Programmēšanas valoda

iesācējiem

1. daļa. Ievads *Python*

Saturs

Ievads

Katram zināms, ka mūsdienu datori spēj veikt daudzas un dažādas lietas – sākot ar mājas lietotājam pierastām – atskaņot mūziku, filmas, spēlēt spēles, veidot profesionāli noformētus dokumentus, un beidzot ar milzīgu datu apjomu apstrādi bankās, lidostās u.c.

Zināms arī, ka visas šīs lietas dators nemāk veikt "kopš dzimšanas" - šim nolūkam nepieciešamas programmas – secīgi pierakstītas darbības, kuras dators prot izpildīt, lai sasniegtu vēlamo rezultātu.

Datoram saprotamās darbības jeb komandas ir ļoti vienkāršas – dators prot saskaitīt un atņemt, ierakstīt skaitļus operatīvajā atmiņā un nolasīt no tās, kā arī sazināties ar savām perifērijas iekārtām – cieto disku, kompaktdisku vai DVD iekārtu, videokarti (kura videosignālus tālāk nosūta monitoram), printeri u.c. Datoru pirmsākumos komandas tika ievadītas kā ciparu "1" un "0" virknītes, jo šie divi cipari ir vienīgie, ko datora procesors "atpazīst". Vēlāk katra šāda virknīte, kas nozīmēja kādu komandu, tika aizstāta ar noteiktu, cilvēkam saprotamāku apzīmējumu – tā radās **asemblera** valoda. Tas tomēr programmēšanas darba apjomu sevišķi nesamazināja, jo komandas jeb darbības tāpat palika ļoti primitīvas. 1950.gados radās t.s. **augstā līmeņa** programmēšanas valodas, kurās instrukcijas tika pierakstītas programmētajam "saprotamākā" formā – darbības kļuva sarežģītākas un tika pierakstītas ar instrukcijām, kas atgādina angļu valodas vārdus. Pēc tam šīs instrukcijas tika "pārveidotas" jeb **translētas** uz datoram saprotamu valodu un pēc tam izpildītas. Šādas valodas sauc par **kompilatoriem**. Cita programmēšanas valodu klase ir t.s. **interpretatori** – valodas, kurās katra darbība tiek secīgi translēta un izpildīta. Šādas programmas parasti ir pieejamas to **pirmkoda** jeb sākotnējā teksta veidā – jebkurš tās var aplūkot un modificēt. Pie tādām pieder arī programmēšanas valoda **Python**. Šī valoda ir ļoti jaudīga – tajā iespējams gan iemācīties programmēt, gan veidot nopietnas informācijas sistēmas, *Internet* mājaslapas u.c.

Grāmatā aprakstītās programmas paredzētas izpildei *GNU/Linux* operētājsistēmā. Izmantojot citu operētājsistēmu (piemēram, *Microsoft Windows*), atsevišķās programmās, iespējams, jāveic izmaiņas. Šīs programmas iespējams sagatavot ar jebkuru tekstu redaktoru (piemēram, *KWrite* vai *Notepad*), programmas teksta faila nosaukumu papildinot ar **.py**, piemēram, **programma1.py**. Pēc tam šīs programmas iespējams izpildīt **Python** vidē:

1) *GNU/Linux* operētājsistēmā programmu jāatver un jāizpila *Python IDLE* vidē vai komandrindā jāizpilda komandas

```
# chmod 755 programma1.py
# ./programma1.py
```

2) *Microsoft Windows* operētājsistēmā programma jāatver un jāizpilda *Python IDLE* vidē vai komandrindā jāizpilda komandu

```
C:\> c:\python\python programma1.py
```

Vienkāršākās aritmētikas izteiksmes

Sākumam izveidosim vienkāršu **Python** programmiņu, kura saskaitīs divus skaitļus un izvadīs rezultātu.

```
#!/usr/bin/python
print 3+4
```

Programmas pirmajā rindiņā tiek norādīts, ka tā jāizpilda ar **Python** valodas interpretatoru, bet otrajā – uz ekrāna tiek izvadīts aprēķinu rezultāts. To veic operators **print** (drukāt). Izmantojot šo operatoru, līdzīgi iespējams izvadīt uz ekrāna tekstu:

```
#!/usr/bin/python
print "Labrit"
```

un arī teksta un aprēķinu rezultātu kombinācijas, tās atdalot ar komatu:

```
#!/usr/bin/python
print "2 * 2 =",2*2
```

Programmēšanas valodā **Python** ir pieejamas šādas aritmētiskās darbības:

\+ - **saskaitīšana**

– - **atņemšana**

* - **reizināšana**

/ - **dalīšana**

% - atlikums no dalīšanas

** - **kāpināšana, piemēram, 2^3 pieraksta 2**3**

Ir iespējams pierakstīt arī arežģītākas izteiksmes, kā piemēram:

$$\frac{(5+7)\cdot 4}{2}$$

Pieraksts izskatīsies šādi:

```
(5+7)*4/2
```

Matemātiskās funkcijas

Python valodā ir iespējams izmantot dažādas matemātiskās funkcijas un konstantes (piemēram, π, kas tiek apzīmēts kā **pi**, vai funkciju **sqrt**, kura aprēķina kvadrātsakni), ja programmā tiek ievietots īpašs operators, kurš ielādē datora atmiņā atbilstošo **Python** moduli – **math**:

```
from math import *
```

Piemērs:

```
#!/usr/bin/python
from math import *
print pi
print sqrt(16)
```

Veicot aprēķinus ar parastu kalkulatoru, sarežģītāku aprēķinu gadījumā nepieciešams starprezultātus pierakstīt uz papīra. Veidojot programmu **Python** valodā, starprezultātiem iespējams piešķirt nosaukumus, t.i., tos **saglabāt mainīgajos**:

```
#!/usr/bin/python
from math import *
x = sqrt(16)
y = x+1
print x
print y
```

Šajā piemērā programma veic aprēķinus, izrēķinot kvadrātsakni no 16. Rezultāts tiek piešķirts mainīgajam x. Šim rezultātam tiek pieskaitīts 1, un jaunais rezultāts tiek saglabāts mainīgajā y. Šie mainīgie saturēs izrēķinātās vērtības līdz brīdim, kamēr tiem tiks piešķirtas jaunas vērtības, vai arī programma beigs darbu. Piemērā mainīgo vērtības tiek izdrukātas uz ekrāna.

Vēl kāds piemērs: ja dots riņķa līnijas rādiuss (apzīmēsim to ar **r**), tad riņķa laukumu **S** aprēķina pēc formulas

$$S = \pi \cdot R^2$$

Izveidosim programmu **Python** valodā, kas veic šādu aprēķinu:

```
#!/usr/bin/python
from math import *
r = 3
S = pi*r**2
print S
```

Pieņemot, ka riņķa līnijas rādiuss ir 3, šajā programmā mainīgajam **r** tiek piešķirta atbilstoša vērtība. Protams, to ir iespējams aizstāt ar jebkuru nepieciešamo vērtību. Pēc tam tiek izrēķināts laukuma vērtība, kura tiek piešķirta mainīgajam **S**. Programmas pēdējā rindiņā aprēķinātais laukums tiek izvadīts uz ekrāna.

Komentāri

Veidojot pietiekoši sarežģītas programmas, bieži rodas nepieciešamība komentēt tādu vai citādu operatoru. Šim nolūkam valodā **Python** ir paredzēts īpašs pieraksta formāts – **komentārs**. Par komentāru tiek uzskatīts brīvi pierakstīts teksts, kurš sākas ar simbolu #. Komentārs var sākties jebkurā programmas rindiņas vietā, un turpinās līdz rindiņas beigām.

Lai ilustrētu komentāru pielietošanu, komentēsim iepriekš izveidoto programmu:

```
#!/usr/bin/python
from math import * # importejam matematisko funkciju moduli
# Rinka linijas radiuss
r = 3
# Aprekinam laukumu
S = pi*r**2
# Izdrukajam rezultatu
print S
```

Piezīme: komentāros nav iespējams izmantot latviešu speciālos simbolus, tādēļ, piemēram, vārdu “Programmēšana” jāpieraksta kā “Programmesana” vai “Programmeeshana”, izmantojot tikai latīņu alfabēta burtus.

Interaktīvu programmu veidošana

Lai noskaidrotu, kas ir **interaktīvas programmas**, mēģināsim izpildīt šādu uzdevumu:

Izmantojot iepriekšējā sadaļā izveidoto programmu riņķa laukuma aprēķināšanai, aprēķināt riņķa laukumu riņķa līnijām ar šādiem rādiusiem:
4, 15, 2, 6, 10, 25, 48, 35, 64.

Liekas, nekas īpašs – atliek vien programmā veikt izmaiņas rindiņā, kurā tiek piešķirta vērtība mainīgajam **r**... Tomēr – vai ērtāk nebūtu, ja mainīgajam **r** varētu piešķirt vērtību, nemainot programmas tekstu? Izrādās – var. Šim nolūkam programmēšanas valodā **Python** ir paredzēta funkcija **input()**. Šī funkcija uz brīdi aptur programmas darbību, pieprasot ievadīt informāciju, izmantojot datora tastatūru. Funkcijas rezultāts ir ievadītā informācija. Pārveidosim mūsu programmu tā, lai tā kļūtu **interaktīva**, t.i., lai tā savas darbības laikā "sazinātos" ar lietotāju, iegūstot papildinformāciju:

```
#!/usr/bin/python
from math import * # importejam matematisko funkciju moduli
# Rinka linijas radiuss
print "Ievadiet radiusu"
r = input()
# Aprekinam laukumu
S = pi*r**2
# Izdrukajam rezultatu
print S
```

Izpildot šo programmu, tā izvadīs tekstu **"Ievadiet radiusu"** un apstāsies, gaidot lietotāja reakciju. Rādiusa vērtība tiek ievadīta, izmantojot datora tastatūru. Kad informācija ievadīta, lietotājam jānospiež taustiņu **"Enter"**. Tad programma veic tālākos aprēķinus un izvada rezultātu:

```
Ievadiet radiusu
15
706.858347058
```

Moduļu ielāde

Iepriekšējos piemēros tika izmantotas papildfunkcijas no **Python** moduļa **math**. Lai tās varētu izmantot, **math** modulis katras programmas sākumā tika ielādēts datora atmiņā, izmantojot operatoru

```
from math import *
```

Pēc ielādes bija iespējams izmantot visas funkcijas, kuras iebūvētas modulī **math**. Bieži vien ir izdevīgāk izmantot otru **Python** piedāvāto metodi moduļu ielādei un tajos iebūvēto funkciju izmantošanai. Izmantojot šo metodi, moduļa **math** ielādei jāizmanto operatoru

```
import math
```

Savukārt, lai izmantotu funkciju **pi** no šī moduļa, jāizmanto operatoru

```
math.pi
```

Lai aprēķinātu skaitļa 4 kvadrātsakni, jāizmanto operatoru

```
math.sqrt(4)
```

utt.

Uzdevums: pārrakstiet iepriekš izveidoto programmu tā, lai tā izmantotu tikko aplūkoto metodi moduļu ielādei un to funkciju izmantošanai!

Funkcijas

Mēs jau esam izveidojuši vairākas programmas, tomēr tās visas izskatās paredzētas "vienreizējai lietošanai" – ja nepieciešams riņķa laukumu izrēķināt vairākas reizes vienā programmā, nāksies vairākas reizes pierakstīt vienādu programmas fragmentu, kurš veiks aprēķinus. Par laimi, valodā **Python** iespējams vairākkārt izmantot vienu un to pašu programmas fragmentu, ja tas tiek pierakstīts kā **apakšprogramma**. Apakšprogramma ir programmas fragments, kurš risina apakšuzdevumu – programmas kopējā uzdevuma daļu.

Python apakšprogrammas sauc arī par **funkcijām** un tās definē ar operatoru **def**. Pārējā apakšprogrammas daļa tiek pierakstīta ar atkāpi no kreisās malas. Shematiski tas izskatās šādi:

```
def nosaukums(argumenti):
  funkcijas operatori
  return rezultāts
```

Operators **def** radies no angļu valodas darbības vārda ***to define*** (definēt, noteikt) un **return** nozīmē "atgriezt". Katrā funkcijā var izmantot vairākus **return** operatorus, bet var arī neizmantot nevienu.

Lai ilustrētu funkciju izmantošanu, pārrakstīsim mūsu programmu riņķa laukuma aprēķināšanai, izmantojot funkciju:

```
#!/usr/bin/python
import math # importejam matematisko funkciju moduli
# Funkcija rinka linijas radiusa aprekinasanai
def Laukums(r):
  S = math.pi*r**2
  return S
# Funkcijas beigas
# Rinka linijas radiuss
print "Ievadiet radiusu"
r = input()
# Aprekinam laukumu
S = Laukums(r)
# Izdrukajam rezultatu
print S
```

Izmantojot funkciju **Laukums**, ir iespējams ērti papildināt šo programmu, lai, piemēram, tā pieprasītu ievadīt divus riņķa līniju rādiusus, aprēķinātu to laukumus un parādītu, cik reizes pirmais ievadītais laukums ir lielāks par otro:

```
  #!/usr/bin/python
  import math # importejam matematisko funkciju moduli
  # Funkcija rinka linijas radiusa aprekinasanai
  def Laukums(r):
    S = math.pi*r**2
    return S
  # Rinka liniju radiusi
  print "Ievadiet radiusu 1"
  r1 = input()print "Ievadiet radiusu 2"
  r2 = input()
  # Aprekinam laukumus
  S1 = Laukums(r1)
  S2 = Laukums(r2)
  # Izdrukajam rezultatu
  print "Rinka ar radiusu", r1, "laukums ir", Laukums(r1)/Laukums(r2), "reizes
lielaks par rinki ar radiusu", r2
```

Šajā gadījumā lietotāja dialogs ar programmu izskatīsies šādi:

```
  Ievadiet radiusu 1
  15
  Ievadiet radiusu 2
  11
  Rinka ar radiusu 15 laukums ir 1.85950413223 reizes lielaks par rinki ar
radiusu 11
```

Teksta rindas

Līdz šim mēs nodarbojāmies praktiski tikai ar skaitļu apstrādi, veicot aprēķinus. Tomēr datora iespējas ar to neaprobežojas. **Python** ļauj apstrādāt arī teksta informāciju.

Teksta rindas sastāv no atsevišķiem simboliem, katrs no kuriem apzīmē kādu burtu, skaitli, speciālo simbolu utt. Bez tam atsevišķiem simboliem ir īpaša nozīmē un tie neapzīmē nevienu "redzamu" simbolu. Piemēram, tāds simbols ir jaunas rindas simbols, kurš apzīmē vietu, kurā jāveic rindas pārnesumu. Valodā **Python** to apzīmē ar diviem "redzamajiem" simboliem – **\n**. Katram simbolam ir arī skaitlisks apzīmējums (kods) – vesels skaitlis no 0 līdz 255. Simboli ar kodiem no 0 līdz 127 ir standartizēti un tiek saukti par simbolu kopu **ASCII** (*American Standard Code for Information Interchange* – amerikāņu standarta kods informācijas apmaiņai). Pārējos simbolus ar kodiem no 128 līdz 255 var izmantot dažādi, tomēr tie parasti tiek izmantoti nacionālo alfabētu speciālo simbolu attēlošanai, piemēram, latviešu specifiskajiem burtiem – **ā, š, č**, utt.. Valodā **Python** ir paredzētas divas funkcijas, kuras ļauj uzzināt simbola kodu un otrādi – pēc koda noteikt simbolu. Aplūkosim tās, izveidojot šādu programmu:

```
#!/usr/bin/python
print ord("A")
print chr(65)
```

Programmas izpildes rezultāts būs

```
65
A
```

Funkcijas **ord** nosaukums radies no angļu valodas vārda ***order*** (kārtas numurs, kārtība). Šīs funkcijas arguments ir simbols un rezultāts – tā kārtas skaitlis **ASCII** tabulā (kods). Funkcijas **chr** nosaukums, savukārt, radies no angļu valodas vārda ***character*** (simbols). Šīs funkcijas arguments ir simbola skaitliskais kods un rezultāts – pats simbols.

Ar teksta rindām iespējams veikt līdzīgas darbības, kā ar skaitļiem:

\+ - **savienošana jeb konkatenācija**

* - **pavairošana jeb atkārtošana**

Piemēram, programmas

```
#!/usr/bin/python
print "Alise"+" "+"brinumzeme"
print "-"*16
```

rezultāts būs

```
Alise brinumzeme
----------------
```

Šajā programmā vispirms trīs rindas tiek savienotas vienā un pēc tam – viena rinda tiek atkārtota 16 reizes.

Teksta rindām, līdzīgi kā skaitļiem, iespējams piešķirt nosaukumus:

```
#!/usr/bin/python
rinda1="Aiz"
rinda2="spoguli"
rinda3="ja"
teksts=rinda1+rinda2+rinda3
print teksts
teksts=rinda1+rinda2+"s"
print teksts
```

Šīs programmas izpildes rezultāts būs

```
Aizspogulija
Aizspogulis
```

Tālāk aplūkosim dažādas **Python** valodas funkcijas, kuras paredzētas teksta rindu apstrādei.

Teksta rindas garumu iespējams uzzināt, izmantojot funkciju **len**. Tās nosaukums radies no angļu valodas vārda ***length*** (garums). Piemēram:

```
#!/usr/bin/python
teksts="Aizspogulija"
print len(teksts)
```

Programmas izpildes rezultāts būs

```
12
```

Pārbaudot, konstatējam, ka vārdā "Aizspogulija" tiešām ir 12 simboli.

Programmēšanas valodā **Python** ir iespējams arī "izgriezt" daļu teksta no rindas. Aplūkosim vairākus piemērus:

```
#!/usr/bin/python
teksts="Aizspogulija"
print "Izgriezam no 2. līdz 7. simbolam:", teksts[1:7]
print "Izgriezam no 4. simbola lidz rindas beigam:", teksts[3:]
print "Izgriezam no rindas sakuma lidz 3. simbolam:", teksts[:3]
print "Izgriezam rindas 3. simbolu:", teksts[2]
```

Šim piemēram būs nepieciešami daži paskaidrojumi. **Python** valoda skaita nevis simbolus pēc kārtas, bet "atstarpes" starp simboliem:

A	I	Z	S	P	O	G	U	L	I	J	A	
← 0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9	← 10	← 11	← 12

Papildus tam, kā redzams zīmējumā, "atstarpes" sāk skaitīt no 0, nevis 1.

Lai no mainīgā teksts "izgrieztu" daļu, ja zināma gan sākuma, gan beigu "atstarpe", jāizmanto **Python** konstrukciju

```
teksts[sākums:beigas]
```

kur ***sākums*** un ***beigas*** ir sākuma un beigu "atstarpju" numuri. Mūsu piemērā starp "atstarpēm" ar numuriem 1 un 7 atrodas teksts "izspog".

Ja beigu "atstarpe" nav norādīta, t.i., konstrukcija tiek pierakstīta formā

```
teksts[sākums:]
```

Python uzskata, ka domāta pēdēja "atstarpe" teksta rindā.

Ja nav norādīta sākuma "atstarpe", t.i., konstrukcija tiek pierakstīta formā

```
teksts[:beigas]
```

Python uzskata, ka domāta atstarpe ar numuru 0, t.i., to pašu var pierakstīt formā

```
teksts[0:beigas]
```

Lai no teksta rindas iegūtu vienu simbolu, kurš seko noteiktai atstarpei, jāizmanto

konstrukciju

```
teksts[atstarpe]
```

Mūsu piemērā pēc atstarpes ar numuru 2 seko burts "Z".

Šāds pieraksts sākumā var likties nedaudz dīvains, tomēr, pierodot pie tā, izrādās, ka nekas sarežģīts šādā pieraksta veidā nav. Lai atvieglotu izpratni par **Python** teksta rindu simbolu numerāciju, varam uzskatīt, ka katram rindas simbolam ir savs kārtas skaitlis, sākot ar 0. Gadījumos, kad jānorāda vairāk, kā viens simbols teksta rindā, sākumsimbolu norāda pēc tā kārtas numura un beigu simbolu – pēc tā kārtas numura, pieskaitot 1. Piemēram, lai norādītu pēdējos četrus simbolus vārdā "Aizspogulija", vispirms konstatējam, ka pēc kārtas jānorāda simboli no 9. līdz 12. Tā kā pirmajam vārda simbolam ir kārtas skaitlis 0, tad jānorāda simboli ar kārtas skaitļiem no 8 līdz 11. Tomēr iepriekš vienojāmies, ka beigu simbola kārtas numuram tiek pieskaitīts 1. Tas nozīmē, ka **Python** valodā atbilstošais pieraksts izskatīsies

```
[8:12]
```

Cita interesanta **Python** valodas funkcija ļauj saskaitīt, cik reizes noteikts teksta fragments parādās citā teksta rindā. Ilustrēsim to ar piemēru:

```
#!/usr/bin/python
import string
teksts = "Mes, saproties, ar vechiem, saproties, aizgajam, saproties,
pastaigat, saproties."
skaits = string.count(teksts, "saproties")
print "Vards 'saproties' atkartojas", skaits, "reizes."
```

Šīs programmas izpildes rezultāts būs

```
Vards 'saproties' atkartojas 4 reizes.
```

Funkcija, kura veic šo "saskaitīšanu", ir iebūvēta modulī **string**, tādēļ vispirms šo moduli jāielādē datora atmiņā. Pēc tam funkciju iespējams izmantot:

```
string.count(teksts, apaksteksts)
```

kur **teksts** ir mainīgais vai teksta rinda, kurā jāmeklē, cik reizes tajā atkārtojas mainīgais vai teksta rinda **apaksteksts**.

Vēl kāda funkcija ļauj atrast pozīciju ("atstarpes" numuru), kurā noteiktā rindā pirmo reizi sastopama noteikta apakšrinda:

```
#!/usr/bin/python
import string
teksts = "Mes, saproties, ar veciem, saproties, aizgajam, saproties,
pastaigat, saproties."
pozicija = string.find(teksts, "saproties")
print "Vards 'saproties' pirmo reizi atrodams pozicija", pozicija
```

Programmas izpildes rezultāts būs

```
Vards 'saproties' pirmo reizi atrodams pozicija 5
```

Pārbaudot, tiešām konstatējam, ka pirmais vārds "saproties" mūsu teksta rindā sākas 5.pozīcijā.

Lai teksta rindā aizvietotu noteiktu apakšrindu ar citu apakšrindu (līdzīgi, kā tekstu redaktoros darbojas funkcija "Meklēt/Aizvietot" (*Find/Replace*), varam izmantot šādu funkciju:

```
#!/usr/bin/python
import string
teksts = "Mes, saproties, ar veciem, saproties, aizgajam, saproties,
pastaigat, saproties."
jaunaisteksts = string.replace(teksts, "saproties", "zinies")
print teksts
print jaunaisteksts
```

Programmas darbības rezultāts būs

```
Mes, saproties, ar veciem, saproties, aizgajam, saproties, pastaigat,
saproties.
Mes, zinies, ar veciem, zinies, aizgajam, zinies, pastaigat, zinies.
```

Uzdevums: izskaidrojiet šīs funkcijas darbību!

Nosacījumi

Nevienu daudz maz nopietnu programmu nav iespējams izveidot, neizmantojot tajā **lēmumu pieņemšanu**. Piemēram, kad Jūs strādājat ar tekstu redaktoru, kurā ir nesaglabāts dokuments un nolemjat to aizvērt, tekstu redaktors Jums pārjautās, vai tiešām Jūs nevēlaties saglabāt savu sagatavoto dokumentu. Atkarībā no Jūsu atbildes dokuments tiek vai arī netiek saglabāts – tiek pieņemts **lēmums** veikt vai neveikt šo darbību.

Programmēšanas valodā **Python** lēmumu pieņemšanu iespējams veikt, izmantojot operatoru **if** (angļu valodā tas nozīmē – "*ja*"). Shematiski šo operatoru iespējams attēlot šādi:

```
if nosacījums:
  Operatori, kas jāizpilda, ja nosacījums izpildās.
  Operatori tiek pierakstīti ar atkāpi.
else:
  Operatori, kas jāizpilda, ja nosacījums neizpildās.
```

Gadījumos, ja jāveic darbības, izpildoties nosacījumam, bet nav jāveic nekas, neizpildoties, lēmuma pieņemšanas konstrukcijas daļu **else** iespējams izlaist.

Nosacījumos izmantojamās salīdzināšanas operācijas:

> - **lielāks**

< - **mazāks**

== - **vienāds**

>= - **lielāks vai vienāds**

<= - **mazāks vai vienāds**

!= - **nav vienāds**

Vienkāršs piemērs:

```
#!/usr/bin/python
print "Ievadiet skaitli"
skaitlis = input()
if skaitlis>=0:
  print "Pozitivs skaitlis"
else:
  print "Negativs skaitlis"
```

Šī ir interaktīva programma, un lietotāja dialogs ar to varētu izskatīties šādi:

```
Ievadiet skaitli
10
Pozitivs skaitlis
```

vai šādi:

```
Ievadiet skaitli
-5
Negativs skaitlis
```

Izmantojot papildus atkāpes no kreisās malas, **Python** programmā iespējams dažādas konstrukcijas ievietot "vienu otrā":

```
#!/usr/bin/python
print "Ievadiet skaitli"
skaitlis = input()
if skaitlis>=0:
  if skaitlis==0:
    print "Nulle"
  else:
    print "Pozitivs skaitlis"
else:
  print "Negativs skaitlis"
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Python loģikas pamati

Jebkurš mūsdienu dators darbojas pēc stingriem loģikas likumiem. Zinātnieki ir izstrādājuši daudz un dažādus loģikas paveidus, tomēr datoros visērtāk ir izmantot **Būla loģiku** *(Boolean logic),* kas nosaukta par godu 19.gadsimta angļu matemātiķim Džordžam Būlam (*Bool*). Šajā loģikā tiek izmantotas divas iespējamās vērtības, kuras nosauksim **"patiesība"** un **"meli"** *(**true** un **false**)*. Lai saīsinātu pierakstu, apzīmēsim tās kā **1** un **0**. Loģikas formulās visbiežāk tiek izmantotas šādas operācijas: **noliegums** (**loģiskais NĒ**), **disjunkcija** (**loģiskais VAI**) un **konjunkcija** (**loģiskais UN**).

Noliegumu Python apzīmē kā **not** un tas darbojas ļoti vienkārši – attiecinot **not** uz kādu vērtību, "patiesība" kļūst par "meliem" un "meli" - par "patiesību":

not 0 → 1

not 1 → 0

Disjunkcijas (tiek pierakstīta kā **or**) izpildē tiek izmantoti divi argumenti, un tās rezultāts būs "meli" tikai tad, ja abi argumenti būs "meli":

0 or 0 → 0

0 or 1 → 1

1 or 0 → 1

1 or 1 → 1

Divu argumentu **konjunkcija** (**and**) būs "patiesība" tikai tad, ja abi argumenti būs "patiesība":

0 and 0 → 0

0 and 1 → 0

1 and 0 → 0

1 and 1 → 1

Piemēram, ja dots mainīgais **A**, kurš satur skaitlisku vērtību un nepieciešams veikt kādas darbības tikai gadījumā, ja **A** ir mazāks par 10, bet lielāks par 0 (t.i., A ir skaitlis no 1 līdz 9), atbilstoša **Python** programma var izskatīties šādi:

```
#!/usr/bin/python
print "Ievadiet A"
A=input()
if A>0:
  if A<10:
    print "Sveiki! 0<A<10."
```

Tomēr, izmantojot **konjunkciju**, programmas teksts ir daudz pārskatāmāks:

```
#!/usr/bin/python
print "Ievadiet A"
A=input()
if A>0 and A<10:
    print "Sveiki! 0<A<10."
```

Līdzīgā veidā ir iespējams vienā izteiksmē kombinēt konjunkciju, disjunkciju un noliegumu, iekļaujot vienā formulā sarežģītu loģiku.

Uzdevums: sastādiet loģikas izteiksmi, kuras rezultāts ir "patiesība" tikai tad, ja mainīgais A ir pozitīvs pāra skaitlis vai jebkurš skaitlis, kurš mazāks par 0!

Cikli

Galvenā datoru izmantošanas priekšrocība ir, ka tie spēj nenogurstot nepārtraukti atkārtot vienas un tās pašas darbības. Šajā sadaļā aplūkosim **ciklus** – konstrukcijas, kuras ļauj vairākkārt izpildīt vienu un to pašu programmas fragmentu nepieciešamo reižu skaitu. Katru tādu atkārtojumu bieži vien sauc par **iterāciju**.

Programmēšanas valodā **Python** ir divu veidu cikli: **while** (***kamēr***) cikls un **for** ("***priekš***") cikls.

Pirmais no ciklu veidiem veic nepieciešamās darbības tik ilgi, kamēr izpildīsies noteikts nosacījums. **Python** šādi cikli tiek pierakstīti formā

```
while nosacījums:
  cikla operatori
```

Tas nozīmē, ka, kamēr izpildīsies nosacījums (tas tiek pierakstīts līdzīgi, kā lēmuma pieņemšanas operatorā **if**), tiks atkārtoti izpildīti cikla operatori. Lai ilustrētu **while** cikla izmantošanu, izveidosim programmu, kura pieprasa ievadīt skaitli, kas lielāks par 0:

```
#!/usr/bin/python
print "Ievadiet skaitli, kas lielaks par 0:"
skaitlis=input()
while skaitlis<=0:
  skaitlis = input()
print "Ievadits skailis", skaitlis
```

Lietotāja dialogs ar programmu izskatīsies šādi:

```
Ievadiet skaitli, kas lielaks par 0:
-1
0
5
Ievadits skailis 5
```

Programmas sākumā lietotājam tiek piedāvāts ievadīt skaitli. Ja derīgs skaitlis (kurš neatbilst nosacījumam "mazāks vai vienāds par nulli", t.i., atbilst nosacījumam "lielāks par nulli") tiek ievadīts uzreiz, **while** cikls nesāks darbu – tiks izvadīts programmas rezultāts (pēdējā programmas rindiņā) un programma beigs darbu. Turpretī, ja tiks ievadīts nederīgs skaitlis, programma veiks **while** ciklā norādīto darbību (atkārtota skaitļa ievadīšana), kamēr netiks ievadīts korekts skaitlis.

Otrs cikla veids ir t.s. **for** cikls. Lai ilustrētu tā lietderīgumu, izveidosim programmu, kura izvada skaitļus no 1 līdz 20, izmantojot **while** ciklu:

```
#!/usr/bin/python
skaitlis=1
while skaitlis<=20:
  print skaitlis
  skaitlis = skaitlis+1
```

Šī programma izveido jaunu mainīgo ar nosaukumu skaitlis, piešķirot tam sākotnējo vērtību 1. Tad seko cikls, kurš atkārtosies, kamēr skaitlis nebūs lielāks par 20. Cikla operatori veic šī mainīgā izvadi uz ekrāna un palielina mainīgā vērtību par 1.

Šāds pieraksts nav pārskatāms – it īpaši, ja cikla iekšējo operatoru ir daudz. Tad var noderēt **for** cikls. Šī cikla shematiskais pieraksts ir šāds:

```
for parametrs in secība:
  cikla operatori
```

Šajā gadījumā **parametrs** ir mainīgais, kura vērtība tiks mainīta katrā cikla iterācijā,

un **secība** ir cikla darbības laikā izmantojamo vērtību virkne. To pieraksta formā

```
[vērtība1, vērtība2, ...]
```

Piemēram, iepriekš aplūkoto programmu iespējams pārrakstīt ar ciklu **for**:

```
#!/usr/bin/python
for skaitlis in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]:
  print skaitlis
```

Lai visu skaitļu virkni nebūtu jāpieraksta tik neērtā formā, iespējams izmantot **Python** iebūvēto funkciju **xrange**, kuras rezultāts ir skaitļu virkne norādītajā intervālā. Intervālu norāda līdzīgi, kā teksta rindām – izmantojot "atstarpes". Piemēram, pārveidojot mūsu programmu tā, lai for ciklā tiktu izmantota **xrange** funkcija, iegūstam rezultātu:

```
#!/usr/bin/python
for skaitlis in xrange(1,21):
  print skaitlis
```

Kā nedaudz sarežģītāku piemēru izveidosim programmu, kura izvadīs visus pāra skaitļus lietotāja norādītajā intervālā:

```
#!/usr/bin/python
print "Ievadiet sakumu"
sakums = input()
print "Ievadiet beigas"
beigas = input()
print "Para skaitli intervala no", sakums, "lidz", beigas, ":"
for skaitlis in xrange(sakums, beigas+1):
  if skaitlis%2==0:
    print skaitlis
print "Tie bija visi para skaitli saja intervala."
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Saraksti

Programmēšanas valodā **Python** ir vēl kāda iespēja informācijas saglabāšanai – **saraksti.** Saraksti ir secīgi (rindā) izvietoti **elementi** ar kopīgu nosaukumu. Elements var būt skaitliska vērtība vai teksta rinda. Sarakstus pieraksta formā

```
[elements1, elements2, ..., elementsn]
```

Piemēram, sarakstu ir iespējams izveidot šādi:

```
a = [1, 2]
b = [21, "junijs", 2005]
```

Tāpat ir iespējams apvienot sarakstus (līdzīgi, kā teksta rindas):

```
c = a+b
```

Uzdevums: kāds būs saraksta c saturs?

Līdzīgi, kā ar teksta rindām, no sarakstiem ir iespējams "izgriezt" elementus pēc to "atstarpju" numuriem:

```
#!/usr/bin/python
saraksts = [1,2,3,4,5]
print saraksts[3]
print saraksts[2:5]
print saraksts[4:]
```

Programmas darbības rezultāts būs

```
4
[3, 4, 5]
[5]
```

Aplūkosim citas darbības, kuras iespējams veikt ar sarakstiem:

```
#!/usr/bin/python
saraksts = [1,2,3,4,4,5]
print "Saraksts:", saraksts
# Elementu skaits
print "Elementu ar vertibu 4 skaits saraksta:", saraksts.count(4)
# Elementu pievienosana
saraksts.append(6)
print "Saraksts pec pievienosanas:", saraksts
# Elementu dzesana
# Tiek dzests pirmais elements ar vertibu 4
saraksts.remove(4)
print "Saraksts pec dzesanas:", saraksts
# Elementu aizvietosana
saraksts[0]=0
print "Saraksts pec aizvietosanas:", saraksts
```

Uzdevums: izskaidrojiet darbības principu katrai no veiktajām operācijām!

Papildus tam, izmantojot funkcijas no **Python** moduļa **string**, iespējams sadalīt teksta rindu sarakstā, kā arī sarakstu savienot vienā teksta rindā:

```
#!/usr/bin/python
import string
teksts = "1,2,3,4,5"
print teksts
# Sadalam teksta rindu. Ka elementu atdalitajs tiek izmantots komats.
saraksts = string.split(teksts, ",")
print saraksts
# Apvienojam sarakstu viena teksta rinda.
# Ka elementu atdalitaju izmantosim simbolu /
apvienotais=string.join(saraksts, "/")
print apvienotais
```

Programmas darbības rezultāts būs

```
1,2,3,4,5
['1', '2', '3', '4', '5']
1/2/3/4/5
```

Uzdevums: izskaidrojiet abu operāciju darbības principus!

Regulāras izteiksmes

Apstrādājot teksta rindas, bieži gadās, ka apstrādājamie dati ir doti kādā noteiktā formātā. Piemēram:

- Sastādīta plāna punkti sākas ar punkta numuru, pēc kā seko aizverošā iekava un punkta teksts (piemēram, ***1) aiziet uz veikalu***);
- Programmas teksts valodā **Python** arī ir sagatavots pēc stingriem gramatikas likumiem, piemēram, funkcijas definīcija sākas ar atslēgvārdu **def**;
- Datums latviešu valodā tiek pierakstīts formā ***diena.mēnesis.gads***, piemēram ***18.06.2005***.

Šajos piemēros norādīto informāciju iespējams sameklēt un apstrādāt, izmantojot konkrētā informācijas pieraksta "pazīmes". Lai šīs pazīmes pierakstītu, tiek izmantotas **regulārās izteiksmes**.

Regulāra izteiksme – tā ir formula, kuru iespējams izmantot, lai teksta rindā atrastu teksta fragmentu pēc noteikta šablona.

Funkcijas darbam ar regulārajām izteiksmēm ir iebūvētas **Python** modulī **re** (***regular expressions***).

Regulārajās izteiksmēs tiek izmantoti īpaši apzīmējumi, no kuriem populārākie norādīti tabulā:

Simbols	Nozīme
`.`	jebkurš viens simbols
`^`	rindas sākums
`$`	rindas beigas
`*`	iepriekš norādītā fragmenta atkārtošanās 0 vai vairāk reizes
`+`	tas pats, tikai vismaz 1 reizi
`?`	iepriekš norādītā fragmenta atkārtošānās 0 vai 1 reizi
`{m,n}`	iepriekš norādītā fragmenta atkārtošanās no **m** līdz **n** reizēm
`[...]`	jebkurš simbols no simbolu kopas kvadrātiekavās
`[^...]`	jebkurš simbols, kurš nepieder simbolu kopai kvadrātiekavās
`\`	ar šī simbola palīdzību iespējams atcelt nākamā simbola speciālo nozīmi. Piemēram, simbols **.** pats par sevi ir ar speciālu nozīmi (jebkurš viens simbols), bet simboli **\.** nozīmē **.** bez speciālas nozīmes
`|`	loģiskais **VAI** – simbols pa kreisi vai pa labi no šī simbola

Piemēri regulārām izteiksmēm:

Šablons	Piemēri teksta rindām, kas atbilst šablonam	Piemēri teksta rindām, kas neatbilst šablonam	Piezīmes
`.*`	"jebkurš teksts"		jebkuru reižu skaitu atkārtojas jebkurš simbols
`[abc]+`	"a", "abab", "abba", "abc"	"ce", "abcd"	rinda, kura nav tukša un sastāv no simboliem **a, b** vai **c**
`[a-z]+`	"aakkaa", "xyz", "python"	"piemers1", "2"	rinda, kura nav tukša un sastāv no mazajiem latīņu alfabēta burtiem
`^[0-9]`	"1rinda", "22"	"teksts1", "r"	rinda, kura sākas ar ciparu

Izveidosim programmu, kura aizstāj visus skaitļus rindā ar simbolu *:

```
#!/usr/bin/python
import re
teksts = "10, 23; 123 (127/2)"
apstradats = re.sub("[0-9]+", "*", teksts)
print apstradats
```

Programmas darbības rezultāts būs

```
*, *; * (*/*)
```

Kā redzam, funkcijas **re.sub** veiktā darbība ir līdzīga, kā **string.replace**, tikai viena fiksēta šablona vietā ļauj izmantot regulāru izteiksmi.

Uzdevums: izskaidrojiet programmas darbību!

Līdzīgi darbojas arī funkcija **re.search**:

```
#!/usr/bin/python
import re
teksts = "10b klase"
apstradats = re.search("^[0-9]+([a-d]).+$", teksts, re.MULTILINE)
print apstradats.group(1)
```

Šīs programmas rezultāts būs

```
b
```

t.i., klases indeksa burts. Meklējamā regulārā izteiksme norādīta iekavās (**[a-d]**); pārējā izteiksmes daļa ir meklēšanas **konteksts**, kurā atrodama meklējamā izteiksme. Šeit mums noderēja rindas sākuma un beigu norādīšanas simboli ^ un **$**. Meklēšanas rezultāts tiek ierakstīts mainīgajā **apstradats**. Ja norādītas vairākas meklējamās izteiksmes, arī rezultāti var būt vairāki. Tiem iespējams piekļūt pēc **indeksiem**:

```
apstradats.group(1)
apstradats.group(2)
```

Uzdevums: izdomājiet un izveidojiet piemēru programmai, kurā meklēšanai regulārā izteiksmē ir vairāki rezultāti!

Vārdnīcas

Sadzīvē bieži nākas saskarties ar jēdzienu **vārdnīca** – gan, lietojot adrešu grāmatiņu, kurā pēc drauga vai paziņas uzvārda un vārda varam atrast viņa adresi un tālruņa numuru, gan, meklējot informāciju enciklopēdijā, gan, protams, lietotojot visparastāko angļu-latviešu vārdnīcu.

Valodā **Python** vārdnīca nozīmē sasaisti starp **atslēgām** (***keys***) un **vērtībām** (***values***), pie kam katrai atslēgai ir viena un tikai viena unikāla vērtība. Vārdnīcas shematiskais pieraksts ir šāds:

```
{ atslēga1: vērtība1, atslēga2: vērtība2, ... }
```

Piemēram:

```
#!/usr/bin/python
vardnica = { "one": "viens", "two": "divi", "three": "tris" }
# Izdrukajam tulkojumu vardam "one"
print "One:", vardnica["one"]
# Pievienojam tulkojumu vardam "zero" un to izdrukajam
vardnica["zero"]="nulle"
print vardnica["zero"]
# Izdzesam tulkojumu vardam "three"
del vardnica["three"]
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Rezultātu formatēšana

Bieži var būt nepieciešams speciāli noformēt mainīgos, lai tos "skaisti" izvadītu uz ekrāna. To iespējams veikt šādi:

```
"formāta rinda" % (mainīgie)
```

Formāta rinda var saturēt šādus (bet ne tikai šādus) formatēšanas elementus:

%% - %

%i – vesels skaitlis

%s – teksta rinda

%f – daļskaitlis

\n – rindas pārnesums

Piemēram:

```
#!/usr/bin/python
skaitlis1=1
skaitlis2=2
teksts="%i+%i=%i\n" % (skaitlis1, skaitlis2, skaitlis1+skaitlis2)
print teksts
```

Uzdevums: kāds būs šīs programmas darbības rezultāts?

Darbs ar failiem

Līdz šim mēs nevarējām izveidot neko nopietnu tādēļ, ka mūsu programmas nespēja strādāt ar **failiem**. Fails – tā ir **baitu (datu) virkne**, kura tiek uzglabāta ārējā atmiņā (piemēram, cietajā diskā vai kompaktdiskā) un pie kuras iespējams piekļūt pēc tās nosaukuma. Datus no faila iespējams nolasīt, kā arī tajā ierakstīt, tomēr pirms tā failu **jāatver** (**open**) un pēc darba beigām **jāaizver** (**close**).

Vienkāršs piemērs programmai, kura ieraksta informāciju failā:

```
#!/usr/bin/python
fails = open("tests.txt", "wt")
for klase in xrange(1,13):
  fails.write("%i.klase\n" % (klase))
fails.close
```

Uzdevums: kāds būs faila "tests.txt" saturs? Izskaidrojiet programmas darbību!

Ierakstītos datus ir iespējams arī nolasīt no faila:

```
#!/usr/bin/python
fails = open("tests.txt", "rt")
rinda = fails.readline()
while rinda!="":
  print rinda
  rinda = fails.readline()
fails.close
```

Uzdevums: izskaidrojiet programmas darbību!

Atverot failu, funkcijai **open** jānorāda divi argumenti: **faila nosaukums** un faila **atvēršanas režīms**. Praksē darbam ar teksta failiem visbiežāk tiek izmantoti divi režīmi – **rt** (**read text** – teksta lasīšanas) un **wt** (**write text** – teksta rakstīšanas).

Šajā grāmatā sniegtā informācija ir minimālais nepieciešamais, lai varētu sākt apgūt programmēšanas valodu **Python**. Papildinformāciju par **Python** iespējams iegūt Internet tīklā – http://www.python.org. Sarežģītāki **Python** aspekti – kļūdu apstrāde, objektorientētā programmēšana u.c. – tiks aplūkoti nākamajā grāmatā.