @imeOsobe.setter  **def** imeOsobe(self, imeOsobe):  self.\_\_imeOsobe = imeOsobe   o = Osoba(**"Ivan"**) print(o.imeOsobe) o.imeOsobe = **"Perica"** print(o.imeOsobe) |
| Izlaz:  Ivan  Perica |

4.11.1.

|  |
| --- |
| **class** Stavka:  **def** \_\_init\_\_(self, naziv, cijena):  self.\_naziv = naziv  self.\_cijenaNeto = cijena   **def** cijenaPdv(self):  **return** self.\_cijenaNeto \* 1.25   **class** Sok(Stavka):  **def** \_\_init\_\_(self, naziv, cijena, volumen):  super().\_\_init\_\_(naziv, cijena)  self.\_volumen = volumen   **def** ispis(self):  print(**"Naziv:"**, self.\_naziv)  print(**"Volumen:"**, self.\_volumen)  print(**"Cijena(bez PDV-a):"**,self.\_cijenaNeto)   **class** Hrana(Stavka):  **def** \_\_init\_\_(self, naziv, cijena, kalorije):  super().\_\_init\_\_(naziv, cijena)  self.\_kalorije = kalorije   **def** ispis(self):  print(**"Naziv:"**, self.\_naziv)  print(**"Kalorije:"**, self.\_kalorije)  print(**"Cijena(bez PDV-a):"**,self.\_cijenaNeto)   p = Sok(**"Naranča"**, 10, 0.5) h = Hrana(**"Pizza"**, 40, 150)  p.ispis() print(**"Cijena (s PDV-om):"**, p.cijenaPdv()) print(**"-------------"**) h.ispis() print(**"Cijena (s PDV-om):"**, h.cijenaPdv()) |
| Izlaz:  Naziv: Sok  Volumen: 0.5  Cijena (bez PDV-a): 10  Cijena (s PDV-om): 12.5  -------------  Naziv: Pizza  Kalorije: 150  Cijena (bez PDV-a): 40  Cijena (s PDV-om): 50.0 |

4.11.2.

|  |
| --- |
| **class** Stavka:  **def** \_\_init\_\_(self, naziv, cijena):  self.\_naziv = naziv  self.\_cijenaNeto = cijena   **def** cijenaPdv(self):  **return** self.\_cijenaNeto \* 1.25  **class** Sok(Stavka):  **def** \_\_init\_\_(self, naziv, cijena, volumen):  super().\_\_init\_\_(naziv, cijena)  self.\_volumen = volumen   **def** ispis(self):  print(**"Naziv:"**, self.\_naziv)  print(**"Volumen:"**, self.\_volumen)  print(**"Cijena(bez PDV-a):"**,self.\_cijenaNeto)   **def** cijenaPdv(self):  **return** self.\_cijenaNeto \* 1.15  **class** Hrana(Stavka):  **def** \_\_init\_\_(self, naziv, cijena, kalorije):  super().\_\_init\_\_(naziv, cijena)  self.\_kalorije = kalorije   **def** ispis(self):  print(**"Naziv:"**, self.\_naziv)  print(**"Kalorije:"**, self.\_kalorije)  print(**"Cijena(bez PDV-a):"**,self.\_cijenaNeto)  **def** cijenaPdv(self):  **return** self.\_cijenaNeto \* 1.05   p = Sok(**"Naranča"**, 10, 0.5) h = Hrana(**"Pizza"**, 40, 150)  p.ispis() print(**"Cijena (s PDV-om):"**, p.cijenaPdv()) print(**"-------------"**) h.ispis() print(**"Cijena (s PDV-om):"**, h.cijenaPdv()) |
| Izlaz:  Naziv: Sok  Volumen: 0.5  Cijena (bez PDV-a): 10  Cijena (s PDV-om): 11.5  -------------  Naziv: Pizza  Kalorije: 150  Cijena (bez PDV-a): 40  Cijena (s PDV-om): 42.0 |

4.11.3.

|  |
| --- |
| **from** math **import** pi  **class** GeometrijskiLikovi:  **def** opseg(a=**None**, b=**None**, c=**None**):  **if** a **is None and** b **is None and** c **is None**:  print(**"Dogodila se greška!"**)   **elif** a **is not None and** b **is None and** c **is None**:  print(2 \* a \* pi)   **elif** a **is not None and** b **is not None and** c **is None**:  print(2 \* a + 2 \* b)   **else**:  print(a + b + c)  GeometrijskiLikovi.opseg()  GeometrijskiLikovi.opseg(1)  GeometrijskiLikovi.opseg(1, 2)  GeometrijskiLikovi.opseg(1, 2, 3) |
| Izlaz:  Dogodila se greška!  6.283185307179586  6  6 |

4.14.1.

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Zivotinja(ABC):   @abstractmethod  **def** glasanje(self):  **pass** @abstractmethod  **def** pokret(self):  **pass   class** Pas(Zivotinja):   **def** glasanje(self):  print(**"Wau wau!"**)   **def** pokret(self):  print(**"Trcanje i hodanje!"**)   **class** Puz(Zivotinja):   **def** glasanje(self):  print(**"Ne glasa se!"**)   **def** pokret(self):  print(**"Puzanje!"**)   pas = Pas() puz = Puz()  print(**"Pas:"**) pas.glasanje() pas.pokret()  print(**"\nPuz:"**) puz.glasanje() puz.pokret() |
| Izlaz:  Pas:  Wau wau!  Trcanje i hodanje!  Puz:  Ne glasa se!  Puzanje! |

4.15.1.

Za kompleksnije programe bolje je koristiti objektno orijentirano programiranje.

4.15.2.

Osnovna cjelina zove se razred. Razred sadržava kompletnu strukturu i funkcionalnosti objekata.

4.15.3.

Konstruktor.

4.15.4.

Ne.

4.15.5.

Parametar self povezuje metodu s objektom.

4.15.6.

Varijablu razreda moguće je dohvatiti iz svih objekata i ona će uvijek imati jednu te istu vrijednost nevezano za to iz kojeg ju se objekta dohvaća, dok je varijabla objekta strogo vezana za neki konkretan objekt.

4.15.7.

public, protected, private.

4.15.8.

Javna varijabla vidljiva je i dohvatljiva iz svih dijelova programskoga kôda, dok je privatna varijabla dohvatljiva samo iz razreda/objekta unutar kojeg je definirana.

4.15.9.

Nasljeđivanje je princip kojim se definiranje nekoga razreda vrši korištenjem postojećega razreda.

4.15.10.

Kada dvije i više metoda (funkcija) imaju identično ime, ali različit broj parametara, to se zove preopterećenje (engl. *Overloading*).

4.15.11.

Nadjačavanje je situacija u kojoj metoda u izvedenom razredu nadjačava (mijenja) metodu iz baznog razreda.

5.5.1.

|  |
| --- |
| **def** ispis(sudionici, rezultati):  i = 0  **for** s **in** sudionici:  print(s, **":"**, rezultati[i])  i += 1   sud = [**"Ivica"**, **"Marica"**, **"Perica"**, **"Nika"**, **"Nikica"**, **"Tihana"**, **"Stjepan"**, **"Petra"**] rez = [10.11, 11.58, 10.08, 10.06, 10.77, 11.22, 11.55, 10.05]  print(**"Prije sortiranja: "**) ispis(sud, rez)  brojSudionika = len(rez) **for** i **in** range(brojSudionika):   minIndex = i  **for** j **in** range(i + 1, brojSudionika):  **if** rez[j] < rez[minIndex]:  minIndex = j   rez[i], rez[minIndex] = rez[minIndex], rez[i]  sud[i], sud[minIndex] = sud[minIndex], sud[i]  print(**"\n\nNakon sortiranja: "**) ispis(sud, rez) |
| Izlaz:  Prije sortiranja:  Ivica : 10.11  Marica : 11.58  Perica : 10.08  Nika : 10.06  Nikica : 10.77  Tihana : 11.22  Stjepan : 11.55  Petra : 10.05  Nakon sortiranja:  Petra : 10.05  Nika : 10.06  Perica : 10.08  Ivica : 10.11  Nikica : 10.77  Tihana : 11.22  Stjepan : 11.55  Marica : 11.58 |

5.5.2.

|  |
| --- |
| **def** ispis(sud, rez):  i = 0  **for** s **in** sud:  print(s, **":"**, rez[i])  i += 1   sud = [**"Ivica"**, **"Marica"**, **"Perica"**, **"Nika"**, **"Nikica"**, **"Tihana"**, **"Stjepan"**, **"Petra"**] rez = [10.11, 11.58, 10.08, 10.06, 10.77, 11.22, 11.55, 10.05]  print(**"Prije sortiranja: "**) ispis(sud, rez)  brojElemenata = len(rez) **for** i **in** range(brojElemenata - 1):   zamjena = **False  for** j **in** range(brojElemenata - 1 - i):  **if** rez[j + 1] < rez[j]:  rez[j], rez[j + 1] = rez[j + 1], rez[j]  sud[j], sud[j + 1] = sud[j + 1], sud[j]  zamjena = **True   if** zamjena == **False**:  **break** print(**"\n\nNakon sortiranja: "**) ispis(sud, rez) |
| Izlaz:  Prije sortiranja:  Ivica : 10.11  Marica : 11.58  Perica : 10.08  Nika : 10.06  Nikica : 10.77  Tihana : 11.22  Stjepan : 11.55  Petra : 10.05  Nakon sortiranja:  Petra : 10.05  Nika : 10.06  Perica : 10.08  Ivica : 10.11  Nikica : 10.77  Tihana : 11.22  Stjepan : 11.55  Marica : 11.58 |

5.6.1.

Ugrađenu metodu sort() nije moguće koristiti u slučaju da postoje, na primjer, dvije liste, u jednoj listi nalaze se imena osoba, a u drugoj listi nalaze se rezultati natjecanja (podaci su povezani preko zajedničkog indeksa). U tom slučaju potrebno je ručno implementirati algoritam koji će istovremeno sortirati obje liste.

5.6.2.

Na početku ovog algoritma pronalazi se najmanja vrijednost, tako pronađena najmanja vrijednost stavlja se na prvo mjesto, nakon toga traži se sljedeća najmanja vrijednost koja se stavlja na drugo mjesto i tako dalje sve do trenutka kada niz bude sortiran.

5.6.3.

Ovaj algoritam prolazi kroz niz te uspoređuje parove vrijednosti, tj. susjedne elemente. U slučaju da su ti elementi (susjedi) u neispravnom poretku, tada se njihove vrijednosti mijenjaju. Algoritam kroz nesortirani dio niza prolazi tako dugo dok niz ne postane sortiran.

5.6.4.

Algoritam zamjene susjednih elemenata moguće je poboljšati tako da se uvede dodatna varijabla u koju će biti zapisano je li se unutar nekog prolaza dogodila zamjena ili ne. Ako unutar nekog prolaza po nesortiranom dijelu niza nema zamjene, niz se proglašava sortiranim te se pomoću naredbe break prekida daljnje uspoređivanje susjednih elemenata.

6.4.1.

|  |
| --- |
| **class** NasumicniBroj:  \_\_singleton = **None   def** \_\_init\_\_(self):  **if** NasumicniBroj.\_\_singleton == **None**:  NasumicniBroj.\_\_singleton = self  NasumicniBroj.\_\_sljedeci = 67  **else**:  **raise** Exception(**"Singleton!"**)   @staticmethod  **def** getInstance():  **if** NasumicniBroj.\_\_singleton == **None**:  NasumicniBroj()  **return** NasumicniBroj.\_\_singleton   **def** sljedeci(self):  x = self.\_\_sljedeci  self.\_\_sljedeci = 9845612 % x + 17 \* 7  **return** x   g1 = NasumicniBroj.getInstance() g2 = NasumicniBroj.getInstance()  print(g1.sljedeci()) print(g2.sljedeci()) print(g1.sljedeci()) print(g2.sljedeci()) print(g1.sljedeci()) print(g2.sljedeci()) print(g1.sljedeci()) print(g2.sljedeci()) |
| Izlaz:  67  148  179  194  231  280  371  133 |

6.4.2.

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Citati:  **def** \_\_init\_\_(self):  self.\_\_pretplatnici = set()  self.\_\_danasnjiCitat = **None   def** noviPretplatnik(self, o):  self.\_\_pretplatnici.add(o)  o.izvor = self   **def** \_\_obavijesti(self):  print(**"\n> Novi podaci:"**)  **for** o **in** self.\_\_pretplatnici:  o.osvjezi(self.\_\_danasnjiCitat)   @property  **def** danasnjiCitat(self):  **return** self.\_\_danasnjiCitat   @danasnjiCitat.setter  **def** danasnjiCitat(self, value):  self.\_\_danasnjiCitat = value  self.\_\_obavijesti()   **class** Pretplatnik(ABC):  **def** \_\_init\_\_(self):  self.\_\_izvor = **None** self.\_\_danasnjiCitat = **None** @abstractmethod  **def** osvjezi(self, v):  **pass   class** KonkretanPretplatnik(Pretplatnik):  **def** \_\_init\_\_(self, n):  super().\_\_init\_\_()  self.\_\_naziv = n   **def** osvjezi(self, v):  self.\_\_danasnjiCitat = v  print(**" Pretplatnik"**, self.\_\_naziv,  **"je primio novi citat"**)  print(**" Citat:"**, self.\_\_danasnjiCitat)   c = Citati()  p1 = KonkretanPretplatnik(**"Ivica"**) p2 = KonkretanPretplatnik(**"Marica"**)  c.noviPretplatnik(p1) c.noviPretplatnik(p2)  c.danasnjiCitat = **"Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci."** c.danasnjiCitat = **"Budi promjena koju želiš vidjeti u svijetu."** c.danasnjiCitat = **"Nikada nije prekasno da budete ono što biste mogli biti."** |
| Izlaz:  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci.  Pretplatnik Ivica je primio novi citat  Citat: Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci.  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Budi promjena koju želiš vidjeti u svijetu.  Pretplatnik Ivica je primio novi citat  Citat: Budi promjena koju želiš vidjeti u svijetu.  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Nikada nije prekasno da budete ono što biste mogli biti.  Pretplatnik Ivica je primio novi citat  Citat: Nikada nije prekasno da budete ono što biste mogli biti. |

6.4.3.

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Citati:  \_\_singleton = **None   def** \_\_init\_\_(self):  **if** Citati.\_\_singleton == **None**:  Citati.\_\_singleton = self  self.\_\_pretplatnici = set()  self.\_\_danasnjiCitat = **None  else**:  **raise** Exception(**"Singleton!"**)   @staticmethod  **def** getInstance():  **if** Citati.\_\_singleton == **None**:  Citati()  **return** Citati.\_\_singleton   **def** noviPretplatnik(self, o):  self.\_\_pretplatnici.add(o)  o.izvor = self   **def** \_\_obavijesti(self):  print(**"\n> Novi podaci:"**)  **for** o **in** self.\_\_pretplatnici:  o.osvjezi(self.\_\_danasnjiCitat)   @property  **def** danasnjiCitat(self):  **return** self.\_\_danasnjiCitat   @danasnjiCitat.setter  **def** danasnjiCitat(self, value):  self.\_\_danasnjiCitat = value  self.\_\_obavijesti()   **class** Pretplatnik(ABC):  **def** \_\_init\_\_(self):  self.\_\_izvor = **None** self.\_\_danasnjiCitat = **None** @abstractmethod  **def** osvjezi(self, v):  **pass**  **class** KonkretanPretplatnik(Pretplatnik):  **def** \_\_init\_\_(self, n):  super().\_\_init\_\_()  self.\_\_naziv = n   **def** osvjezi(self, v):  self.\_\_danasnjiCitat = v  print(**" Pretplatnik"**, self.\_\_naziv,  **"je primio novi citat"**)  print(**" Citat:"**, self.\_\_danasnjiCitat)   c1 = Citati.getInstance() c2 = Citati.getInstance()  p1 = KonkretanPretplatnik(**"Ivica"**) p2 = KonkretanPretplatnik(**"Marica"**)  c1.noviPretplatnik(p1) c1.noviPretplatnik(p2)  c2.danasnjiCitat = **"Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci."** c2.danasnjiCitat = **"Budi promjena koju želiš vidjeti u svijetu."** c2.danasnjiCitat = **"Nikada nije prekasno da budete ono što biste mogli biti."** |
| Izlaz:  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci.  Pretplatnik Ivica je primio novi citat  Citat: Brod je siguran u luci, ali brod nije izgrađen da bi bio u luci.  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Budi promjena koju želiš vidjeti u svijetu.  Pretplatnik Ivica je primio novi citat  Citat: Budi promjena koju želiš vidjeti u svijetu.  > Novi podaci:  Pretplatnik Marica je primio novi citat  Citat: Nikada nije prekasno da budete ono što biste mogli biti.  Pretplatnik Ivica je primio novi citat  Citat: Nikada nije prekasno da budete ono što biste mogli biti. |

6.5.1.

Algoritam definira korake koje je potrebno napraviti kako bi se postigao neki cilj, dok je oblikovni obrazac uputa prema kojoj je moguće riješiti neki problem na najbolji mogući način.

6.5.2.

"Jedinstveni objekt" omogućava samo jedno kreiranje objekta nekoga razreda, što znači da ne može postojati više različitih objekata jednoga te istog razreda.

6.5.3.

Oblikovni obrazac "promatrač" omogućava da se na jednostavan način obavijeste svi objekti koji su pretplaćeni na obavještavanje da je došlo do promjene podatka unutar glavnog objekta.

Literatura

1. A. B. Downey, *Think Python*, O'Reilly Media, 2012.
2. Aleksandar Stojanović, *Elementi računalnih programa*, Element, 2012.
3. David Beazley, Brian Jones, *Python Cookbook, 3rd Edition*, O'Reilly Media, 2013.
4. FER, <http://www.zemris.fer.hr/~ssegvic/pubs/ooup1Principles.pdf> dohvaćeno 15.06.2019.
5. G. van Rossum, *Introduction to Python 3; Documentation for Python*, SoHoBooks, 2010.
6. John Paul Mueller, *Beginning Programming with Python For Dummies*, Wiley, 2014.
7. *Learn Python*, <https://www.learnpython.org/>, dohvaćeno 11.02.2019.
8. Leo Budin, Predrag Brođanac, Zlatka Markučič, Smiljana Perić, Dejan Škvorc, Magdalena Babić, *Računalno razmišljanje i programiranje u Pythonu*, Element, 2017.
9. Leo Budin, Predrag Brođanac, Zlatka Markučič, Smiljana Perić, *Napredno rješavanje problema programiranjem u Pythonu*, Element, 2013.
10. Leo Budin, Predrag Brođanac, Zlatka Markučič, Smiljana Perić, *Rješavanje problema programiranjem u Pythonu*, Element, 2012.
11. Mark Lutz, *Programming Python, 4th Edition*, O'Reilly Media, 2010.
12. Marko Hruška, *Osnove programiranja (Python)*, Srce, 2018.
13. Predrag Brođanac, Leo Budin, Zlatka Markučič, Smiljana Perić, *Izrada primjenskih programa u Pythonu*, Element, 2017.
14. *Python 3.x Documentation*, <https://docs.python.org/3/>, dohvaćeno 11.02.2019.
15. *Python Notes for Professionals*, GoalKicker, <https://goalkicker.com/PythonBook/>
16. *Python Tutorials*, <https://pythonspot.com/>, dohvaćeno 11.02.2019.
17. Toma Rončević, *Uvod u programiranje*, 2016.
18. Zoran Kalafatić, Antonio Pošćić, Siniša Šegvić, Julijan Šribar, *Python za znatiželjne*, Element, 2016.

***Bilješke:***

1. Sadržaj preuzet s: FER, http://www.zemris.fer.hr/~ssegvic/pubs/ooup1Principles.pdf [↑](#endnote-ref-1)