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

07
Zbirke funkcija i moduli, algoritmi za operacije s prirodnim brojevima

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

Raščlanjivanje problema

Rastavljanje problema na manje dijelove je jedan od vrlo djelotvornih načina za pripremu programa. Ako neki složeniji problem rješavamo kao cjelinu, moramo se brinuti o mnogo međusobno isprepletenih detalja. Kada ga rastavimo na manje dijelove, svakom se dijelu možemo posvetiti mnogo detaljnije i bolje pripremiti njegovo rješenje.

Dosad smo već naučili da dijelove rješenja pripremamo u obliku funkcija koje ćemo u glavnoj funkciji ili glavnom dijelu programa na odgovarajući način povezati kako bismo riješili neki složeniji problem.

U ovom ćemo poglavlju naučiti kako se zbirka funkcija potrebna za rješavanje nekog problema pohranjuje u obliku tzv. modula. Funkcije iz takvog modula možemo nakon toga rabiti za rješavanje drugih problema. Za tu smo svrhu odabrali algoritme s prirodnim brojevima tj. algoritme koji se bave prostim i složenim brojevima, rastavljanjem brojeva na proste faktore te određivanjem najvećeg zajedničkog djelitelja i najmanjeg zajedničkog višekratnika dvaju brojeva.

Najprije ćemo opisati najstariji poznati postupak određivanja prostih brojeva poznat kao Eratostenovo sito. U imenu se pojavljuje naziv sito jer na neki način tim postupkom kao da prosijavamo složene brojeve. Taj postupak nije optimalan za pronalaženje velikih prostih broja, ali nas lijepo uvodi u naš problem.

Eratostenovo sito

Eratosten je bio grčki matematičar koji je živio u 3. stoljeću te je osmislio postupak kojim se iz niza brojeva postupno izbacuju složeni brojevi. Oni "propadaju" kroz otvore na situ u kojem na kraju ostaju samo prosti brojevi. Prvo redom zapišemo prirodne brojeve počevši s brojem 2 pa do broja do kojega želimo "prosijati" složene brojeve. Sam postupak izbacivanja složenih brojeva započinjemo brojem 2 i iz popisa izbacujemo sve njegove višekratnike. Nadalje izbacujemo višekratnike broja 3 i tako redom dok ne dođemo do posljednjeg broja. Očito će na kraju u popisu preostati samo oni brojevi koji nisu bili djeljivi niti jednim brojem, odnosno u popisu će ostati samo prosti brojevi.

Ilustrirajmo postupak na primjeru prirodnih brojeva od 2 do 30:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	

Prvo ćemo izbaciti sve višekratnike broja 2 (označeni su crvenom bojom):

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	

Preostali su brojevi:

U ovako dobivenom nizu izbacit ćemo sve višekratnike sljedećeg broja (3):

te će niz imati oblik:

Nadalje izbacujemo višekratnike broja 5. Višekratnici broja 5, osim njega samoga bili bi brojevi: 10, 15, 20, 25 i 30. Primijetimo da su brojevi 10, 15, 20 i 30 već izbačeni, dakle, prvi višekratnik broja 5 koji će biti izbačen bit će broj 25. Primijetimo da se radi o umnošku tog broja sa samim sobom. Vratimo li se na izbacivanje višekratnika broja 3, uočit ćemo da je prvi izbačeni višekratnik broja 3 bio upravo broj 9, dakle opet umnožak tog broja sa samim sobom. Nakon izbacivanja višekratnika broja 5 u nizu će ostati sljedeći brojevi:

Sljedeći broj čije višekratnike trebamo izbacivati je broj 7, kao što smo već ustanovili za manje brojeve, prvi broj koji bismo trebali izbaciti bio bi broj 49 (umnožak broja 7 sa samim sobom). Kako se broj 49 više ne nalazi u nizu (veći je od broja 30) možemo zaključiti da postupak možemo završiti.

Opisani postupak je naizgled jednostavan, no može biti dugotrajan imamo li puno brojeva. Stoga ćemo u nastavku napisati program koji će to načiniti umjesto nas.

Pretpostavimo da želimo dobiti popis svih prostih brojeva do nekog zadanog broja *n* (u gornjem primjeru *n* je bio 30). Prvo ćemo sve prirodne brojeve od 2 do zadanog broja *n* pohraniti u listu `popis`.

Naučili smo da listu `popis` možemo stvoriti na više načina:

```
>>> n = 30
>>> popis = []
>>> for i in range (2, n + 1):
        popis.append(i)

>>> popis
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30]
```

```
>>> popis = list(range(2, n + 1))
>>> popis
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30]
```

```
>>> popis = [i for i in range (2, n + 1)]
>>> popis
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30]
```

Nakon što smo kreirali listu prirodnih brojeva od 2 do *n*, preostaje nam izbaciti redom višekratnike prostih brojeva počevši od najmanjeg. To možemo načiniti tako da za svaki broj izračunavamo njegove višekratnike te ako se on nalazi u listi `popis`, izbacimo ga. Prvo izbacimo višekratnike broja 2:

```
>>> for j in range(2, n // 2 + 1):
        t = 2 * j
        if t in popis:
            popis.remove(t)

>>> popis
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Nakon izbacivanja višekratnika broja 2 izbacimo redom višekratnike broja 3 (#1), zatim broja 5 (#2).

```
>>> for j in range(3, n // 3 +1):                    #1
        t = 3 * j
        if t in popis:
            popis.remove(t)

>>> popis
[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29]

>>> for j in range(5, n // 5 +1):                    #2
        t = 5 * j
        if t in popis:
            popis.remove(t)

>>> popis
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Vidimo da smo za zadani broj 30 na ovaj način dobili listu prostih brojeva. Sljedeći prosti broj u listi je 7, a njegov prvi višekratnik koji bi se mogao nalaziti u listi je 49 i veći je od našeg zadanog broja, pa postupak možemo završiti.

Primjer:

Napišimo program koji će vraćati popis prostih brojeva manjih od upisanog broja `n`.

```
#Ispitivanje Eratostenova sita - program_7_1.py

def eratosten(n):
    popis = [i for i in range(2, n + 1)]                    #3
    k = 0                                                   #4
    while popis[k] ** 2 <= n:                               #5
        for j in range(2, n // popis[k] + 1):
            t = popis[k] * j
            if t in popis:
                popis.remove(t)
        k += 1
    return popis                                            #6

def main():
    n = int(input('Upiši broj n = '))
    print('Prosti brojevi manji ili jednaki {} su:'.format(n))
    prosti = eratosten(n)                                   #7
    for p in prosti:                                        #8
        print(p, end=' ')

main()
```

Naredbom (#3) kreirali smo početnu listu. Varijabla `k` je u naredbi (#4) postavljena tako da pokazuje na prvi element te početne liste. U petlji koja počinje naredbom (#5) povećavat ćemo njezinu vrijednost tako da pokazuje na sljedeći element te liste, tj. na sljedeći prosti broj čije višekratnike treba izbaciti. Uvjet u naredbi (#5) pokazuje da to činimo tako dugo dok umnožak sljedećeg broja samog sa sobom, tj. kvadrat tog broja, ne prelazi zadani broj *n*.

Funkcija će nakon zadnjeg izbacivanja naredbom (#6) vratiti listu `popis` u kojoj su preostali samo prosti brojevi. U funkciji `main()` nakon poziva funkcije `eratosten()` u varijablu `prosti` (#7) pohranit će se lista prostih brojeva. Proste brojeve ćemo ispisati s pomoću petlje (#8).

Nakon pokretanja ovog programa i unošenja, primjerice broja 30, na zaslonu će biti ispisano:

```
>>>
Upiši broj n = 30

Prosti brojevi manji ili jednaki 30 su:
2 3 5 7 11 13 17 19 23 29
```

odnosno nakon ponovnog pokretanja i upisivanja broja 100, na zaslonu će biti ispisano:

```
>>>
Upiši broj n = 100

Prosti brojevi manji ili jednaki 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
```

Priprema funkcija u prozoru interaktivnog sučelja i u prozoru editora

Tijekom izučavanja uporabe programskog jezika *Python* uvjerili smo se da je vrlo praktično neka početna ispitivanja djelovanja programskih odsječaka obaviti u interaktivnom sučelju. Nekoliko uzastopnih naredbi koje rješavaju neki jednostavni problem možemo napisati u interaktivnom sučelju i odmah vidjeti njihovo djelovanje.

Međutim, veliki je nedostatak priprema funkcija u interaktivnom sučelju to što će cijeli napisani tekst biti izgubljen nakon što zatvorimo prozor interaktivnog sučelja.

Zbog toga je korisno funkcije prirediti u prozoru editora. Nakon što smo funkciju napisali, možemo je pospremiti u datoteku recimo pod imenom `radna_datoteka.py` u našu radnu mapu `moj_py` i zatim ju izvesti standardnim postupkom naredbom ***Run Modul*** ili pritiskom na funkcijsku tipku *(F5)*. Vidimo da se neće ništa dogoditi osim što će se u interaktivnom sučelju pojaviti ispis:

```
========== RESTART: C:/moj_py/radna_datoteka.py ==========
```

Međutim, u interaktivnom sučelju će se nakon toga moći pozivati funkcija koju smo zapisali u tu datoteku tako da ju možemo normalno pozivati u interaktivnom sučelju.

Prilikom kopiranja teksta funkcije iz interaktivnog sučelja u *Pythonov* editor, moramo paziti na pravilne uvlake. Može se dogoditi da je zaglavlje funkcije dobro pozicionirano (od prvog stupca) no da ostale naredbe nisu pravilno pozicionirane (uvučene). To možemo popraviti tako da označimo naredbe na koje želimo primijeniti oblikovanje te nakon toga odaberemo naredbu ***Dedent Region*** u padajućem izborniku ***Format***.

Napredni postupak pronalaženja prostih brojeva

Znamo da je prosti broje djeljiv samo s brojem 1 i sa samim sobom. Dakle, broj je prost ako ima samo dva djelitelja.
Na tim spoznajama oblikovat ćemo postupak za pronalaženje prostih brojeva

Napišimo funkciju koja će vraćati listu svih djelitelja zadanog broja `n`.

```
>>> def djelitelji(n):
        djel = []
        for d in range(1, n + 1):
            if n % d == 0:
                djel.append(d)
        return djel
```

Funkcija `djelitelji()` vratit će listu svih djelitelja broja *n*. Takva lista nastat će tako da u praznu listu `djel` postupno dodajemo djelitelje metodom `append()`. Funkciju možemo napisati i u kraćem obliku:

```
>>> def djelitelji_2(n):
        return [d for d in range(1, n + 1) if n % d == 0]

>>> djelitelji_2(16)
[1, 2, 4, 8, 16]
>>> djelitelji_2(17)
[1, 17]
```

Umjesto u interaktivnom sučelju funkcije možemo zapisati i u editoru. Funkcija će u editoru izgledati ovako:

```
def djelitelji(n):
    djel = []
    for d in range(1, n + 1):
        if n % d == 0:
            djel.append(d)
    return djel
```

odnosno funkcija `djelitelji_2()` će izgledati ovako:

```
def djelitelji_2(n):
    return [d for d in range(1, n + 1) if n % d == 0]
```

Nakon pohranjivanja tih funkcija u datoteku `radna_datoteka.py` moći ćemo ih pozivati nakon što datoteku pokrenemo i u interaktivnom se sučelju pojavi ispis:

```
============= RESTART: C:/moj_py/radna_datoteka.py =============
```

Pogledajmo nekoliko poziva naših funkcija:

```
>>> djelitelji(68)
[1, 2, 4, 17, 34, 68]
>>> djelitelji_2(94)
[1, 2, 47, 94]
```

Primjer 7.3.

```
>>> for n in range(2, 31):
        print('Djelitelji broja {:2d} su {}:'.format(n, djelitelji_2(n)))

Djelitelji broja  2 su: [1, 2]
Djelitelji broja  3 su: [1, 3]
Djelitelji broja  4 su: [1, 2, 4]
Djelitelji broja  5 su: [1, 5]
Djelitelji broja  6 su: [1, 2, 3, 6]
Djelitelji broja  7 su: [1, 7]
Djelitelji broja  8 su: [1, 2, 4, 8]
Djelitelji broja  9 su: [1, 3, 9]
Djelitelji broja 10 su: [1, 2, 5, 10]
Djelitelji broja 11 su: [1, 11]
Djelitelji broja 12 su: [1, 2, 3, 4, 6, 12]
Djelitelji broja 13 su: [1, 13]
Djelitelji broja 14 su: [1, 2, 7, 14]
Djelitelji broja 15 su: [1, 3, 5, 15]
Djelitelji broja 16 su: [1, 2, 4, 8, 16]
Djelitelji broja 17 su: [1, 17]
Djelitelji broja 18 su: [1, 2, 3, 6, 9, 18]
Djelitelji broja 19 su: [1, 19]
Djelitelji broja 20 su: [1, 2, 4, 5, 10, 20]
Djelitelji broja 21 su: [1, 3, 7, 21]
Djelitelji broja 22 su: [1, 2, 11, 22]
Djelitelji broja 23 su: [1, 23]
Djelitelji broja 24 su: [1, 2, 3, 4, 6, 8, 12, 24]
Djelitelji broja 25 su: [1, 5, 25]
Djelitelji broja 26 su: [1, 2, 13, 26]
Djelitelji broja 27 su: [1, 3, 9, 27]
Djelitelji broja 28 su: [1, 2, 4, 7, 14, 28]
Djelitelji broja 29 su: [1, 29]
Djelitelji broja 30 su: [1, 2, 3, 5, 6, 10, 15, 30]
```

Ovdje još jednom možemo primijetiti da prosti brojevi (2, 3, 5, 7, 11, 13, 17 i 19, 23, 29) imaju samo dva djelitelja (broj 1 i samog sebe) dok složeni brojevi imaju više od dvaju djelitelja. Broj će dakle biti prost ako funkcija `djelitelji()`, odnosno `djelitelji_2()` vrati listu koja ima samo dva elementa.

Napišimo funkciju koja će provjeravati je li broj prost tako da prebroji djelitelje broja.

```
>>> def prost_1(n):
        return len(djelitelji(n)) == 2
```

Ta će funkcija vraćati vrijednost `True` *ako je broj prost i vrijednost* `False` *za složene brojeve.*

Funkciju bi mogli prirediti i na drugi način

Iz ispisa u primjeru 7.3. uvjerili smo se da je broj prost ako ima samo dva djelitelja: broj 1 i sam taj broj. Broj je složen ako ima i drugih djelitelja osim 1 i samoga sebe. U tablici vidimo da su najmanji djelitelji većine brojeva do 30 brojevi 2 ili 3, odnosno da je 5 najmanji djelitelj jedino broja 25. Uočimo da je $25 = 5^2$.

Izvedemo li funkciju `prost_1()` za veće brojeve *n*, uvjerit ćemo se da će najmanji djelitelji svih brojeva manjih od 49 biti 2, 3 ili 5, te da će za broj 49 to biti broj 7 i to zbog toga što je $49 = 7^2$. Dakle, najmanji djelitelj broja 49 je 7.

Prema tome, možemo zaključiti da traženje djelitelja *d* broja *n* ima smisla obavljati tako dugo dok je `d**2 <= n`.

Napišimo funkciju koja će provjeravati je li broj prost na način da se funkcija prekine čim naiđe na prvi djelitelj zadanog broja.

```
>>> def prost(n):
        d = 2
        while d ** 2 <= n:                    #1
            if n % d == 0:
                return False                  #2
            d += 1
        return True                           #3
```

U petlji (#1) povećavamo broj *d* redom počevši od 2 sve dok je zadovoljen uvjet petlje kojim ispitujemo je li `d**2 <= n`. Čim smo naišli na prvi djelitelj broja *n* naredbom (#2), napustit ćemo funkciju i broj *n* proglasiti složenim. Ako se u rasponu brojeva od 2 do ispunjenja uvjeta petlje nije pronašao ni jedan djelitelj, funkcija će naredbom (#3) vratiti vrijednost `True`, tj. proglasiti broj *n* prostim.

Jasno nam je da je ovako pripremljena funkcija mnogo djelotvornija od naše prvotne funkcije koja najprije određuje sve djelitelje nekog broja i tek nakon toga provjerava postoje li samo dva takva djelitelja.

Razlika u brzini izvođenja ovih dviju funkcija za ispitivanje broja nije zamjetljiva za male brojeve kojima se mi uobičajeno bavimo, ali je to za neke primjene u kojima treba ispitati velike brojeve jako važno. Razliku u brzini izvođenja možemo ustanoviti sljedećim eksperimentom u interaktivnom sučelju.

Pozovemo li funkciju `prost_1()`

```
>>> prost_1(123456789)
False
```

odgovor da je zadani broj složen ispisat će se nakon jedne do dvije minute (ovisno o brzini računala na kojem radimo), dok će se pozivom funkcije `prost()`

```
>>> prost(123456789)
False
```

odgovor ispisati praktički trenutačno. Zaključujemo da je preporučljivo rabiti ovaj drugi oblik funkcije.

Sada smo u mogućnosti napisati i drukčiju funkciju za ispis liste prostih brojeva koja može zamijeniti funkciju `eratosten()`. Ona može izgledati ovako:

```
>>> def prosti_brojevi(n):
        return  [i for i in range(2, n + 1) if prost(i)]
```

Izvedimo funkciju za $n = 50$:

```
>>> prosti_brojevi(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Najveći prosti broj manji od nekog broja možemo odrediti tako da iz liste generiranih prostih brojeva odaberemo zadnji element. Znamo da se zadnji element neke liste može dohvatiti indeksom −1 pa možemo pisati:

```
>>> prosti_brojevi(50)[-1]
47
```

```
>>> eratosten(50)[-1]
47
```

Ispis broja pojavit će se u oba slučaja odmah nakon što smo pritisnuli tipku *(Unos)*.

Međutim, za velike brojeve trajanje izvođenja tih dviju funkcija bitno se razlikuje. Tako ćemo vrlo brzo funkcijom `prosti_brojevi()` dobiti najveći prosti broj manji od 100 000. Pogledajmo:

```
>>> prosti_brojevi(100000)[-1]
99991
```

No, ako upotrijebimo funkciju `eratosten()` na ispis ćemo čekati koju minutu:

```
>>> eratosten(100000)[-1]
99991
```

Iz ovog možemo zaključiti da za pronalaženje liste prostih brojeva treba odabrati funkciju `prosti_brojevi()`.

Primjer:

Napišimo program koji će tražiti unošenje dvaju prirodnih brojeva m *i* n *te nakon toga ispisivati sve proste brojeve u intervalu od* m *do* n.

```
#Pronalaženje prostih brojeva u zadanom intervalu - program_7_2.py

def prost(n):
    d = 2
    while d ** 2 <= n:
        if n % d == 0:
            return False
        d += 1
    return True

def main():
    m = int(input('Upiši donju granicu:  '))
    n = int(input('Upiši gornju granicu: '))
    print ('Prosti brojevi u intervalu [{}, {}] su:'.format(m, n))
    print([i for i in range(m, n + 1) if prost(i)])

main()
```

Pokretanjem programa i unošenjem vrijednosti 17 i 83 dobit ćemo sljedeći ispis:

```
Upiši donju granicu: 17
Upiši gornju granicu: 83
Prosti brojevi u intervalu [17, 83] su:
[17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]
```

Uporaba modula

Python je programski jezik koji se lako prilagođuje različitim područjima primjene. Njegova je velika prednost što se on sastoji od osnovnog dijela koji je jednostavan i mnogo modula koji su prilagođeni pojedinim primjenama.

Na taj je način olakšano savladavanje jezika. Svi koji žele rabiti *Python* u osnovnom obliku moraju naučiti samo njegov osnovni dio i nakon toga upoznati samo one module koji će im olakšati pripremu programa za rješavanje problema u njihovu području djelovanja.

Mi smo se dosad susreli s dva modula: `turtle` i `random`.

Moduli se mogu pokretati na više načina:

```
>>> import turtle                    #1
>>> turtle.fd(100)
>>> turtle.lt(90)
```

```
>>> from turtle import *             #2
>>> fd(100)
>>> lt(90)
```

U naredbi (#2) zvjezdica označava da se importiraju sve funkcije modula. Posebnost ovakvog načina importiranja je što pri pozivu pojedinih funkcija ne moramo navoditi ime modula kao što smo to morali činiti kada smo modul uvezli na način kao u naredbi (#1).

Nadalje, iz modula možemo importirati i pojedinačne funkcije. Tako smo iz modula `random` importirali funkciju za generiranje nasumičnih brojeva `randint()` na sljedeći način:

```
>>> from random import randint
>>> randint(10, 20)                  #3
14
```

Promjena načina pripreme programa - program pripremljen kao modul

Rekli smo da svaki program pohranjen u svoju datoteku može biti importiran. Prisjetimo se da se svaka programska datoteka uobičajeno sastoji od definicija funkcija nakon kojih počinjemo pisati naredbe glavnog programa. Naredbe glavnog programa počinjemo pisati počevši od prvog stupca svakog retka. Naredbe glavnog programa izvode se redom od prve po redu pa do zadnje.

Zbog preglednosti odlučili smo u našim programima i naredbe glavnog programa pisati kao posebnu funkciju koju smo nazvali `main()`. U tom se slučaju naš "glavni program" sastoji od samo od jedne jedine naredbe – poziva funkcije `main()`. U dosadašnjim smo se primjerima uglavnom koristili takvim načinom oblikovanja programa.

Kada se datoteka takvog programa importira, istovremeno će se pokrenuti i izvesti i njezin glavni program. To je nekada poželjno, ali nekada nam može i smetati. Zbog toga u *Pythonu* postoji mogućnost drukčijeg oblikovanja programa što će nam osigurati da se:

- glavni program izvodi ako je njegova datoteka pokrenuta za izvođenje (naredbom ***Run Modul***) ili
- glavni se program ne izvodi ako je njegova datoteka importirana, ali se sve njegove funkcije mogu pozivati i izvoditi.

Postavlja se pitanje kako to *Python* obavlja?

Promjena načina pripreme programa - program pripremljen kao modul

Rekli smo da svaki program pohranjen u svoju datoteku može biti importiran. Prisjetimo se da se svaka programska datoteka uobičajeno sastoji od definicija funkcija nakon kojih počinjemo pisati naredbe glavnog programa. Naredbe glavnog programa počinjemo pisati počevši od prvog stupca svakog retka. Naredbe glavnog programa izvode se redom od prve po redu pa do zadnje.

Zbog preglednosti odlučili smo u našim programima i naredbe glavnog programa pisati kao posebnu funkciju koju smo nazvali `main()`. U tom se slučaju naš "glavni program" sastoji od samo od jedne jedine naredbe – poziva funkcije `main()`. U dosadašnjim smo se primjerima uglavnom koristili takvim načinom oblikovanja programa.

Kada se datoteka takvog programa importira, istovremeno će se pokrenuti i izvesti i njezin glavni program. To je nekada poželjno, ali nekada nam može i smetati. Zbog toga u *Pythonu* postoji mogućnost drukčijeg oblikovanja programa što će nam osigurati da se:

- glavni program izvodi ako je njegova datoteka pokrenuta za izvođenje (naredbom ***Run Modul***) ili
- glavni se program ne izvodi ako je njegova datoteka importirana, ali se sve njegove funkcije mogu pozivati i izvoditi.

Postavlja se pitanje kako to *Python* obavlja?

U programskom okružju koje omogućuje izvođenje programa postoje "unutarnji" podatci u kojima se bilježe sve aktivnosti programa tijekom izvođenja, kao i unutarnje funkcije koje upravljaju izvođenjem programa. Važan dio tog programskog okružja je i javljanje raznih pogrešaka koje se događaju tijekom izvođenja.

Među tim unutarnjim podatcima nalazi se i jedna varijabla s nazivom `__name__` (unutarnje varijable obično počinju i završavaju dvjema uzastopnim podvlakama). U tu varijablu programsko okružje *Pythona* pohranjuje:

- string `'__main__'` ako je datoteka `'ime_datoteke'` pokrenuta kao program naredbom ***Run Modul*** iz padajućeg izbornika ***Run***, odnosno
- string `'ime_datoteke'` ako je datoteka importirana.

U našem programu mi možemo dohvatiti tu unutarnju varijablu `__name__`, ispitati njezinu vrijednost te dozvoliti izvođenje glavnog programa samo ako je njezina vrijednost jednaka `'__main__'`.

U našem programu mi možemo dohvatiti tu unutarnju varijablu `__name__`, ispitati njezinu vrijednost te dozvoliti izvođenje glavnog programa samo ako je njezina vrijednost jednaka `'__main__'`.

To znači da ćemo umjesto:

```
def main():
    naredba 1
    naredba 2
    ...
    naredba k

main()
```

sve naredbe napisati na sljedeći način:

```
if __name__ == '__main__':
    naredba 1
    naredba 2
    ...
    naredba k
```

Ovakvo pisanje glavnog programa omogućuje da svaki program u kojem su definirane neke funkcije možemo oblikovati kao modul koji ćemo moći importirati u neke druge programe (ili njegove funkcije pozivati u interaktivnom sučelju).

U načelu je preporučljivo sve programe, bez obzira namjeravamo li njegove funkcije kasnije rabiti u drugim programima, pripremiti u obliku modula. Zbog toga je razumljivo da se u padajućem izborniku ***Run*** naredba za pokretanje programa zove ***Run Module***.

Zbog toga ćemo odsad na dalje naše programe oblikovati na dva načina:

- funkcijom `main()` ako smatramo da nam njegove funkcije neće biti potrebne, ili
- naredbama glavnog programa napisanim iza ispitivanja uvjeta `__name__ == '__main__'` ako smatramo da su njegove funkcije šire upotrebljive.

Dobro je prilikom stvaranja modula na početku svake od funkcija unutar dokumentacijskog stringa navesti osnovno djelovanje te funkcije.

Pripremimo modul pod nazivom `prirodni` koji će sadržavati funkcije `djelitelji()`, `prost()` i `prosti_brojevi()`. U svaku od funkcija dodajmo dokumentacijski string. U glavni dio programa ćemo dodati i naredbe za provjeru djelovanja pojedinih funkcija.

```
#Modul: prirodni.py
'''
   modul prirodni
   sadrži funkcije za operacije s prirodnim brojevima
'''

def djelitelji(n):
    '''Funkcija vraća listu svih djelitelja broja n'''
    djel = []
    for d in range(1, n + 1):
        if n % d == 0:
            djel.append(d)
    return djel

def prost(n):
    '''Funkcija vraća vrijednosti:
         True   ako je broj prost
         False  ako je broj složen
    '''
    d = 2
    while d ** 2 <= n:
        if n % d == 0:
            return False
        d += 1
    return True
```

```
def prosti_brojevi(n):
    '''Funkcija vraća listu prostih brojeva manjih ili jednakih n'''
    return [i for i in range(2, n) if prost(i)]

if __name__ == '__main__':
    n = int(input('Upiši broj n = '))
    print('U intervalu [2, {}] su prosti: \n{}'.format(n, prosti_brojevi(n)))
    print('Djelitelji broja {} su: \n{}'.format(n, djelitelji(n)))
```

Pokrenemo li modul kao program i upišemo broj 50, ispis će izgledati ovako:

```
Upiši broj n = 50
U intervalu [2, 50] su prosti:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
Djelitelji broja 50 su:
[1, 2, 5, 10, 25, 50]
```

Važno !

Napomenimo još jednom da su ugrađeni moduli *Pythona* uvijek dostupni i u interaktivnom sučelju i u uređivačkom prozoru. Međutim, naš modul koji smo pripremili na gore opisani način interaktivno sučelje neće moći uvijek "naći". Importiranje u program će biti moguće samo ako su taj program i modul smješteni u istu mapu. Isto tako, u interaktivno sučelje moći ćemo modul importirati samo ako je prije toga bio izveden neki program iz te mape, jer je tako sučelje "naučilo" prepoznati sve datoteke koje se nalaze u toj mapi.

Pogledajmo:

```
>>> from prirodni import *
```

Sve funkcije sad možemo pozivati i u interaktivnom sučelju pri čemu će se pojaviti čak i dokumentacijski string koji opisuje djelovanje funkcije.

```
>>> prost(17)
True
>>> prost(9)
False
```

Za svaki modul koji importiramo u interaktivno sučelje s `import ime_modula` možemo naredbom `help(ime_modula)` ispisati opis modula i popis funkcija s njihovim dokumentacijskim stringovima. Za naš modul `prirodni` taj ispis izgleda ovako:

```
>>> import prirodni
>>> help(prirodni)
Help on module prirodni:

NAME
prirodni

DESCRIPTION
modul prirodni
sadrži funkcije za operacije s prirodnim brojevima

FUNCTIONS
djelitelji(n)
    Funkcija vraća listu svih djelitelja broja n

prost(n)
    Funkcija vraća vrijednosti:
    True   ako je broj prost
    False  ako je broj složen

prosti_brojevi(n)
    Funkcija vraća listu svih prostih brojeva od 2 do n
```

Iz ovog ispisa vidimo ime i opis modula te popis naziva svih funkcija s opisima koje smo napisali u dokumentacijske stringove.

Euklidski algoritam za pronalaženje najvećeg zajedničkog djelitelja

U staroj Grčkoj kao i u Egiptu postojala je potreba za određivanjem površina zemljišta i građevina. Površine su izračunavali na osnovi mjerenja dimenzija zemljišta. Iz tog razloga cijela grana matematike dobila je naziv geometrija (od grčkog "mjerenje zemlje").

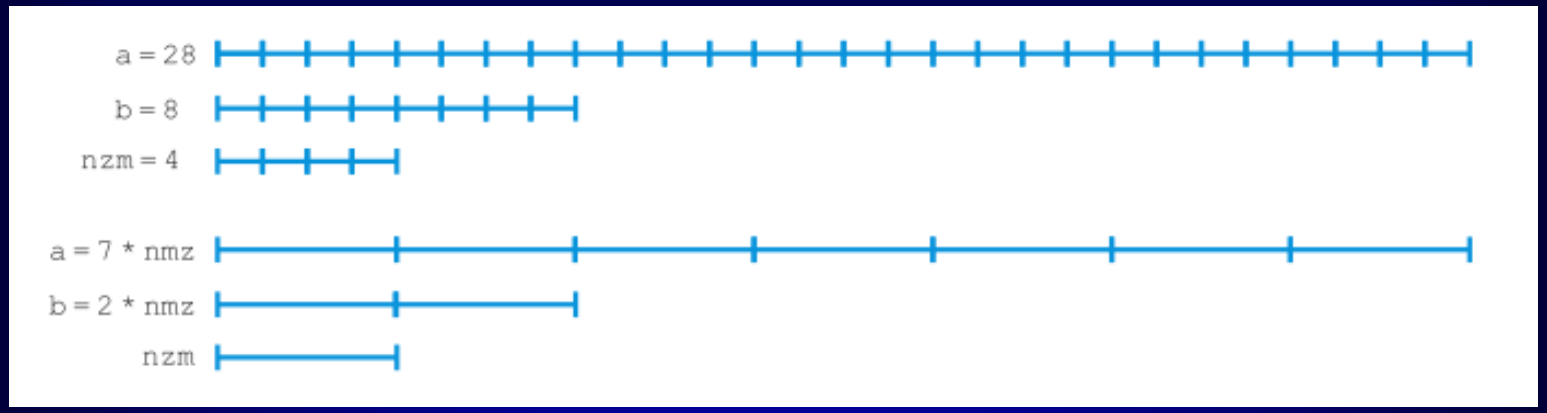
Grci su za mjerenje rabili užad na kojima su u jednakim razmacima bili načinjeni čvorovi. Osim toga za geometrijske konstrukcije koristili su naprave slične današnjem ravnalu i šestaru. Podešenim i učvršćenim kracima šestara prenosili su dužine određene duljine.

Oni su došli na pomisao da se duljina može mjeriti nekom drugom dužinom. Danas takvu dužinu kojom mjerimo drugu dužinu obično zovemo jediničnim dužinom i kažemo da je njezina duljina jednaka 1.

Postavlja se pitanje možemo li načiniti mjeru dulju od jedinične mjere kako bismo manjim brojem pomicanja te nove mjere mogli izmjeriti zadanu dužinu. Primijenimo li znanja o prirodnim brojevima možemo zaključiti da takve mjere moraju biti djelitelji mjerene dužine.

Dalje se postavlja pitanje možemo li za dvije dužine naći mjeru (osim jedinične) kojom ćemo u manje koraka moći izmjeriti obje dužine. I sada možemo zaključiti da je to zajednički djelitelj tih dviju dužina. Nas zanima najveći od tih zajedničkih djelitelja.

Ovdje ćemo uz naziv ***najveći zajednički djelitelj*** rabiti i naziv ***najveća zajednička mjera***



Na slici su prikazane dvije dužine s duljinama `a = 28` i `b = 8`. Lako se možemo uvjeriti da je najveći zajednički djelitelj $D(28, 8) = 4$. Prema tome, najveća zajednička mjera tih dužina je `nzm = 4`. Prema tome, bit će `a = 7 * nzm` i `b = 2 * nzm`.

Grci su znali i sljedeća svojstva najveće zajedničke mjere ili kako mi to danas zovemo najvećeg zajedničkog djelitelja:

1) $D(a, a) = a$

2) $D(a, b) = D(a - b, b)$ ako je $a > b$

3) $D(a, b) = D(b, a)$.

Oko 300. godine pr. Kr. matematičar **Euklid** zapisao je spoznaje grčkih matematičara i napisao djelo *Elementi* koje je postavilo osnove tzv. euklidske geometrije. Po tom djelu se i algoritam za određivanje najveće zajedničke mjere odnosno najvećeg zajedničkog djelitelja naziva euklidskim algoritmom. Princip ovog algoritma je sljedeći:

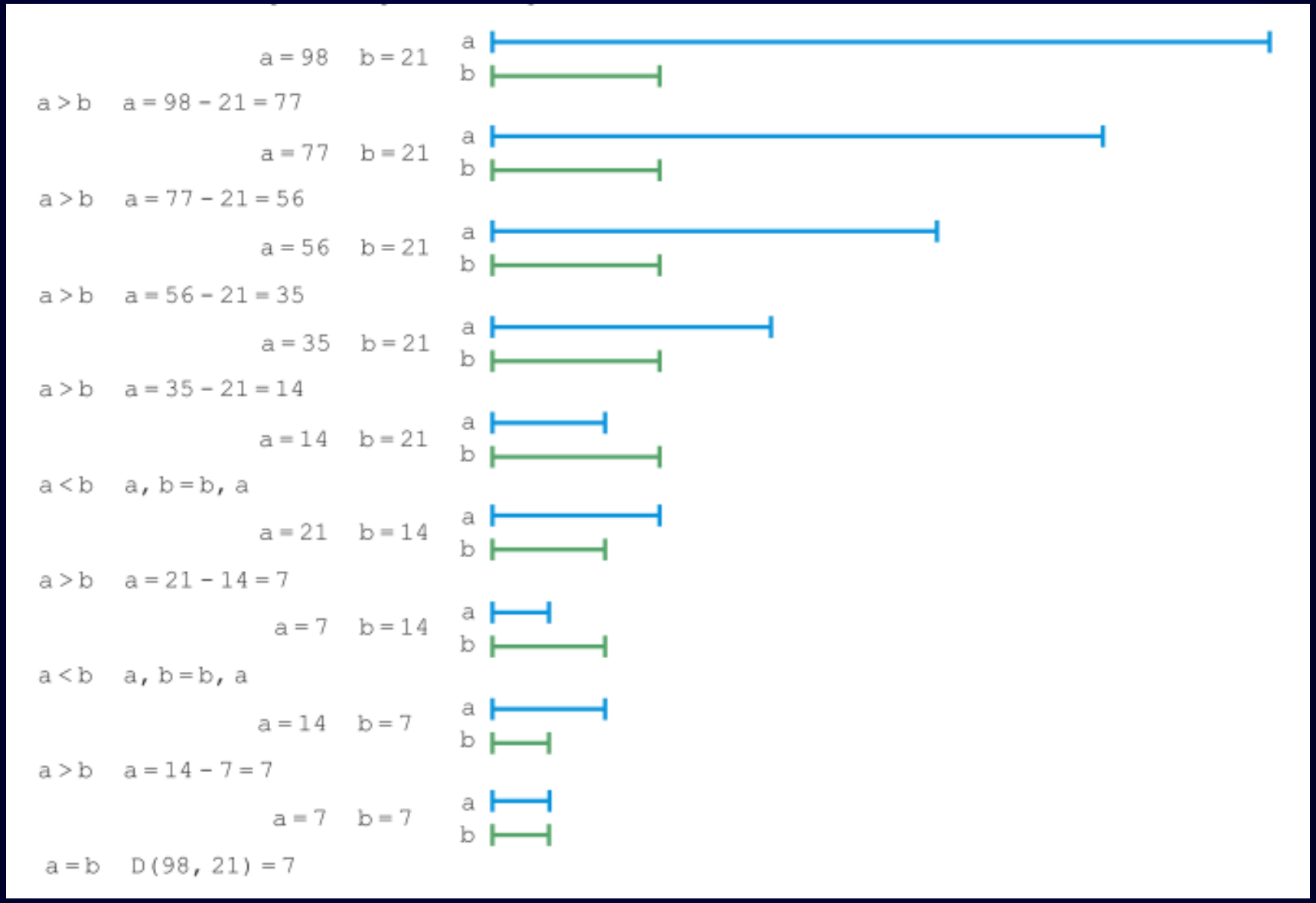
- neka su zadana dva prirodna broja *m* i *n*
- ako su brojevi jednaki, najveća zajednička mjera jednaka je jednom od tih brojeva (bilo kojem jer su brojevi jednaki)
- ako brojevi nisu jednaki, najveća zajednička mjera brojeva *m* i *n* jednaka je najvećoj zajedničkoj mjeri razlike većeg i manjeg broja i manjeg broja
- postupak se ponavlja dok se brojevi ne izjednače.

Napišimo funkciju koja će se koristiti euklidskim algoritmom za izračunavanje najveće zajedničke mjere dvaju prirodnih brojeva.

```
>>> def mjera(a, b):
        while a != b:
                if a > b:
                        a = a - b
                else:
                        b = b - a
        return a

>>> mjera(98, 21)
7
>>>
```

Djelovanje funkcije prikazano je sljedećom slikom:



Taj se postupak može unaprijediti:

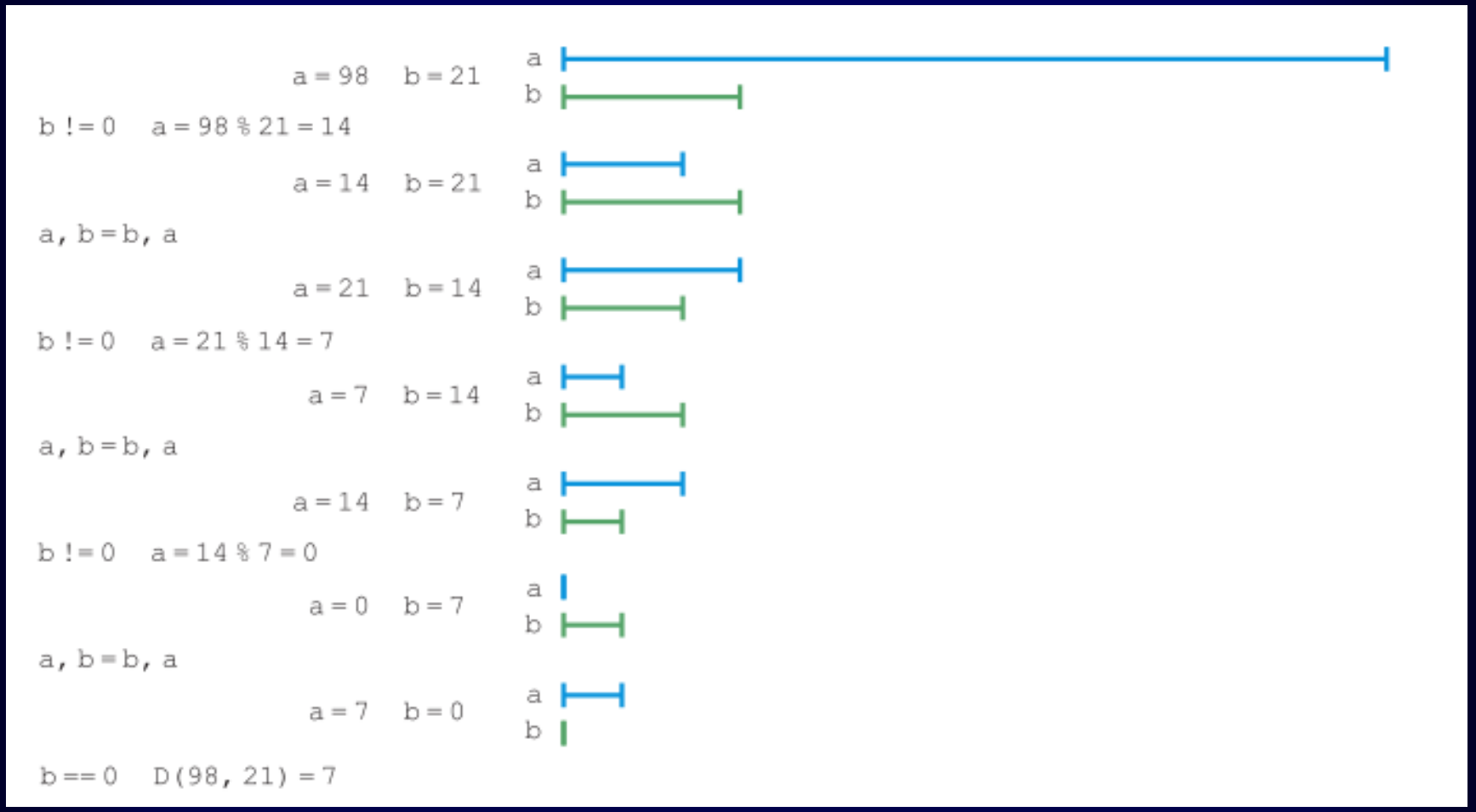
- višekratno oduzimanje može se nadomjesiti dijeljenjem pri čemu nam je važan samo ostatak djeljenja ;
- ostatak cjelobrojong djeljenja je uvijek manji od djelitelja tako da nakon svakog djeljenja bez provjere možemo zamijeniti vrijednosti varijabli `a` i `b` ;
- kada se vrijednosti `a` i `b` izjednače dobit ćemo ostatak jednak nuli i to može biti uvjet za okončanje postupka.

Uzmemo li sve to u obzir, funkcija za izračunavanje najveće zajedničke mjere, tj. najvećeg zajedničkog djelitelja postaje vrlo jednostavna:

```
>>> def nzd(a, b):
        while b != 0:                    #1
            a = a % b                    #2
            a, b = b, a                  #3
        return a
```

Zbog naredbe (#1) petlja `while` se ponavlja se sve dok `b` ne postane jednak nuli. Naredbom (#2) izračunava ostatak dijeljenja broja `a` djeliteljem `b`. Znamo da je ostatak uvijek manji od djelitelja tako prije zamjene vrijednosti varijabli naredbom (#3) ne moramo uspoređivati brojeve `a` i `b`. Nakon te zamjene varijabla `b` sadrži ostatak dijeljenja dok varijabla *a* sadrži vrijednost djelitelja. Prema tome, ako je ostatak jednak nuli tj. uvjet `b != 0` više nije ispunjen, petlja `while` će okončati i funkcija će vratiti najveću zajedničku mjeru (najveći zajednički djelitelj) koja piše u varijabli `a`.

Djelovanje dorađenog euklidskog postupka:



Napišimo funkciju koja će računati najmanji zajednički višekratnik dvaju brojeva koristeći se funkcijom `nzm()`.

Znamo da najveći zajednički višekratnik možemo dobiti tako da umnožak brojeva `a` *i* `b` *podijelimo s njihovim najvećim zajedničkim djeliteljem (najvećom zajedničkom mjerom) i lako ćemo napisati funkciju za njegovo izračunavanje:*

```
>>> def nzv(a, b):
        return a * b // nzd(a, b)
```

Sada možemo provjeriti djelovanje tih funkcija:

```
>>> nzd(234, 378)
18
>>> nzv(234, 378)
4914
```

```
>>> nzd(462, 330)
66
>>> nzv(462, 330)
2310
```

Programski modul za operacije s prirodnim brojevima

U ovom smo poglavlju naučili kako se složeniji problemi lakše rješavaju ako ih razložimo na manje zadatke te kako se pripremaju programske funkcije za njihovo rješavanje. Naučili smo da se funkcije mogu oblikovati na više načina i da za rješavanje pojedinih problema treba odabrati prikladan oblik funkcije.

U poglavlju smo se bavili prirodnim brojevima i razrađivali funkcije za provođenje operacija s prirodnim brojevima. Bilo bi korisno između svih funkcija, koje smo pritom razradili, odabrati one s najboljim svojstvima i smjestiti ih u jedan modul kako bismo ih kasnije mogli pronaći kada nam zatrebaju.

Odabrane funkcije smjestit ćemo u modul nazvan `prirodni_brojevi` i opremiti ga na uobičajeni način. Za sve funkcije napisat ćemo dokumentacijske stringove i dodat ćemo u njegov dio iza ispitivanja uvjeta `__name__ == '__main__'` niz naredbi kojima se može demonstrirati djelovanje pojedinih funkcija ako se modul pokrene kao glavni program. Znamo već da se te naredbe neće izvoditi kada modul bude importiran u neki drugi program ili u interaktivno sučelje.

Iako smo već ustanovili da je umjesto funkcije `eratosten()` bolje koristiti se funkcijom `prosti_brojevi()` jer je ona mnogo brža, uvrstit ćemo i nju iz povijesnih razloga u ovaj modul.

U modul su uključene i funkcije `prosti_faktori()` i `zajednički_faktori()` koje ovdje nisu posebno razmatrane.

```python
#Modul s funkcijama za operacije s prirodnim brojevima
#prirodni_brojevi.py
'''
   Modul prirodni_brojevi sadrži funkcije
   za operacije s prirodnim brojevima
'''
def eratosten(n):
    '''Funkcija određuje listu prostih brojeva manjih ili jednakih n.
       Njezino je trajanje izvođenja znatno dulje od izvođenja funkcije
       prosti_brojevi(). Uključena je u ovaj modul zbog povijesnih
       razloga ali i zbog mogućnosti usporedbe tih dvaju funkcija.
    '''
    popis = [i for i in range(2, n + 1)]
    k = 0
    while popis[k] ** 2 <= n + 1:
        for j in range(2, n // popis[k] + 1):
            t = popis[k] * j
            if t in popis:
                popis.remove(t)
        k += 1
    return popis
```

```
def prost(n):
    '''Funkcija vraća vrijednosti:
          True   ako je broj prost
          False  ako je broj složen
    '''
    d = 2
    while d ** 2 <= n:
        if n % d == 0:
            return False
        d += 1
    return True

def prosti_brojevi(n):
    '''Funkcija vraća listu prostih brojeva manjih ili jednakih n'''
    return  [i for i in range(2, n + 1) if prost(i)]

def djelitelji(n):
    '''Funkcija vraća listu svih djelitelja broja n'''
    djel = []
    for d in range(1, n + 1):
        if n % d == 0:
            djel.append(d)
    return djel
```

```python
def prosti_faktori(n):
    '''Funkcija vraća listu prostih faktora broja n'''
    r = []
    prosti = prosti_brojevi(n)
    i = 0
    while n > 1:
        if n % prosti[i] == 0:
            r.append(prosti[i])
            n //= prosti[i]
        else:
            i += 1
    return r

def zajednički_faktori(n, m):
    '''Funkcija vraća listu zajedničkih prostih faktora brojeva m i n'''
    r = []
    prosti = prosti_brojevi(min(m, n))
    i = 0
    while i < len(prosti):
        if m % prosti[i] == 0 and n % prosti[i] == 0:
            r.append(prosti[i])
            m //= prosti[i]
            n //= prosti[i]
        else:
            i += 1
    return r
```

```
def nzd(m, n):
    '''Funkcija vraća najveći zajednički djelitelj brojeva m i n'''
    while n != 0:
        m = m % n
        m, n = n, m
    return m

def nzv(m, n):
    '''Funkcija vraća najmanji zajednički višekratnik brojeva m i n'''
    return m * n // nzd(m, n)

if __name__ == '__main__':
    m = int(input('Upiši broj m = '))
    n = int(input('Upiši borj n = '))
    print('U intervalu [2, {}] prosti brojevi su: \n{}\n'\
          .format(m, prosti_brojevi(m)))
    print('Djelitelji broja {} su:\n{}\n'.format(m, djelitelji(m)))
    print('Prosti faktori broja {} su:\n{}\n'.format(m, prosti_faktori(m)))
    print('Prosti faktori broja {} su:\n{}\n'.format(n, prosti_faktori(n)))
    print('Zajednički prosti faktori brojeva {} i {} su:\n{}\n'\
          .format(m, n, zajednički_faktori(m, n)))
    print('Najveći zajednički djelitelj brojeva {} i {} je: {}\n'\
          .format(m, n, nzd(m, n)))
    print('Najmanji zajednički višekratnik brojeva {} i {} je: {}'\
          .format(m, n, nzv(m, m)))
```

Ako modul izvedemo kao glavni program, dobit ćemo uz utipkavanje brojeva 234 i 378 sljedeći ispis:

```
Upiši broj m = 234
Upiši broj n = 378
U intervalu [2, 234] prosti brojevi 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]

Djelitelji broja 234 su:
[1, 2, 3, 6, 9, 13, 18, 26, 39, 78, 117, 234]

Prosti faktori broja 234 su:
[2, 3, 3, 13]

Prosti faktori broja 378 su:
[2, 3, 3, 3, 7]

Zajednički prosti faktori brojeva 234 i 378 su:
[2, 3, 3]

Najveći zajednički djelitelj brojeva 234 i 378 je: 18

Najmanji zajednički višekratnik brojeva 234 i 378 je: 4914
```

Modul se može importirati u interaktivno sučelje:

```
>>> from prirodni_brojevi import *
```

i nakon toga pozivati njegove funkcije:

```
>>> djelitelji(4385)
[1, 5, 877, 4385]
>>> prosti_faktori(462)
[2, 3, 7, 11]
>>> prosti_faktori(330)
[2, 3, 5, 11]
>>> nzd(462, 330)
66
>>> nzv(462, 330)
2310
```