Podloge za stručno usavršavanje učitelja osnovnih škola
za domenu
Računalno razmišljanje i programiranje

05
Ispitivanje uvjeta, donošenje odluka u programima

Uz dozvolu izdavača korišteni su sadržaji iz priručnika:

Leo Budin, Predrag Brođanac, Zlatka Markučič
Smiljana Perić, Dejan Škvorc, Magdalena Babić

Računalno razmišljanje i programiranje u Pythonu
Element, Zagreb, 2017

Donošenje odluka, operatori usporedbe

Kao i svakodnevnome životu, tako i u programiranju ponekad moramo odlučiti hoćemo li nešto napraviti na jedan ili drugi način. Ovisno o uvjetima koji nastupe, naše se ponašanje razlikuje. Primjerice, ako je vani hladno, oblačimo kaput, a ako je toplo, onda izlazimo bez kaputa. Ako je danas nedjelja, onda spavamo do 9 sati, a ako je neki drugi dan, ustajemo u 7 sati. U ovim primjerima vidimo dva važna elementa:

- uvjet koji ispitujemo
- odluka koju donosimo ako je uvjet ispunjen.

Uvjeti koje ispitujemo u gornjim primjerima su vanjska temperatura i dan u tjednu. Odluka koju donosimo vezano uz ispitivanje vanjske temperature je izlazak s kaputom ili bez njega. Odluka koju donosimo vezano uz ispitivanje dana u tjednu je vrijeme do kada možemo spavati.

U programskim jezicima, pa tako i u *Pythonu*, postoje naredbe kojima se ostvaruje ispitivanje uvjeta i donošenje odluke. Ovisno o tome koju odluku donesemo, a to ovisi o tome je li uvjet koji ispitujemo ispunjen ili nije, izvodit će se različite naredbe programa. Takav oblik izvođenja programa naziva se uvjetno izvođenje dijela programa, a takve naredbe nazivamo uvjetnim naredbama.

Uvjeti koje naredbe ispituju moraju biti tako oblikovani da daju jednoznačni odgovor je li neki uvjet ispunjen ili nije.

U sljedećoj tablici prikazani su uobičajeni matematički operatori za usporedbe, njihovi nazivi i odgovarajuće oznake za te operatore u Pythonu

Operator usporedbe u matematici		Operator usporedbe u *Pythonu*	
simbol	naziv	zapis	treba utipkati
<	manje	<	znak: <
≤	manje ili jednako	<=	dva znaka: < i =
=	jednako	==	dvaput znak: =
≥	veće ili jednako	>=	dva znaka: > i =
>	veće	>	znak: >
≠	različito	!=	dva znaka: ! i =

Slično kao i aritmetički operatori, ovi operatori djeluju na dvije vrijednosti. Uporabom operatora dobiva se odgovor na pitanje o njihovu odnosu. Primjerice, uporaba operanda > daje odgovor na pitanje je li vrijednost lijeve strane veća od vrijednosti s desne strane. Ako je odgovor potvrdan, tada će vrijednost izraza biti `True` (engl. *true* – istina, istinito), a ako nije, vrijednost izraza bit će `False` (engl. *false* – neistinito, lažno). Dakle, izrazi u kojim se rabe operatori usporedbe daju samo dvije vrijednosti: `True` ili `False`. Takve izraze nazivamo i **uvjetima**. Ako je vrijednost izraza jednaka `True`, kažemo da je uvjet ispunjen, a ako je vrijednost `False`, govorimo da uvjet nije ispunjen.

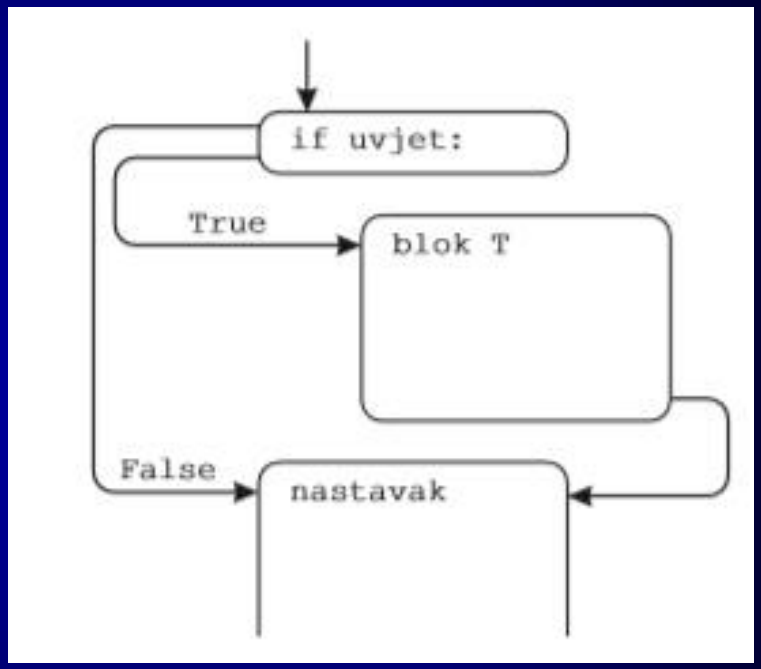
Pogledajmo u interaktivnom sučelju nekoliko primjera

```
>>> a = 8
>>> 3 < a
True
>>> 3 == a
False
>>> 3 >= a
False
>>> 3 != a
True
```

Grananje naredbom `if`

Naredba `if` (engl. *if* – ako, ako je) sastoji se od ključne riječi `if`, izraza čija se istinitost ispituje (uvjet) i dvotočke. Nakon te naredbe slijedi blok naredbi koji će se izvesti ako je vrijednost izraza istinita tj. ako je uvjet ispunjen. Blok naredbi mora se pisati s uvlakom kao kod definiranja funkcije petlje koja započinje ključnom riječi `for`. Taj se blok naredbi (nazovimo ga `blok T`) može sastojati i od samo jedne naredbe! Nakon obavljanja bloka naredbi `T` program će se nastaviti izvođenjem prve naredbe u nastavku programa.

Ako vrijednost izraza nije istinita, `blok T` se neće izvoditi i program će odmah nastaviti s izvođenjem nastavka.



Pogledajmo primjer u kojem ćemo uz naredbu grananja rabiti još i naredbu ponavljanja

```
>>> for i in range(1,11):                    #4
        if i % 2 == 0:                       #5
            print(i, end=' ')                #6

2 4 6 8 10
```

U petlji `for` (#4) varijabla petlje `i` poprimat će redom vrijednosti: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Za svaku generiranu vrijednost u naredbi `if` ispituje se je li ostatak dijeljenja s dva jednak nuli, tj. je li broj paran (#5). Blok naredbi sastoji se samo od naredbe za ispis (#6) koja će se izvesti samo ako je broj paran. S obzirom na to da smo u pozivu funkcije `print()` promijenili postavljenu vrijednost parametra `end`, brojevi će biti ispisivani bez prelaska u novi red.

Ako u gornjem primjeru promijenimo operator usporedbe u naredbi `if`, iz == u != dobit ćemo ispis neparnih brojeva od 1 do 9.

```
>>> for i in range(1,11):
        if i % 2 != 0:
            print(i, end=' ')

1 3 5 7 9
```

Primjer:

Iz matematike nam je poznato da su svi prirodni brojevi počevši od broja 2 djeljivi sa 1 i sa samim sobom. Brojevi koji su djeljivi samo sa 1 i sa samim sobom nazivaju se prostim brojevima.

Oni brojevi koji su djeljivi i s nekim drugim brojevima su složeni brojevi. Ako je prirodni broj djeljiv s nekim drugim brojem, onda je ostatak pri dijeljenju jednak 0. Ta nam činjenica omogućuje jednostavno određivanje brojeva koji su djeljivi s nekim brojem. Provjerimo u interaktivnom sučelju algoritam kojim ćemo ispisati s kojim je sve brojevima djeljiv broj `a`*:*

```
>>> a = 128
>>> for k in range(1, a + 1):                    #7
        if a % k == 0:                           #8
            print(k, end=' ')                    #9

1 2 4 8 16 32 64 128
>>> a = 127
>>> for k in range(1, a + 1):
        if a % k == 0:
            print(k, end=' ')

1 127
```

Za zadani broj, primjerice 128, u petlji `for` *(#7) ispitivat će se ostatci dijeljenja sa svim brojevima* `k` *iz intervala [1, 128]. Ako je ostatak dijeljenja jednak 0, u naredbi* `if` *(#8) izvest će se naredba ispisa (#9) te će se ispisati broj* `k`*.*

Vidimo da je broj 128 složeni broj i da je, uz 1 i 128, djeljiv s brojevima: 2, 4, 8, 16, 32 i 64.

Možemo zaključiti da je 127 prosti broj jer smo jednakim ispitivanjem ustanovili da je djeljiv samo sa 1 i sa samim sobom.

Primjer:

Ispišimo djelitelje svih brojeva iz nekog intervala. U našem ćemo primjeru to načiniti za brojeve iz intervala [2, 10]:

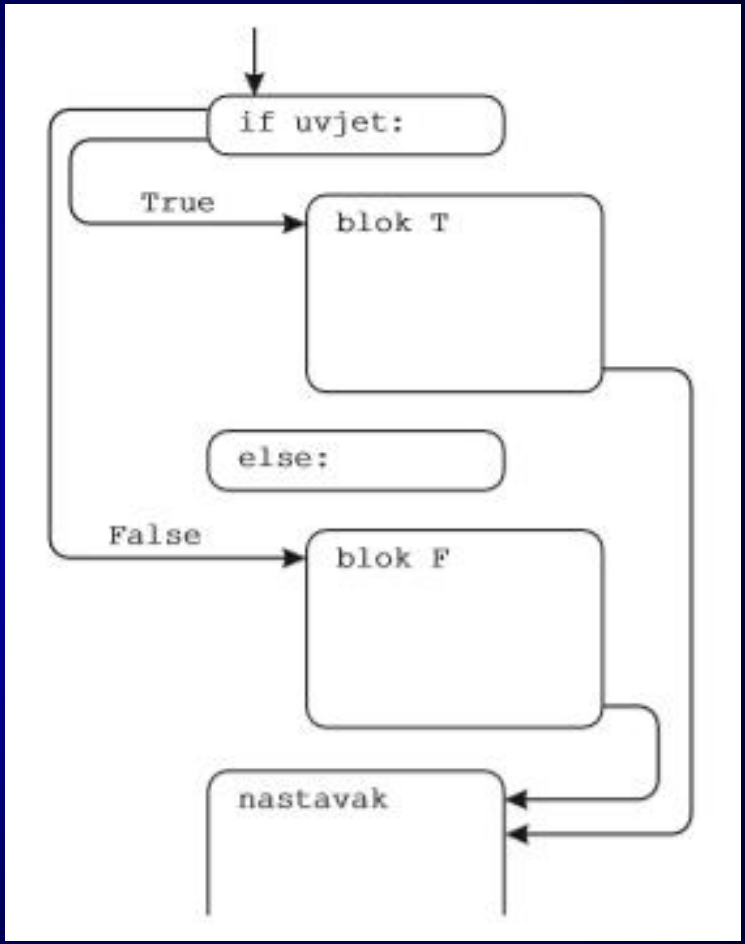
```
>>> for i in range(2,11):                                          #10
        print('Broj {0:2d} djeljiv je s brojevima'.format(i), end=': ')  #11
        for k in range(1, i+1):                                    #12
            if i % k == 0:
                print(k, end=' ')                                  #13
        print()

Broj  2 djeljiv je s brojevima: 1 2
Broj  3 djeljiv je s brojevima: 1 3
Broj  4 djeljiv je s brojevima: 1 2 4
Broj  5 djeljiv je s brojevima: 1 5
Broj  6 djeljiv je s brojevima: 1 2 3 6
Broj  7 djeljiv je s brojevima: 1 7
Broj  8 djeljiv je s brojevima: 1 2 4 8
Broj  9 djeljiv je s brojevima: 1 3 9
Broj 10 djeljiv je s brojevima: 1 2 5 10
```

U gornjem nizu naredbi definirali smo dvije petlje `for` (#10) i (#11). U prvoj petlji odredili smo interval brojeva za koje želimo utvrditi djeljivost, a u drugoj za dani broj `i` određen prvom petljom ispitujemo djeljivost s brojevima od 1 do vrijednosti varijable `i`. Uočimo dvije male posebnosti. U prvom redu, u naredbi (#12) nalazimo da je vrijednost parametra `end=': '`, tako da će znak : i nakon njega praznina biti nadovezani na string (to smo mogli postići i tako da znak : dodamo na kraju stringa i napišemo `end=' '`). Druga posebnost je dodatna naredba (#13) kojom pozivamo funkciju `print()` bez ikakvih parametara. Ona nam je potrebna kako bi ispis za novu vrijednost `i` u nadređenoj petlji počeo u novom redu!

Grananje složenom naredbom `if-else`

U jednostavnoj naredbi `if`, ako je vrijednost uvjeta `True`, izvest će se blok naredbi `blok T`, ako je rezultat uvjeta `False`, izvest će se blok naredbi `nastavak`. U nekim situacijama kada je rezultat uvjeta `False`, bit će potrebno da se izvede neki drugi blok naredbi `blok F`. Nakon izvođenja jednog od blokova naredbi (`bloka T` ili `bloka F`) program će nastaviti izvoditi naredbe nastavka.

Takvo ponašanje programa omogućuje nam kombinacija naredbi `if-else` (engl. *else* – inače). Iza ključne riječi `if` treba (s uvlakom) napisati blok naredbi `blok T` koji će se izvoditi ako je uvjet ispunjen. Nakon toga se bez uvlake (poravnato s `if`) mora napisati ključna riječ `else` i iza nje (s uvlakom) dolazi `blok F`.



Provjerimo djelovanje složene naredbe `if-else` u interaktivnom sučelju na primjeru funkcije koja će provjeravati jesu li dva broja jednaka:

```
>>> def usporedba(a, b):
        if a == b:
            print('Brojevi su jednaki')                    #1
        else:
            print('Brojevi su različiti')                  #2

>>> usporedba(5, 7)
Brojevi su različiti
>>> usporedba(7, 7)
Brojevi su jednaki
```

U funkciji `usporedba()` izvest će se naredba (`#1`) ako je rezultat uvjeta `a == b` `True`, odnosno naredba (`#2`) ako je `False`. U to se možemo uvjeriti pozivajući funkciju s različitim vrijednostima ulaznih parametara.

Primjer:

Pogledajmo kako bi mogao izgledati program koji će nam generirati tablicu prostih brojeva unutar nekog zadanog intervala:

```
#Ispis tablice prostih brojeva - program_5_1.py

def prosti(a):                                                    #3
    '''Funkcija vraća
            True - kada je broj a prosti broj i
            False - kada je a složeni broj
    '''
    broj = 0                                                      #4
    for i in range(1, a + 1):
        if a % i == 0:
            broj += 1                                             #5
    if broj == 2:                                                 #6
        return True                                               #7
    else:
        return False                                              #8

def main():
    dg = int(input('Upiši donju granicu intervala:  '))           #9
    gg = int(input('Upiši gornju granicu intervala: '))           #10
    br = 0                                                        #11
    print('\nProsti brojevi u zadanom intervalu [{},{}] su:\n'.format(dg, gg))
    for i in range(dg, gg + 1):
        if prosti(i):
            print(i, end=' ')
            br += 1
    print('\n\nU zadanom intervalu ima {} prostih brojeva.'.format(br))  #12

main()
```

Funkcija `prosti(a)` (#3) vratit će vrijednost `True` ako je broj `a` prost broj i vrijednost `False` ako je on složen. Vidimo da je u definiciji funkcije na dva mjesta napisana naredba `return`, no svaki puta kada pozovemo funkciju, izvest će se samo jedna od njih.

Naredbom (#4) postavili smo brojilo `broj` u nulu. Brojilo će se povećavati za jedan uvijek kada pronađemo da varijabla petlje `i` dijeli broj `a` bez ostatka (#5). Ako je po završetku petlje brojilo (#6) doseglo broj 2, broj je prost i izvest će se naredba (#7). U suprotnom, bit će izvedena naredba (#8).

U glavnoj funkciji zatražit ćemo upisivanje donje i gornje granice intervala (#9) i (#10) za koji želimo odrediti proste brojeve. Osim ispisa prostih brojeva, želimo ih prebrojiti i ispisati koliko ih ima u tom intervalu. Zbog toga smo uveli brojilo `br` (#11) koje će povećavati vrijednost za 1 svaki puta kad ispišemo prosti broj.

Uočimo da će naredbom (#12) funkcija `print()` najprije zbog stringa `'\n\n'` prijeći u novi redak i zatim preskočiti još jedan redak.

Želimo li ispisati proste brojeve iz intervala [1, 100] nakon pokretanja programa utipkat ćemo za donju granicu broj 1, a za gornju granicu broj 100 i dobiti sljedeći ispis:

```
Upiši donju granicu intervala:  1
Upiši gornju granicu intervala: 100

Prosti brojevi u zadanom intervalu [1,100] su:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

U zadanom intervalu ima 25 prostih brojeva.
```

Isto tako možemo ispisati i tablicu prostih brojeva primjerice u intervalu [1, 1000]:

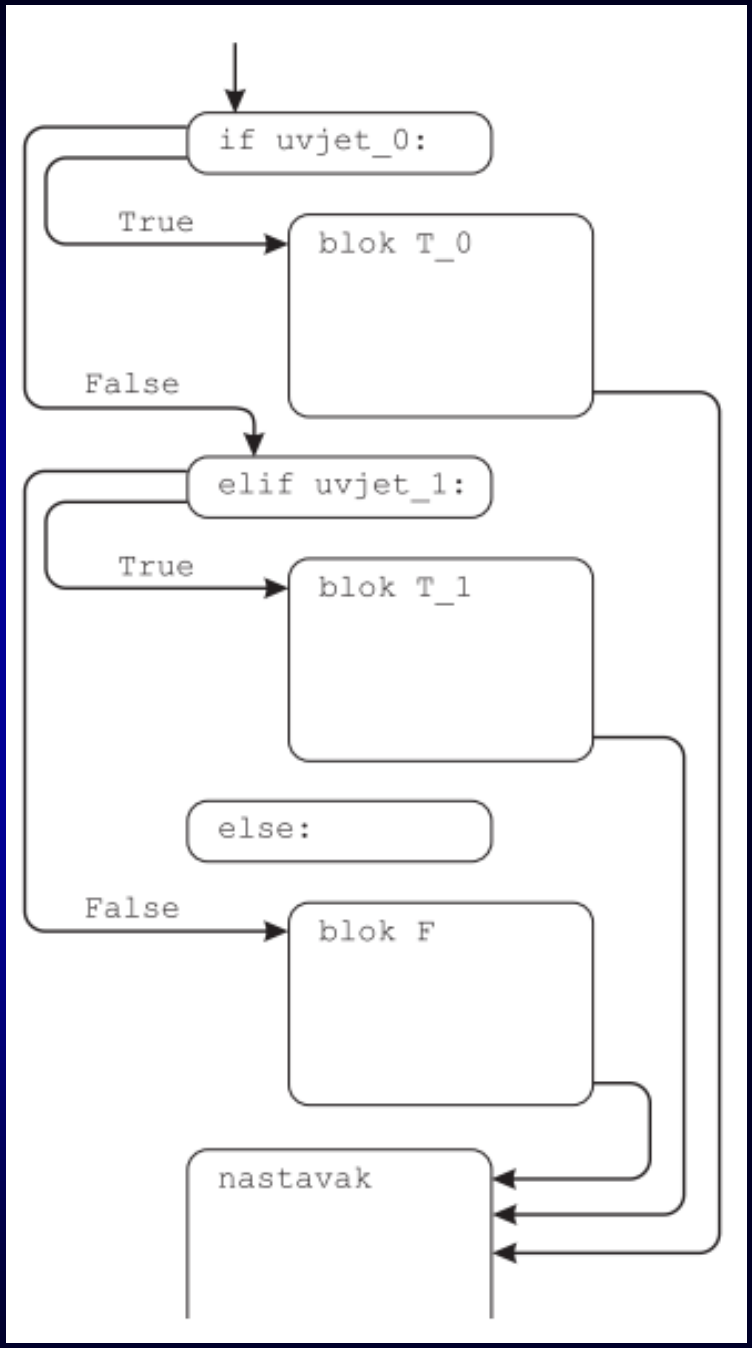
```
Upiši donju granicu intervala:  1
Upiši gornju granicu intervala: 1000

Prosti brojevi u zadanom intervalu [1,1000] su:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311
313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431
433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557
563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661
673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
941 947 953 967 971 977 983 991 997

U zadanom intervalu ima 168 prostih brojeva.
```

Grananje složenom naredbom

if-elif-else

Često se pri rješavanu nekih zadataka pojavljuje potreba ispitivanja više uvjeta. To možemo postići uzastopnom uporabom prethodno opisanih naredbi **if** i **if-else**. No taj se postupak može pojednostavniti ako primijenimo složenu naredbu **if-elif-else** koja na objedinjeni način ispituje više uvjeta. Tijek izvođenja ove složene naredbe s dva uvjeta prikazuje slika 5.3.

Ako je `uvjet_0` ispunjen, izvest će se blok naredbi `blok T_0` i zatim izvođenje prelazi na `nastavak`. Naredbe programa koje čine `blok T_1` i `blok F` se preskaču. Ako `uvjet_0` nije ispunjen, sljedeći se ispituje `uvjet_1`. Ako je `uvjet_1` ispunjen, izvršava se blok `T_1` i zatim izvođenje prelazi na `nastavak`. Naredbe programa koje čine `blok F` opet se preskaču. Tek ako nije ispunjen niti `uvjet_0` niti `uvjet_1`, izvodi se `blok F` i zatim izvođenje prelazi na `nastavak`. Važno je uočiti da se po izvršenju bilo kojeg od uvjetnih blokova izvođenje programa nastavlja blokom `nastavak`.



Na ovaj se način može ispitivati i veći broj uvjeta. Pritom za svaki od njih treba dodati jednu novu naredbu **elif** `uvjet`.

Naredba **else** i pripadni `blok F` može se i izostaviti iz lanca **if-elif** ako pri rješavanju nekog zadatka nisu potrebni (slično kao kod jednostavne naredbe **if**).

Pogledajmo djelovanje složene naredbe **if-elif-else** na sljedećem jednostavnom primjeru u interaktivnom sučelju:

```
>>> for i in range(20):
        if i < 4:
            y = 1
        elif i < 8:
            y = 2
        elif i < 12:
            y = 3
        elif i < 16:
            y = 4
        else:
            y = 5
        print(y, end=' ')

1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5
```

U prva četiri prolaza kroz petlju ispunjen je uvjet `i < 4` tako da `y` poprima vrijednost 1, u sljedeća četiri prolaza vrijednost 2 i tako redom do vrijednosti 4. U zadnja četiri prolaza niti jedan od uvjeta nije ispunjen te će se izvesti naredba `y = 5` nakon ključne riječi **else**.

Primjer:

Pri provjeri znanja učenik može ostvariti najviše 80 bodova. Prolaznu ocjenu dobiva učenik s najmanje 40 bodova. Ocjenu odličan dobiva učenik sa 70 i više bodova. Ostale ocjene dodjeljuju se u skladu s ostvarenim bodovima na način opisan dokumetacijskim stringom u sljedećem programu:

```
#Dodjela ocjena temeljem ostvarenih bodova - program_5_2.py

def ocjena(bodovi):
    '''Ocjene se dodjeljuju na temelju ostvarenih bodova na
    provjeri zanja. Najveći mogući broj bodova je b = 80.
    Ocjene se određuju na sljedeći način:
        odličan      bodovi >= 70
        vrlo dobar   60 <= bodovi < 70
        dobar        50 <= bodovi < 60
        dovoljan     40 <= bodovi < 50
        nedovoljan   bodovi < 40'''
    if bodovi >= 70:
        ocjena = 'odličan'
    elif bodovi >= 60:
        ocjena = 'vrlo dobar'
    elif bodovi >= 50:
        ocjena = 'dobar'
    elif bodovi >= 40:
        ocjena = 'dovoljan'
    else:
        ocjena = 'nedovoljan'
    return ocjena

def main():
    ime = input('Upisati ime : ')
    bodovi = int(input('Upisati broj bodova: '))
    print('{} ima ocjenu {}!'.format(ime, ocjena(bodovi)))

main()
```

Logičke varijable, tip `bool`

U svakodnevnom se životu kaže da je nešto logično ako se to može obrazložiti nekim dokazima. Osnovni pojam u logici je **logički sud**. Pod tim se nazivom podrazumijeva svaka tvrdnja koja se ocjenjuje samo s gledišta istinitosti i lažnosti. Valjani logički sud mora se oblikovati tako da se može jednoznačno odrediti je li on **istinit** ili **lažan**.

Prema tome, naši usporedbeni izrazi su zapravo logički sudovi. Zbog toga se vrijednosti `True` i `False` zovu i **logičke vrijednosti**. Za istraživanje sudova razvijena je posebna grana matematike – **matematička logika**. Osnove matematičke logike čini algebra sudova poznata i pod nazivima **logička algebra** i **Booleova algebra**[1]. Zbog toga se logičke vrijednosti nazivaju i Booleovim vrijednostima pa u *Python*u one pripadaju tipu `bool`.

1 George Boole (1815. – 1864.) britanski matematičar, prvi je uveo matematičku simboliku za zapisivanje logičkih izraza

Dosad smo upoznali tipove `int` i `str`. Tip `bool` je treći tip podataka koji upoznajemo. U interaktivnim sučelju uz pomoć funkcije `type()` možemo saznati kojeg je tipa pojedina varijabla odnosno vrijednost te varijable:

```
>>> a = 15                          #1
>>> b = '17'                        #2
>>> c = 15 < 17                     #3
>>> type(a)                         #4
<class 'int'>
>>> type(b)                         #5
<class 'str'>
>>> type(c)                         #6
<class 'bool'>
```

U naredbama (#1), (#2) i (#3) pridijelili smo varijablama vrijednosti određenog tipa. Naredbama (#4), (#5) i (#6) saznat ćemo da one pripadaju klasama `int`, `str` i `bool`. Riječ `class` treba za naše potrebe protumačiti kao tip. Vidimo da je varijabla `c` tipa `bool`.

Sjetimo se da postoje funkcije `int()` i `str()` kojima smo pretvarali vrijednosti tipa `str` u tip `int` i obrnuto vrijednosti tipa `int` u tip `str`.

Slično postoji i funkcija `bool()` koja pretvara vrijednosti drugih tipova u tip `bool`. Provjerimo u interaktivnom sučelju kako ta funkcija djeluje na vrijednosti tipa `int`:

```
>>> bool(15)                                                    #7
True
>>> bool(-100)                                                  #8
True
>>> bool(1)                                                     #9
True
>>> bool(0)                                                    #10
False
```

Iz navedenih naredbi možemo zaključiti da će funkcija `bool()` vratiti logičku vrijednost `True` odnosno `False` ovisno o vrijednosti argumenata koji je cijeli broj. Funkcija će vratiti vrijednost `False` (#10) ako je vrijednost tipa `int` jednaka 0, dok će za sve ostale cjelobrojne vrijednosti vratiti logičku vrijednost `True` (#7), (#8), (#9).

To znači da umjesto `if x != 0` možemo jednostavnije pisati `if x`. Na oba načina dobit ćemo iste rezultate:

```
>>> for x in range(-5, 5):
        if x != 0:                                              #11
            print('{0:3d}'.format(x), end=' ')

-5  -4  -3  -2  -1   1   2   3   4
>>> for x in range(-5, 5):
        if x:                                                   #12
            print('{0:3d}'.format(x), end=' ')

-5  -4  -3  -2  -1   1   2   3   4
```

Pogledajmo kako će funkcija `bool()` djelovati na vrijednosti tipa `str`:

```
>>> bool('a')                                                    #13
True
>>> bool('abc')                                                  #14
True
>>> bool('abcčćddefghijklmnošprsštuvzž')                         #15
True
>>> bool('  ')                                                   #16
True
>>> bool('0')                                                    #17
True
>>> bool('')                                                     #18
False
```

Iz primjera (#13), (#14) i (#15) vidimo da će za string bilo koje duljine funkcija `bool()` vratiti vrijednost `True`. Iz naredbi (#16) i (#17) je vidljivo da to vrijedi i za string koji se sastoji od jedne jedine praznine ili znamenke. Jedino za prazni string (koji pišemo kao dva navodnika napisana jedan uz drugi) funkcija vraća vrijednost `False` (#18).

Pogledajmo u interaktivnom sučelju što vraćaju funkcije `int()` i `str()` za vrijednosti `True` i `False`:

```
>>> int(True)                                                    #19
1
>>> int(False)                                                   #20
0
>>> str(True)                                                    #21
'True'
>>> str(False)                                                   #22
'False'
```

Rezultati naredbi (#19) i (#20) pokazuju da funkcija `int()` za vrijednost `True` vraća broj `1` te za vrijednost `False` vraća broj `0`. To se podudara sa suprotnom pretvorbom tipa `int` u tip `bool` u naredbama (#7) do (#10) – svaki broj različit od nule (pa tako i broj `1`) vratit će se `True` dok će se za broj `0` vratiti vrijednost `False`. To znači da ćemo dobiti:

```
>>> bool(int(True))
True
>>> bool(int(False))
False
```

Međutim, funkcija `str()` prevest će vrijednosti varijabli `True` i `False` (#21) i (#22) u stringove `'True'` i `'False'` koji se funkcijom `bool()` ne mogu ispravno prevesti natrag u logičke vrijednosti, jer je:

```
>>> bool(str(True))
True
>>> bool(str(False))
True
```

Uređivanje ispisa vrijednosti tip `int`, `str` *i* `bool` *metodom* `format()`

Prije nego nastavimo pisati neke nove programe bit će korisno upoznati još neke mogućnosti metode `format()` koje će nam pomoći pri uređivanju ispisa u našim programima.

```
>>> print('{:5d}/'.format(1))                    #1
    1/
>>> print('{:5d}/'.format(12))                   #2
   12/
>>> print('{:5d}/'.format(12345))                #3
12345/
>>> print('{:5d}/'.format(123456789))            #4
123456789/
```

U svim naredbama od (#1) do (#4) predviđeno je polje širine 5 znakova za ispis broja. Na kraju ispisnog stringa iza mjesta za broj umetnuta je kosa crta kako bi se lakše pratilo kako izgleda ispis. U naredbi (#1) ispisujemo jednoznamenkasti broj. On će biti poravnat s desnim krajem polja. U naredbi (#2) ispisuje se dvoznamenkasti broj koji je također ispisan uz desni rub. U naredbi (#3) peteroznamenkasti broj popunjava cijelo predviđeno polje. Brojevi s većim brojem znamenaka bit će ispisani u cijelosti i "nasilno" će proširit predviđenu širinu polja kao što to pokazuje naredba (#4).

Ako ispred broja koji određuje širinu polja za ispis napišemo znak <, broj će biti poravnan s lijevim rubom polja, a ako upišemo znak > broj će biti poravnan uz desni rub. Ako se ispred broja koji određuje širinu polja za ispis napiše znak ^, broj će biti smješten u sredinu polja ako je broj mjesta i broj koji se ispisuje broj s neparnim brojem znamenaka, a ako nije, onda će ispis biti u sredini, ali pomaknut za jedno mjesto ulijevo. Pogledajmo:

```
>>> print('{:<5d}/'.format(1))                    #5
1    /
>>> print('{:<5d}/'.format(12))                   #6
12   /
>>> print('{:>5d}/'.format(123))                  #7
  123/
>>> print('{:^5d}/'.format(1))                    #8
  1  /
>>> print('{:^5d}/'.format(12))                   #9
 12  /
>>> print('{:^4d}/'.format(1))                    #10
 1  /
>>> print('{:^4d}/'.format(12))                   #11
 12 /
```

Naredbama (#5) i (#6) brojevi su ispisani poravnani uz lijevi rub polja od pet znakova. Naredba (#7) smješta broj uz desni rub predviđenog polja za ispis. Naredba (#8) smješta broj u sredinu predviđenog polja jer je duljina broja kao i broj predviđenih mjesta neparan broj, dok naredba (#9) brojeve smješta za jedno mjesto ulijevo jer je broj znamenaka paran, a broj predviđenih mjesta neparan broj. U ispisima (#10) i (#11) vidimo kako to izgleda ako je predviđeni broj mjesta za ispis paran.

Kod ispisa stringova postupa se na isti način s tim da se iza broja koji određuje širinu polja ne piše nikakvo slovo. Evo kako takvo formatiranje djeluje:

```
>>> print('{:5}/'.format('abc'))                    #12
abc  /
>>> print('{:5}/'.format('abcde'))                   #13
abcde/
>>> print('{:5}/'.format('abcdefghi'))               #14
abcdefghi/
```

U naredbi (#12) ispisuje se string kraći od predviđene širine polja. Uočimo da se za razliku od brojeva on ispisuje počevši od lijevog ruba polja predviđene širine. Kao i kod brojeva, ako broj predviđenih mjesta odgovara duljini stringa, vrijednost će se ispisati u predviđenom broju mjesta (#13). Imamo li situaciju da je duljina stringa veća od predviđenog broja mjesta za ispis, string će se ispisati tako da će se polje za ispis proširiti koliko je to potrebno (#14). Jednaku situaciju smo imali i kod ispisa brojeva čiji je broj znamenaka bio veći od predviđene duljine polja za ispis (#4).

Stringove možemo ispisati poravnate s desnim rubom ili u sredini polja ako prije broja koji određuje širinu polja stavimo znak > odnosno ^ :

```
>>> print('{:>5}/'.format('a'))                    #13
    a/
>>> print('{:>5}/'.format('abc'))                  #14
  abc/
>>> print('{:^5}/'.format('a'))                    #15
  a  /
>>> print('{:^5}/'.format('ab'))                   #16
 ab  /
```

Vidimo da su naredbe (#13) i (#14) ispisale stringove uz desni rub polja, naredba (#15) u sredini polja, a naredba (#16) string smješta za jedno mjesto ulijevo jer je broj znamenaka paran, a broj predviđenih mjesta neparan broj. Usporedi s naredbama (#8) i (#9).

Pogledajmo još kako se ispisuju vrijednosti tipa `bool`. U interaktivnom sučelju možemo pisati:

```
>>> g = True                                                    #17
>>> h = False                                                   #18
>>> print(g, h, sep='; ')                                       #19
True; False
>>> print('g = ' + str(g) + '; ' + 'h = ' + str(h))             #20
g = True; h = False
>>> print('g = {}; h = {}'.format(g, h))                        #21
g = True; h = False
>>> print('g = {:1}; h = {:1}'.format(g, h))                    #22
g = 1; h = 0
>>> print('g = {:1d}; h = {:1d}'.format(g, h))                  #23
g = 1; h = 0
```

Naredbama (#17) i (#18) pripisali smo varijablama `g` i `h` logičke vrijednosti. Funkcija `print()` u naredbi (#19) ispisat će ih razdvojene znakom točke sa zarezom i bjelinom, jer smo tako odredili parametrom `sep`. U naredbi (#20) oblikovali smo nadovezivanjem pojedinačnih stringova ispis koji se vidi nakon izvođenja te naredbe. Uočimo da smo pritom morali funkcijom `str()` pretvoriti vrijednosti tipa `bool` i pripadne stringove. Jednaki ispis možemo na mnogo jednostavniji način pripremiti uporabom metode `format()`. Mnogo jednostavnija naredba (#21) na taj će način obaviti jednaki ispis kao i naredba (#20).

Zanimljivo je da u vitičastim zagradama možemo zadati i širinu polja za ispis (i to na način kao kod ispisa stringova ili na način kao kod ispisa brojeva – s dodanim slovom `d`). U tim slučajevima će se umjesto vrijednosti `True` ispisati broj `1`, a umjesto vrijednosti `False` broj `0`. U naredbama (#22) i (#23) odabrali smo polje širine `1`.

Ispitivanje složenih uvjeta

U dosadašnjim primjerima u `if` naredbi pojavljivali su se samo jednostavni uvjeti.

Pri rješavanju složenijih problema bit će potrebno istodobno ispitivati više uvjeta i na temelju njihovih vrijednosti donositi odluke. Složeni uvjeti grade se od većeg broja jednostavnih uvjeta uporabom **logičkih izraza**.

Dosad smo upoznali aritmetičke operatore, operatore za stringove i usporedbene operatore. U *Pythonu* postoje i tri logička operatora za ostvarenje logičkih izraza. To su operatori `and`, `or` i `not`. Logički operatori `and` i `or` djeluju na dva operanda, a operator `not` na samo jedan.

Operator `and`

Želimo li, primjerice ispisati brojeve iz intervala [0, 9] koji su veći od 3 i manji od 7, onda to možemo načiniti na sljedeći način:

```
>>> for i in range(10):
        if i > 3 and i < 7:                    #1
            print(i, end=' ')                  #2

4 5 6
```

Složeni uvjet (logički izaz) u kojem smo se koristili logičkim operatorom `and` (`#1`) je istinit samo za vrijednosti `4`, `5` i `6` te će samo za njih biti izvedena naredba (`#2`).

Operator `and` daje rezultat `True` samo onda kada oba operanda imaju vrijednost `True`, a za ostale kombinacije vrijednosti daje rezultat `False`.

Sljedećim naredbama ćemo ispisati tablicu rezultata izraza `g and h` za sve kombinacije vrijednosti varijabli:

```
>>> for g in range(2):
        for h in range(2):
            print('{} and {} = {}'.format(g, h, g and h))          #3

0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

U naredbi (#3) ispisuje se kao treća vrijednost rezultat izraza `g and h` u skladu s definiranim djelovanjem operatora `and` koji daje rezultat `True` samo onda kada oba operanda imaju vrijednost `True`, a za ostale kombinacije vrijednosti daje rezultat `False`.

Operator `or`

Želimo li, primjerice ispisati brojeve iz intervala [0, 9] koji su manji od 3 ili veći od 7 onda to možemo načiniti korištenjem operatora `or` koji daje rezultat `True` kada barem jedan operand ima vrijednost `True` i vrijednost `False` samo onda kada oba operanda imaju vrijednost `False`. Pogledajmo kako bi mogao izgledati program koji će ispisati tražene brojeve:

```
>>> for i in range(10):
        if i < 3 or i > 7:
            print(i, end=' ')

0 1 2 8 9
```

Sljedećim ćemo naredbama ispisati tablicu rezultata izraza `g or h` za sve kombinacije vrijednosti varijabli:

```
>>> for g in range(2):
        for h in range(2):
            print('{} or {} = {}'.format(g, h, g or h))          #4

0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

Operator `not`

Konačno, pogledajmo kako bismo napisali program kojim ćemo donijeti odluku o odijevanju kaputa. No, sada postavimo uvjet ovako: Ako temperatura nije veća od ili jednaka 15 °C, odjeni kaput. Zadatak ćemo riješiti korištenjem operatora `not`.

```
>>> temperatura = 12
    if not temperatura >= 15:
        print('Obuci kaput ')
```

Prikažimo tablicu djelovanja operatora `not` uz prikaz s nazivima logičkih vrijednosti.

```
>>> for g in range(2):
        print('not {} = {}'.format(bool(g), not g))                    #5

not False = True
not True = False
```

U naredbi (#5) funkcija `bool()` prevodi vrijednosti `0` u `False`, odnosno vrijednosti `1` u `True`.

Podjsetimo se kako metoda `format()` ispisuje vrijednsosti tipa `bool`.

S ova tri operatora možemo uobličiti sve potrebne složene uvjete za donošenje odluka u programima. Operator `not` ima najviši prioritet, zatim slijedi `and` i na kraju s najnižim prioritetom je operator `or`. Napomenimo još da u u *Pythonu* operatori usporedbe imaju viši prioritet od logičkih operatora.

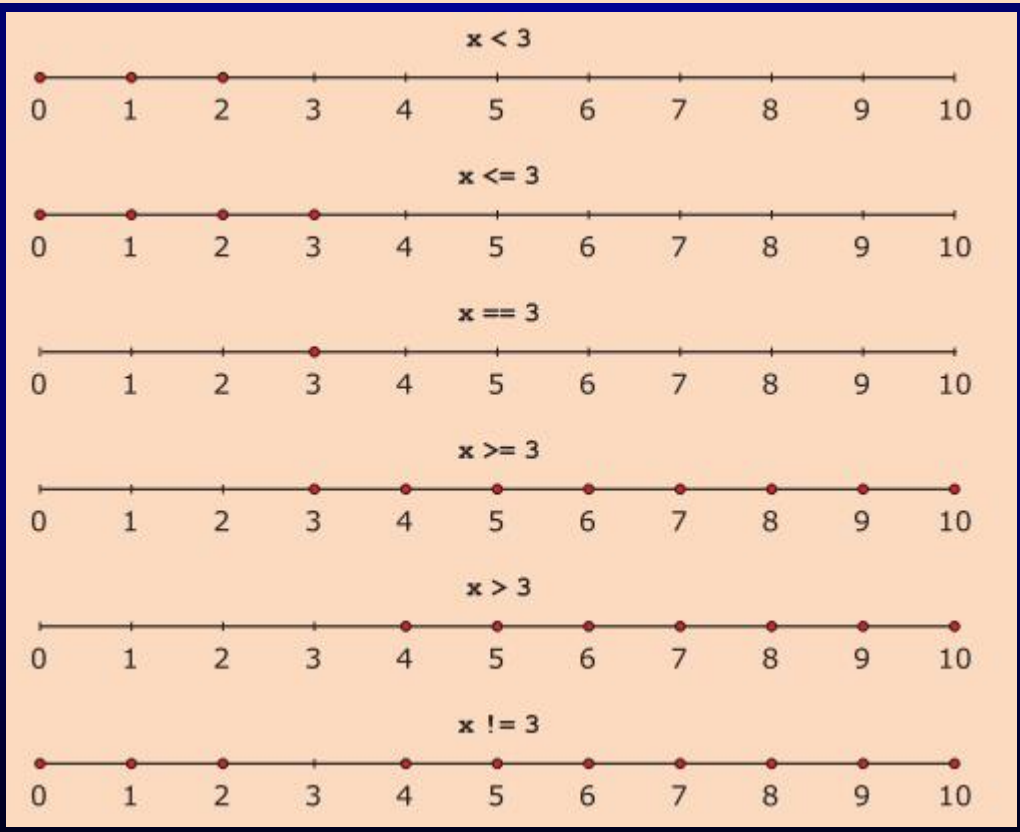
Napišimo program koji će ispitivati brojeve iz intervala [0, 10] s brojem 3 i to sa svih šest operatora usporedbe. Pritom ćemo ispisivati sve brojeve za koje su istiniti sljedeći logički uvjeti: `x < 3`, `x <= 3`, `x == 3`, `x >= 3`, `x > 3` i `x != 3`. Nakon toga ćemo uspoređivati iste brojeve s brojem `7` i ispisati sve brojeve koji zadovoljavaju sljedeće logičke uvjete: `x < 7`, `x <= 7`, `x == 7`, `x >= 7`, `x > 7` i `x != 7`.

```
#Program prikazuje ispunjenje pojedinih uvjeta - program_5_3.py

def main():
    print('Za brojeve x iz intervala [0,10]:')
    print('{:>6} je istinito za x ='.format('x < 3'), end=' ')
    for x in range(11):
        if x < 3:
            print(x, end=' ')
    print('\n{:>6} je istinito za x ='.format('x <= 3'), end=' ')          #6
    for x in range(11):
        if x <= 3:
            print(x, end=' ')
    print('\n{:>6} je istinito za x ='.format('x == 3'), end=' ')          #7
    for x in range(11):
        if x == 3:
            print(x, end=' '
    print('\n{:>6} je istinito za x ='.format('x >= 3'), end=' ')          #8
    for x in range(11):
        if x >= 3:
            print(x, end=' ')
    print('\n{:>6} je istinito za x ='.format('x > 3'), end=' ')           #9
    for x in range(11):
        if x > 3:
            print(x, end=' ')
    print('\n{:>6} je istinito za x ='.format('x != 3'), end=' ')          #10
    for x in range(11):
        if x != 3:
            print(x, end=' ')

main()
```

Uočimo da ispisni stringovi u naredbama od (#6) do (#10) počinju znakom za prijelaz u novi redak \n. To je potrebno jer se naredbama ispred njih ispisivanje u petlji završava bez prijelaza u novi redak. Isto tako, uočimo da smo predvidjeli šest mjesta te da je ispis poravnan s desne strane. Izvođenjem tog programa dobivamo sljedeći ispis:

```
Za brojeve x iz intervala [0,10]:
 x < 3 je istinito za x = 0 1 2
x <= 3 je istinito za x = 0 1 2 3
x == 3 je istinito za x = 3
x >= 3 je istinito za x = 3 4 5 6 7 8 9 10
 x > 3 je istinito za x = 4 5 6 7 8 9 10
x != 3 je istinito za x = 0 1 2 4 5 6 7 8 9 10
```

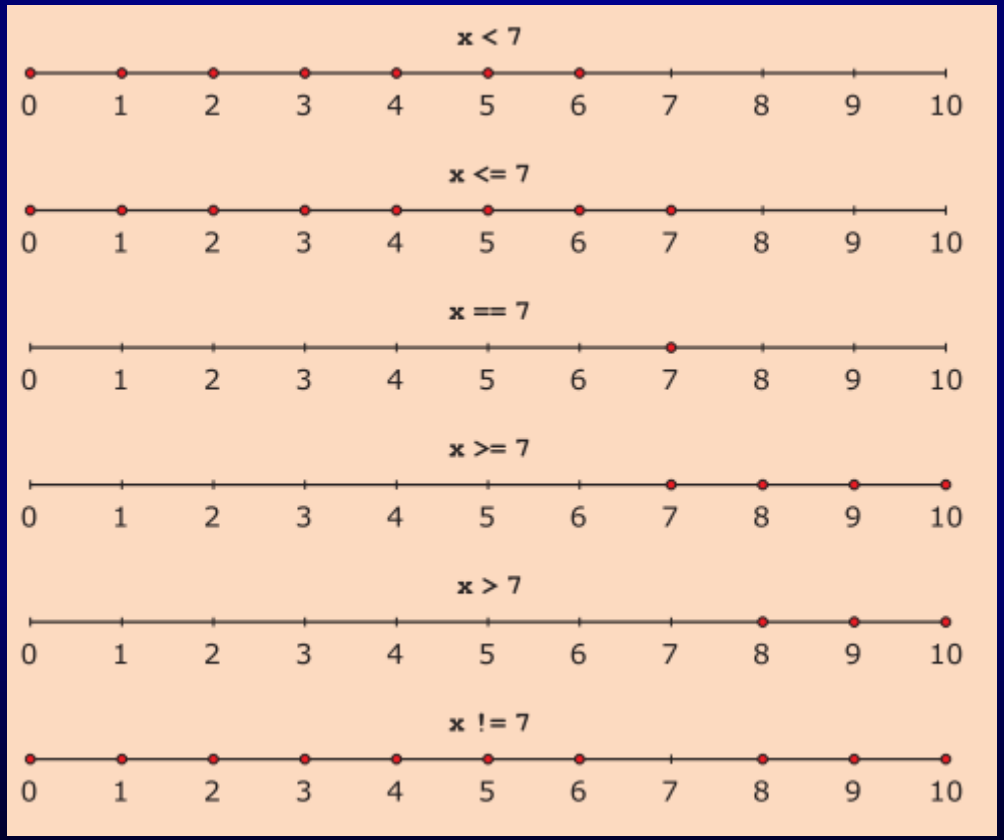
Brojeve koje je program ispisao možemo prikazati na brojevnim pravcima prikazanim na slici 5.4.



Promijenimo li u programu broj usporedbe sa 3 na 7, dobit ćemo sljedeći ispis:

```
Za brojeve x iz intervala [0,10]:
 x < 7 je istinito za x = 0 1 2 3 4 5 6
x <= 7 je istinito za x = 0 1 2 3 4 5 6 7
x == 7 je istinito za x = 7
x >= 7 je istinito za x = 7 8 9 10
 x > 7 je istinito za x = 8 9 10
x != 7 je istinito za x = 0 1 2 3 4 5 6 8 9 10
```

Brojevi koji zadovoljavaju pojedine uvjete usporedbe s brojem 7 prikazani su na brojevnim pravcima na slici 5.5.



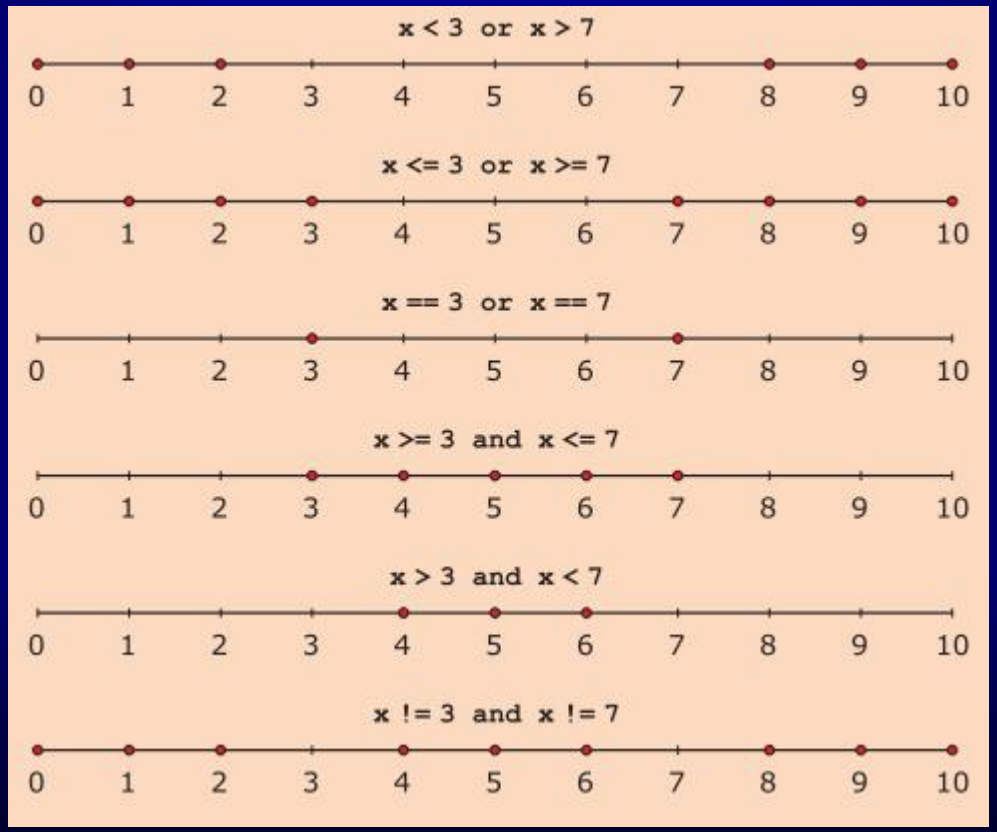
Ovi pojedinačni izrazi mogu se povezati logičkim operatorima u složene logičke izraze pa možemo ispisivati primjerice brojeve iz intervala [0, 10] koji su manji od broja 3 ili veći od broja 7 i slično. Sljedeći program ispisuje različite primjere takvih složenih uvjeta.

```
#Program prikazuje ispunjenje složenih uvjeta - program_5_4.py

def main():
    print('Za broj x iz intervala [0,10]:')
    print('{:^17} je istinito za x ='.format('x < 3 or x > 7'), end=' ')
    for x in range(11):
        if x < 3 or x > 7:
            print(x, end=' ')
    print('\n{:^17} je istinito za x ='.format('x <= 3 or x >= 7'), end=' ')
    for x in range(11):
        if x <= 3 or x >= 7:
            print(x, end=' ')
    print('\n{:^17} je istinito za x ='.format('x == 3 or x == 7'), end=' ')
    for x in range(11):
        if x == 3 or x == 7:
            print(x, end=' ')
    print('\n{:^17} je istinito uz x ='.format('x >= 3 and x <= 7'), end=' ')
    for x in range(11):
        if x >= 3 and x <= 7:
            print(x, end=' ')
    print('\n{:^17} je istinito za x ='.format('x > 3 and x < 7'), end=' ')
    for x in range(11):
        if x > 3 and x < 7:
            print(x, end=' ')
    print('\n{:^17} je istinito za x ='.format('x != 3 and x != 7'), end=' ')
    for x in range(11):
        if x != 3 and x != 7:
            print(x, end=' ')

main()
```

Izvođenjem programa dobivamo sljedeći ispis:

```
Za broj x iz intervala [0,10]:
 x < 3 or x > 7   je istinito za x = 0 1 2 8 9 10
x <= 3 or x >= 7  je istinito za x = 0 1 2 3 7 8 9 10
x == 3 or x == 7  je istinito za x = 3 7
x >= 3 and x <= 7 je istinito za x = 3 4 5 6 7
 x > 3 and x < 7  je istinito za x = 4 5 6
x != 3 and x != 7 je istinito za x = 0 1 2 4 5 6 8 9 10
```

Ti su brojevi prikazani slikom 5.6.



Često se pri rješavanju raznih problema pojavljuje potreba razmatranja brojeva iz nekog intervala, tj. između neke donje granice i neke gornje granice. Pritom granice intervala mogu biti uključene ili isključene. Prisjetimo se da se u matematici intervali koji uključuju granice nazivaju **zatvorenim intervalima** i zapisuju s pomoću uglatih zagrada. Primjerice, zapis [1, 5] obuhvaća brojeve 1, 2, 3, 4 i 5. Intervali koji ne uključuju granice nazivaju se **otvorenim intervalima** i zapisuju se korištenjem znaka < i >. Primjerice, zapis <1, 5> obuhvaća brojeve 2, 3 i 4. Intervali mogu biti i poluzatvoreni tako da je s jedne strane granica uključena, a s druge nije uključena u interval.

Iz prethodnog primjera vidimo da granice zatvorenog intervala uključujemo s operatorima usporedbe >= i <= dok za otvorene intervale vrijede operatori > i <.

U interaktivnom sučelju možemo brojeve iz otvorenog i zatvorenog intervala ispisivati na sljedeći način:

```
>>> dg, gg = 7, 13                                          #11
>>> for x in range(20):                                     #12
        if x >= dg and x <= gg:                             #13
            print(x, end=' ')

7 8 9 10 11 12 13
>>> for x in range(20):
        if x > dg and x < gg:                               #14
            print(x, end=' ')

8 9 10 11 12
```

U naredbi (#11) zadali smo donju i gornju granicu intervala. U petlji `for` (#12) varijabla petlje `x` mora poprimati vrijednosti koje će sigurno pokriti zadani interval. Odabrali smo da se ona mijenja u granicama od `0` do `19`. Složeni uvjet (#13) "odabire" i ispisuje vrijednosti iz zatvorenog intervala [7, 13]. Jednako tako, uvjet iz naredbe (#14) odredit će ispis brojeva iz otvorenog intervala <7, 13>.

Predložena rješenja su točna i daju nam željene rezultate. Međutim, mi imamo i mnogo jednostavnije rješenje koje nam omogućuje funkcija `range()`. Ta je funkcija upravo i uvedena za generiranje raznovrsnih intervala. Pogledajmo:

```
>>> for x in range(dg, gg + 1):                                    #15
        print(x, end=' ')

7 8 9 10 11 12 13
>>> for x in range(dg + 1, gg):                                    #16
        print(x, end=' ')

8 9 10 11 12
```

U naredbi (#15) varijabla petlje `x` poprimat će redom vrijednosti od `dg` do `gg`, dok će u naredbi (#16) ona poprimati vrijednosti od `dg + 1` do `gg - 1`.

U pripremi programa često se može dogoditi da neiskusni programer napiše programe koji su složeniji nego li je to potrebno.

Ostvarenje programske petlje ispitivanjem uvjeta, jednostavna `while` *petlja*

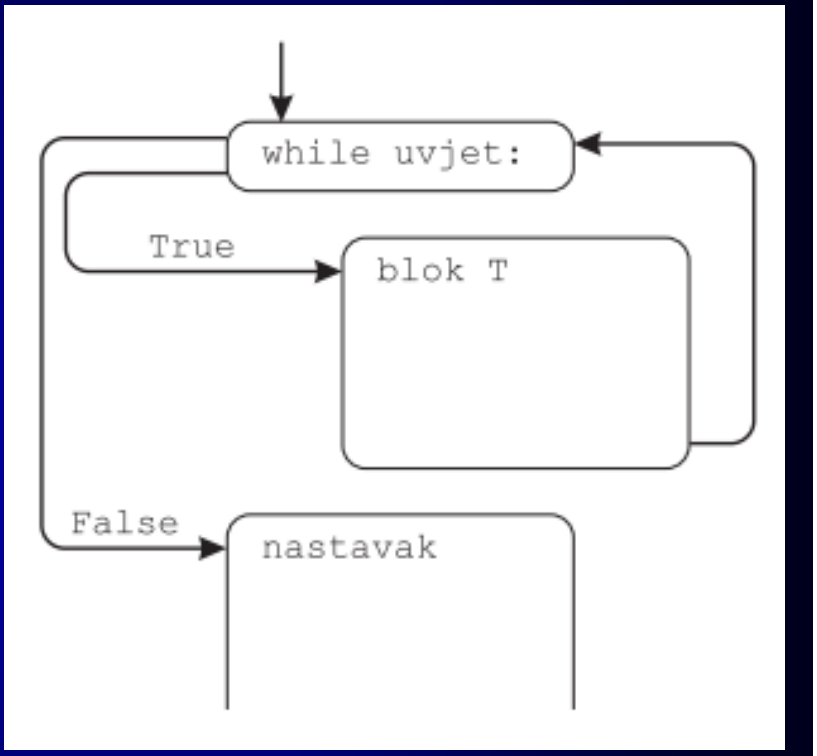
Dosad smo upoznali petlju **`for`** i njezine mogućnosti za ostvarivanje različitih programskih rješenja. Za tu petlju moramo unaprijed odrediti broj ponavljanja izvođenja naredbi u tijelu petlje.

Međutim, pri rješavanju nekih zadataka ne možemo uvijek unaprijed odrediti koliko će se puta neki blok naredbi morati ponoviti. Zbog toga u *Pythonu* (a tako je i u drugim programskim jezicima) postoji naredba **`while`** (engl. *while* – dok, dok je) kojom se ostvaruje uvjetno ponavljanje bloka naredbi. Ona se sastoji se od ključne riječi **`while`**, uvjeta čija se istinitost ispituje i dvotočke. Nakon toga slijedi blok naredbi u tijelu naredbe **`while`** koje se moraju pisati s uvlakom. Naredba **`while`** prije svakog ponavljanja izvođenja naredbi u tijelu petlje ispituje uvjet ponavljanja. Naredbe u tijelu petlje izvode se sve dok je uvjet ispunjen.

Pravilo pisanja te naredbe slično je pravilu pisanja naredbe **`if`** koju smo usvojili u odjeljku 5.2. Izvođenje te naredbe ima sličnosti s izvođenjem naredbe **`if`** – blok uvučenih naredbi, `blok T`, izvest će se samo onda ako je vrijednost izraza **`True`**.

Razlika između naredbe **while** i **if** je u ponašanju programa na kraju izvođenja `bloka T`: kod naredbe **if** prelazi se na nastavak programa dok se kod naredbe **while** izvođenje vraća na početak i ponovno ispituje uvjet napisan iza ključne riječi **while**. Ako je uvjet i dalje ispunjen, ponovno će se izvesti `blok T`. Upravo to vraćanje na ispitivanje i početak bloka stvara zatvorenu petlju, odakle i dolazi ime te programske konstrukcije (engleski je naziv za petlju *loop*).

Taj je tijek izvođenja prikazan slikom 5.7. Na nastavak programa prelazi se samo kada uvjet u naredbi **while** nije zadovoljen. Prema tome, očekuje se da će se u `bloku T` vrijednosti nekih varijabli tako promijeniti da uvjet poprimi vrijednost **False**.



Petljom `while` možemo postići jednaki učinak kao i petljom `for`. Pogledajmo jednostavni primjer u interaktivnom sučelju:

```
>>> for i in range(10):                                    #1
        print(i, end=' ')                                  #2

0 1 2 3 4 5 6 7 8 9
>>> i = 0                                                  #3
>>> while i < 10:                                         #4
        print(i, end=' ')
        i += 1                                             #5

0 1 2 3 4 5 6 7 8 9
```

Petlja `for` (#1) izvest će naredbu (#2) deset puta i pritom ispisati vrijednosti varijable `i`. Jednaki učinak možemo postići i petljom `while`. No, moramo se pobrinuti o nekim detaljima. Prvo, moramo varijabli `i` pripisati početnu vrijednost (#3) i zatim postaviti u naredbi (#4) uvjet tako da bude istinit za sve vrijednosti varijable `i` koja je manja od `10`. Nakon ispisa vrijednosti varijable, njezina se vrijednost uvećava za `1` (#5) tako da će sigurno doseći vrijednost `10`. No, ta vrijednost neće biti ispisana jer će s tom vrijednošću uvjet `i < 10` poprimiti vrijednost `False` što uzrokuje odlazak na nastavak programa.

Na sličan način možemo ispisati i padajući niz brojeva od 10 do 0:

```
>>> i = 10                                                 #6
>>> while i >= 0:                                         #7
        print(i, end=' ')
        i -= 1                                             #8

10 9 8 7 6 5 4 3 2 1 0
```

Pri oblikovanju `while` petlji mogu se dogoditi pogreške koje uzrokuju beskonačni broj njezina ponavljanja. Ako primjerice u naredbi (#8) umjesto znaka – zabunom napišemo znak +, izazvat ćemo beskonačno ponavljanje petlje (#9). Provjerimo:

```
>>> i = 10
>>> while i >= 0:
        print(i, end=' ')
        i += 1                                                    #9

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
```

Brojevi će se nastaviti ispisivati nezaustavljivo velikom brzinom sa sve većim i većim vrijednostima. Jedan od načina da taj ispis zaustavimo je istovremeni pritisak na tipki *(Ctrl)+(C)*. Ispis će se zaustaviti uz ovakvu obavijest:

```
232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249
250 251 252 253 254 255 256 257 258Traceback (most recent call last):
  File "<pyshell#11>", line 2, in <module>
    print(i, end=' ')                                             #13
  File "C:\Users\...\Python35-32\lib\idlelib\PyShell.py", line 1344,
in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
```

Ova je obavijest puna detalja koji nam u ovom trenutku nisu bitni. No, u zadnjem se redu može pročitati da je prekid izvođenja izazvan tipkovnicom (`KeyboardInterrupt`).

Ovaj postupak prekidanja programa dobro će nam doći prilikom izrade i ispitivanja programskih rješenja.

Primjer (5.6) :

Vrlo često ćemo se susretati sa situacijom da je pokretanje nekog programa uvjetovano ispravnim upisom neke riječi, rezultata nekog izraza ili nizom znakova. Primjerice, želimo li uporabu nekog programa zaštititi tako da ga može pokrenuti samo jedna osoba, tada bismo mogli zatražiti upis imena i omogućiti izvođenje programa samo onda ako je to ime prethodno pohranjeno.

Napišimo funkciju koja će prepoznavati je li upisano ime jednako pohranjenom imenu.

```
>>> def prepoznavanje_imena():
        pohranjeno_ime = 'Marko'                          #10
        upisano_ime = ''
        while upisano_ime != pohranjeno_ime:              #11
            upisano_ime = input('Upiši ime: ')            #12
        print('Lijep pozdrav,', upisano_ime)              #13
        return True                                       #14
```

U definiciji funkcije smo naredbom (#10) odredili da samo Marko smije dobiti dozvolu za daljnje djelovanje. Funkcija će vratiti vrijednost `True` ako uspijemo savladati prolaz kroz `while` petlju. Uvjet koji se ispituje je tako smišljen da će poprimiti vrijednost `False` samo onda kada je string upisanog imena jednak stringu pohranjenog imena. U tom se slučaju izvođenje nastavlja s naredbom (#12) nakon čega slijedi ispis (#13) te vraćanje vrijednosti `True` (#14).

Za bilo koji drugi string uvjet petlje ima vrijednost `True` što uzrokuje izvođenje naredbe (#12) (to je jedina naredba bloka `blok T!`). Petlja će se ponavljati tako dugo dok ne upišemo string jednak stringu pohranjenog imena. Kako bi se uvjet u naredbi (#11) mogao prvi puta ispitati (i prije prvog upisivanja imena) moramo imati pripisanu neku vrijednost varijabli `upisano_ime`. Najjednostavnije je za tu početnu vrijednost odabrati prazni string.

Ispitivanje definirane funkcije:

```
>>> dozvola = False                                    #15
>>> dozvola = prepoznavanje_imena()                    #16
Upiši ime: Ana
Upiši ime: Marija
Upiši ime: Marko
Lijep pozdrav, Marko
>>> dozvola                                            #17
True
```

Za ispitivanje definirane funkcije najprije smo naredbom (#15) nekoj varijabli `dozvola` pridodali vrijednost `False`. Nakon izvođenja naredbe (#16) varijabla `dozvola` poprimit će vrijednost koju će vratiti funkcija `prepoznavanje_imena()`. Naredbom (#17) ustanovili smo da se ta vrijednost uistinu promijenila.

Primjer:

Napišimo funkciju kojom ćemo pogađati nasumično generirani broj iz zadanog intervala [a, b]:

```
>>> from random import randint                          #18
>>> def pogađanje_broja(a, b):                          #19
        tajni_broj = randint(a, b)                      #20
        broj = a - 1                                    #21
        while broj != tajni_broj:                       #22
                broj = int(input('Pogodi tajni broj: '))
        print('Bravo!')
```

U naredbi (#18) dobavljamo iz modula `random` funkciju `randint` (vidjeti odjeljak 4.3.). Parametri `a` i `b` u naredbi (#19) određuju donju i gornju granicu intervala nasumičnih brojeva koje će poprimiti varijabla `tajni_broj` (#20). Početnu vrijednost varijable `broj` moramo odabrati tako da ne bude jednaka nekoj od mogućih nasumičnih vrijednosti. U naredbi (#21) je određeno da ta vrijednost bude za jedan manja od donje granice intervala iz kojeg će se generirati nasumični brojevi.

U petlji (#22) upisivat ćemo brojeve tako dugo dok ne pogodimo nasumično generirani broj.

Pri jednom pozivu funkcije to bi moglo izgledati ovako:

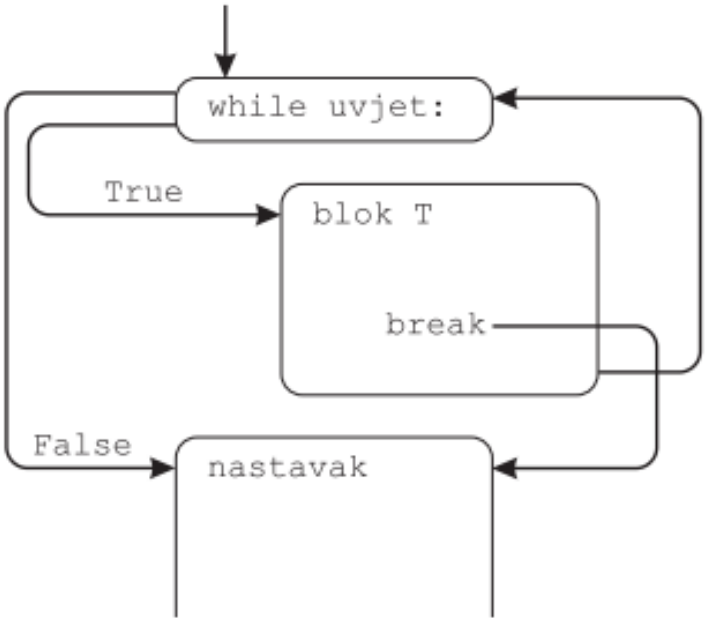
```
>>> pogađanje_broja(3, 8)
Pogodi tajni broj: 5
Pogodi tajni broj: 3
Pogodi tajni broj: 7
Bravo!
```

Izlazak iz `while` *petlje naredbom* `break`

Uobičajeno je da petlja **`while`** završava s izvođenjem kada prije novog ciklusa ponavljanja više nije ispunjen uvjet ponavljanja.

Međutim, pokazalo se da bi za neka programska rješenja bilo praktično pokrenutu petlju **`while`** napustiti i prije. U tu je svrhu smišljena naredba **`break`** (engl. *break* – prekid).

Ona djeluje tako da se izvođenje prebacuje na prvu naredbu nastavka. Opisani tijek izvođenja pod utjecajem naredbe **`break`** prikazuje slika 5.8.



Pogledajmo moguće načine njezine uporabe:

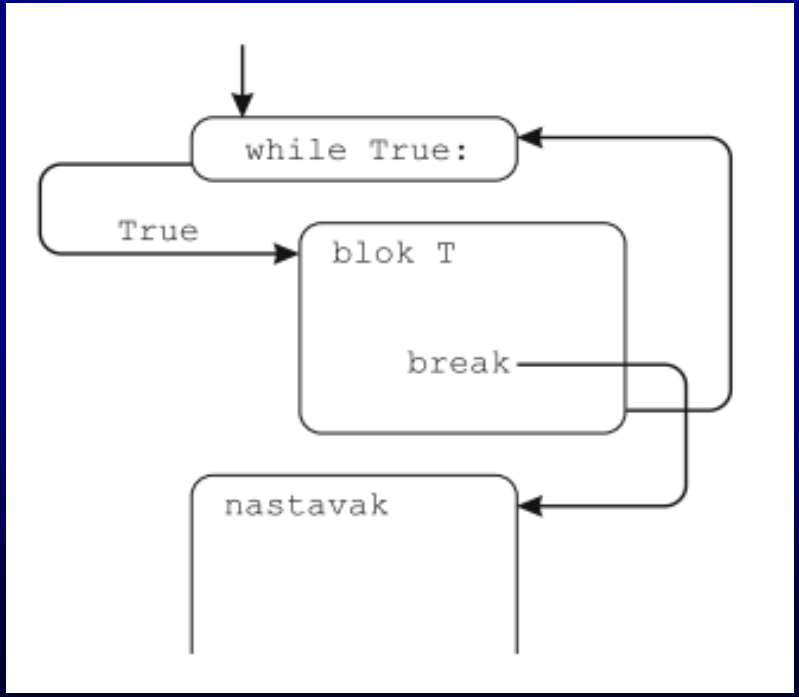
```
>>> i = 0                                                    #4
>>> while True:                                              #5
        print(i, end=' ')
        i += 1
        if i == 10:
            break                                            #6

0 1 2 3 4 5 6 7 8 9
```

Naredbom (#4) zadali smo početnu vrijednost varijable `i`. Naredbom (#5) počinje petlja koja ima uvjet jednak `True`. Ovako zadani uvjet bit će uvijek ispunjen i na njega se ne može djelovati nikakvom naredbom unutar petlje. Jedini mogući način napuštanja petlje je naredba `break` (#6). Ona će biti izvedena kada varijabla `i` poprimi vrijednost `10`. Vidimo da smo na taj način ispisali niz brojeva od `0` do `9`.

```
>>> while True:                                              #7
        print(i, end=' ')
        i -= 1
        if i == -1:                                          #8
            break                                            #9

10 9 8 7 6 5 4 3 2 1 0
```

Druga petlja koja počinje naredbom (#7) ispisat će brojeve u silaznom nizu zbog toga što se oni naredbom (#8) u svakom prolazu kroz petlju smanjuju za jedan. Vidimo da je varijabla `i` na početku imala vrijednost `10`. S tom je njezinom vrijednošću, naime, završila prva petlja.

Ako smo iza ključne riječi `while` postavili kao uvjet vrijednost `True` onda nijedna naredba u bloku petlje ne može taj uvjet promijeniti u `False`.
Prema tome u prikazu tijeka programa možemo izostaviti granu označenu s `False` koja vodi prema nastavku programa.
Jedini mogući način za napuštanje petlje je naredbom `break` !



Primjer:
Funkciju za prepoznavanje imena možemo preoblikovati tako da se u njoj rabi naredba `while True`:

```
>>> def prepoznavanje_imena_1():
        pohranjeno_ime = 'Marko'
        while True:
            if pohranjeno_ime == input('Upiši ime: '):          #1
                break                                           #2
        print('Lijep pozdrav,', pohranjeno_ime)                 #3
        return True
```

U primjeru 5.6. morali smo uvesti varijablu `upisano_ime` prije nego se ona pojavila u uvjetu `while` petlje. S obzirom na to da je ovdje uvjet u petlji `while` jednak `True`, to nam nije potrebno. Štoviše, varijabla `upisano_ime` nije nam ni potrebna jer u naredbi (#1) sadržaj varijable `pohranjeno_ime` neposredno uspoređujemo s utipkanim stringom koji će prenijeti funkcija `input()`. Ovdje se `blok T` sastoji samo od dviju naredbi (#1) i (#2). Kada uvjet u naredbi `if` ima vrijednost `False` (tj. kada utipkano ime nije jednako pohranjenom imenu), tada se druga naredba `break` neće izvoditi i izvođenje se vraća na početak petlje `while`. Kada utipkamo ime jednako pohranjenom imenu, djelovat će naredba `break` i izvođenje se nastavlja naredbom (#3).

```
>>> dozvola = False
>>> dozvola = prepoznavanje_imena_1()
Upiši ime: Ana
Upiši ime: Marija
Upiši ime: Marko
Lijep pozdrav, Marko
>>> dozvola
True
```

Često se pri rješavanju raznih problema pojavljuje potreba za upisivanjem niza naziva, imena ili brojeva. Ako unaprijed ne znamo koliko će takvih podataka biti, onda moramo na neki način odabrati neki prepoznatljivi string čijim ćemo utipkavanjem označiti kraj upisa.

Već smo naučili da unos podataka s tipkovnice obavlja funkcija `input()`. Funkcija `input()` djeluje tako da prikuplja znak po znak s tipkovnice i nakon pritiska tipke (Unos) vraća utipkani niz znakova kao string. Pogledajmo to u interaktivnom sučelju:

```
>>> upis = input('Upisati niz znakova: ')
Upisati niz znakova: abcd
>>> upis
'abcd'
```

Varijabla `upis` će nakon što smo na tipkovnici tipkama (A)(B)(C)(D)(Unos) obavili upisivanje sadržavati string `'abcd'`.

Postavlja se pitanje koji bismo string mogli odabrati za zaustavljanje utipkavanja. Svaki bismo puta sami mogli o tome odlučivati i odabrati neki string za koji znamo da se neće pojaviti u nizu naših stringova.

No, zašto ne bi to mogao biti prazan string. Naučili smo u odjeljku 5.5 da prazan string pri ispitivanju uvjeta ima vrijednost `False`. Prazan string ćemo jednostavno dobiti tako da nakon pojavljivanja prompta za upis bez prethodnog diranja bilo koje tipke pritisnemo tipku (Unos). To izgleda ovako:

```
>>> upis = input('Upisati niz znakova: ')
Upisati niz znakova:
>>> upis
''
>>> not upis
True
```

Nakon pritiska tipke (Unos) varijabla `upis` sadržavat će prazni string `''`. Kada na tu varijablu primijenimo logičku operaciju `not`, dobit ćemo vrijednost `True` što može poslužiti kao uvjet za napuštanje petlje `while` naredbom `break`. Ta je mogućnost iskorištena u sljedećem programu:

Program za upisivanje niza imena:

```
#Program za upisivanje niza imena - program_5_5.py

def upis_imena():
    imena = ''                                                          #1
    broj_imena = 0                                                      #2
    while True:
        ime = input('Upisati ime (nakon zadnjeg samo tipku (Unos)): ')
        if not ime:                                                     #3
            break
        imena += ' ' + ime                                              #4
        broj_imena += 1                                                 #5
    return imena, broj_imena

def main():
    imena, broj_imena = upis_imena()
    print('Upisano je {} imena.'.format(broj_imena))
    print('Upisana imena su: {}'.format(imena))

main()
```

U naredbi (#1) uvedena je varijabla `imena` kao prazan string, a u koji ćemo nadovezivati novoupisana imena naredbom (#4), dok je u naredbi (#2) uvedeno brojilo upisanih imena koje ćemo uvećavati u naredbi (#5) za svako novo ime. Podsjetimo se da operator + za stringove određuje nadovezivanje, a za brojeve zbrajanje!

Upisivanje imena obavlja se u petlji `while True` koju ćemo napustiti kada u naredbi (#3) program ustanovi da je upisani string prazan (tj. da smo pritisnuli samo tipku (Unos)).

Izvođenjem tog programa mogli bismo dobiti sljedeći izgled ispisa:

```
Upisati ime (nakon zadnjeg samo tipku (Unos)): Ana
Upisati ime (nakon zadnjeg samo tipku (Unos)): Pero
Upisati ime (nakon zadnjeg samo tipku (Unos)): Marin
Upisati ime (nakon zadnjeg samo tipku (Unos)): Darko
Upisati ime (nakon zadnjeg samo tipku (Unos)): Sanja
Upisati ime (nakon zadnjeg samo tipku (Unos)):
Upisano je 5 imena.
Upisana imena su:  Ana Pero Marin Darko Sanja
```

Na sličan bismo način mogli postupiti i pri upisivanju niza brojeva. S obzirom na to da brojeve funkcija `input()` *unosi također kao stringove koje prije računanja moramo funkcijom* `int()` *prevoditi u brojeve, onda je razumno ulazni string rabiti što je moguće dulje kao string i pretvoriti ga u tip* `int` *tek kada moramo nešto s njim izračunati. Sljedeći program koji učitava niz brojeva i izračunava njihov zbroj mogao bi izgledati ovako:*

```
#Program za upisivanje niza imena - program_5_6.py

def upis_brojeva_i_zbroj():
    brojevi = ''                                                    #6
    zbroj = 0                                                       #7
    while True:
        broj = input('Upisati broj (nakon zadnjeg samo tipku (Unos)): ')
        if not broj:
            break
        brojevi += ' ' + broj                                       #8
        zbroj += int(broj)                                          #9
    return brojevi, zbroj

def main():
    brojevi, zbroj = upis_brojeva_i_zbroj()
    print('Upisani su sljedeći brojevi: {}'.format(brojevi))
    print('Njihov je zbroj: {}'.format(zbroj))

main()
```

Ovaj program i `program_5_5.py` vrlo su slični. Istaknimo samo neke razlike. U naredbi (#6) varijabla s praznim stringom je preimenovana u `brojevi`, jer ćemo u njoj naredbom (#8) nadovezati sve utipkane brojeve (ali kao stringove!). Nadalje, u naredbi (#7) uvedena je varijabla `zbroj` s početnom vrijednošću 0. U tu ćemo varijablu pohranjivati zbroj upisanih brojeva. U naredbi (#9) izračunava se taj zbroj, ali tako da prije pribrajanja funkcijom `int()` string pretvorimo u broj. Ostatak programa je jasan sam po sebi. Izvođenjem programa možemo dobiti sljedeći ispis:

```
Upisati broj (nakon zadnjeg samo tipku (Unos)): 4
Upisati broj (nakon zadnjeg samo tipku (Unos)): 6
Upisati broj (nakon zadnjeg samo tipku (Unos)): 8
Upisati broj (nakon zadnjeg samo tipku (Unos)):
Upisani su sljedeći brojevi:  4 6 8
Njihov je zbroj: 18
```

Program prihvaća proizvoljno velike brojeve, tako možemo dobiti i ovakav ispis:

```
Upisati broj (nakon zadnjeg samo tipku (Unos)): 1
Upisati broj (nakon zadnjeg samo tipku (Unos)): 1111111111111
Upisati broj (nakon zadnjeg samo tipku (Unos)): 1000000000002
Upisati broj (nakon zadnjeg samo tipku (Unos)):
Upisani su sljedeći brojevi:  1 1111111111111 1000000000002
Njihov je zbroj: 2111111111114
```