Programmēšanas valoda

iesācējiem

2. daļa. *Ciklu darbības kontrole. Moduļu izstrāde. Kļūdu situāciju apstrāde. Objektorientētā programmēšana.*

Saturs

Ievads....3
Ciklu darbības kontrole....4
Operators, kurš neko nedara: pass....7
Vairākmoduļu programmu izstrāde....9
Kļūdu situāciju apstrāde....11
Objektorientētā programmēšana....13
Python standartmoduļu bibliotēka....19
Modulis os - piekļuve operētājsistēmas resursiem....19
Modulis sys - specifiska sistēmas funkcionalitāte....20
Modulis re - regulāras izteiksmes....20
Modulis math - matemātiskās funkcijas....20
Modulis random - gadījuma skaitļu ģenerators....21
Modulis string - darbs ar teksta rindām....21
Nobeigumam....24

Ievads

Pirmajā grāmatas "*Programmēšanas valoda Python iesācējiem*" daļā aplūkojām *Python* programmēšanas pašus pamatus. Lai gan ar to pietiek, lai veidotu programmas vienkāršu uzdevumu risināšanai, tomēr dažādu *Python* papildiespēju izmantošanai jāapgūst vēl vairākas *Python* iespējas.

Šī grāmata ir otrā daļa sērijā, kuras uzdevums ir papildināt nelielo latviešu valodā pieejamo programmēšanas mācību materiālu klāstu, kā arī popularizēt programmēšanas valodu *Python*. Šajā daļā aplūkosim tieši tās *Python* iespējas, kuras nav ļoti nepieciešamas elementāru programmu izveidošanai, bet kuru izpratne ļaus viegli apgūt daudzas citas zināšanas - grafiskā interfeisa, datu bāžu, multimēdiju un, galu galā - sarežģītu un iespējām bagātu programmu izstrādi.

Grāmatā aplūkotas tādas *Python* iespējas, kā ciklu darbības kontrole, moduļu izstrāde, kļūdu situāciju apstrāde un objektorientētā programmēšana.

Ir plāni nākotnē papildināt šo sēriju ar nākamajām daļām, kurās tiktu aplūkota grafiskā interfeisa izstrāde - gan ar *Python* standartkomplektācijā iekļauto *Tkinter* moduli, gan ar daudz modernāko un jaudīgāko *wxPython*. Tiktu aplūkoti arī multimēdiju programmu (t.sk. grafisko datorspēļu) izstrādes pamatprincipi, kā arī uzņēmumos izmantojamu informācijas sistēmu izveide.

Ciklu darbības kontrole

Grāmatas "*Programmēšanas valoda Python iesācējiem*" 1.daļā aplūkojām ciklu izstrādes pamatprincipus. Ja atceraties, valodā *Python* eksistē divu veidu cikli - `while` un `for` cikli. `while` ciklā iekļautās darbības tiek izpildītas, kamēr izpildās t.s. while nosacījums, bet `for` izpilda darbības noteiktu reižu skaitu, to uzskaitei izmantojot īpašu skaitītāju.

Tipisks piemērs `while` ciklam izskatās šādi:

```
#!/usr/bin/python
# Ievadam para skaitli
A = input()
while (A%2)!=0:
    print "Tas nav para skaitlis!"
    A = input()
print "Ievadits para skaitlis", A
```

`for` ciklu, savukārt, varam pierakstīt šādi:

```
#!/usr/bin/python
# Izdrukajam visus skaitlus no 1 lidz 20
for A in xrange(1, 21):
    print A
```

Kamēr cikls ir triviāls un tā prognozējamais izpildes laiks nav liels, viss, šķiet, ir kārtībā. Tomēr var gadīties situācijas, kurās cikla darbību vēlams pārtraukt, tiklīdz izpildās (vai neizpildās) kāds nosacījums. Piemēram, izveidosim programmu, kura ļauj ievadīt riņķa līniju diametrus un aprēķina šo riņķa līniju garumu summu. Ja tiek ievadīts garums "0", programma izvada aprēķināto summu un beidz darbu. Algoritms varētu būt šāds:

```
1. Atkārtojam darbību:
1.1. Ļaujam lietotājam ievadīt riņķa līnijas diametru.
1.2. Ja ievadītais diametrs ir mazāks vai vienāds ar nulli, pārtraucam darbu, izvadot līdz šim aprēķinātos garumus
1.3. Aprēķinām riņķa līnijas garumu.
1.4. Pieskaitām aprēķināto riņķa līnijas garumu kopējai summai.
2. Izvadām rezultātu.
```

Ar tām *Python* zināšanām, kādas mums ir šobrīd, šādu algoritmu pierakstīt *Python* valodā mēs varam, piemēram, šādi:

```
#!/usr/bin/python
# Mums vajadzes funkciju pi no modula math
import math
# Sakotneja summa ir 0
Summa = 0
# Laujam ievadit pirmo diametru
Diam = input()
# Ievades cikls
while Diam != 0:
    # Aprekinam rinka linijas garumu
    Garums = Diam * math.pi
    # Pieskaitam summai
    Summa = Summa + Garums
    # Laujam ievadit nakoso diametru
    Diam = input()
print Summa
```

"It kā" viss izskatās jauki, tikai... Jaukāk būtu, ja mums nebūtu jāsarežģī programmu

ar liekiem `input()`... To ir iespējams atrisināt, izmantojot *Python* valodā pieejamo ciklu vadības operatoru `break`. `break` izpilde pārtrauc cikla darbību un pāriet uz programmas tālāku izpildi no vietas, kura seko tūlīt pēc cikla. Lai sekmīgi izmantotu `break` operatoru, atcerēsimies, ka `while` cikla izpildes nosacījums ir loģiska izteiksme un mēs viegli varam pierakstīt loģisku izteiksmi, kuras rezultāts vienmēr būs "patiess":

```
1==1
```

Patiešām, 1 taču vienmēr ir vienāds ar 1! Ja mums nepieciešama loģiska izteiksme, kuras rezultāts ir "aplams", varam to pierakstīt kā

```
1==2
```

vai

```
1>2
```

Šādas loģiskās izteiksmes ir iespējams pierakstīt kā jebkuru matemātisku izteiksmi, kuras rezultāts būs vienmēr "patiess" vai vienmēr "aplams".

Tagad pamēģināsim pārrakstīt mūsu programmu tā, lai izmantotu operatoru `break`:

```
#!/usr/bin/python
# Mums vajadzes funkciju pi no modula math
import math
# Sakotneja summa ir 0
Summa = 0
# Ievades cikls.
while 1==1:
    # Laujam ievadit diametra vertibu
    Diam = input()
    # Ja diametrs ir 0, partraucam darbu
    if Diam==0:
          break
    # Aprekinam rinka linijas garumu
    Garums = Diam * math.pi
    # Pieskaitam to summai
    Summa = Summa + Garums
# Izvadam summu
print Summa
```

Redzam, ka operatora `break` izmantošana ir padarījusi mūsu programmu loģiskāku un pārskatāmāku.

Otrs ciklu kontroles operators ir `continue`. Tas pārtrauc cikla tekošās **iterācijas** (atkārtojuma) izpildi un pāriet pie nākamās iterācijas izpildes. Piemēram, izveidosim programmu, kura izvadīs visus skaitļus no 1 līdz 20, izņemot 11:

```
#!/usr/bin/python
for A in xrange(1, 21):
    if A!=11:
          print A
```

Šajā gadījumā programmas loģika nosaka, ka skaitlis tiek izvadīts, ja tas neatbilst nosacījumam, pie kura skaitli nedrīkst izvadīt... Sanāk tāda dīvaina loģika - daram kaut ko, ja tas **nav neatļauts**. Izmantojot operatoru `continue`, mēs varam programmas loģiku organizēt pēc iespējas tuvāk uzdevumam - **ja nav atļauts, nedarām**:

```
#!/usr/bin/python
for A in xrange(1, 21):
    if A==11:
          continue
    print A
```

Uzdevums: izskaidrojiet, ar ko atšķiras šī programma no iepriekšējās!

Izmantojot operatorus `break` un `continue`, bieži ir iespējams būtiski vienkāršot sarežģītu ciklu loģiku un izvairīties no lieku darbību izpildes.

Operators, kurš neko nedara: *pass*

Veidojot sarežģītākas programmas, to funkcionalitāte bieži vien tiek sadalīta loģiskās vienībās - **funkcijās**. Kā atceramies no šīs grāmatas pirmās daļas, funkciju definē šādi:

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

Kad funkcija izveidota, to iespējams izsaukt jeb izmantot citās programmas daļās, izmantojot tās nosaukumu. Diemžēl bieži gadās situācijas, kad izdevīgāk ir vispirms izveidot funkcijas izsaukumu, un tikai tad programmēt pašu funkciju. Ja funkcija ir sarežģīta, tas ļauj ērtāk atrast iespējamās kļūdas funkcijas definīcijā un pārbaudīt funkcijas darbības rezultātus. Tādās situācijās programmēšanu var nosacīti sadalīt divās fāzēs - **prototipa izveide** un **funkcionalitātes realizācija**. Piemēram, funkciju, kura aprēķinās riņķa līnijas garumu, ja dots tās diametrs, mēs varam prototipēt šādi:

```
#!/usr/bin/python
# Mums vajag math.pi...
import math
# Funkcija RinkaGarums
def RinkaGarums(Diam):
    return 0
# Izsaucosa programma
print "Ievadiet diametru"
D = input()
G = RinkaGarums(Diam)
print "Garums ir", G
```

Kad esam pārliecinājušies, ka programma korekti izsauc funkciju `RinkaGarums`, kuras rezultāts pagaidām ir nulle, varam korekti izveidot arī pašu funkciju:

```
def RinkaGarums(Diam):
    Garums = Diam * math.pi
    return Garums
```

Problēmas rodas, kad tiek veidotas tā saucamās **bezrezultāta** funkcijas, kuras citās programmēšanas valodās (piemēram, *Pascal*) sauc par **procedūrām**. Šīs funkcijas var būt ar vai bez argumentiem, tomēr tām nav rezultāta. Jautāsiet - kam tas ir vajadzīgs? Atbilde ir ļoti vienkārša - bieži vien programmās ir jāveic dažādas **rutīnas** darbības, kurām nav nepieciešams rezultāts. Piemēram, tā var būt teksta izvade:

```
#!/usr/bin/python
def DrukaSummu(A,B):
    print A,"+",B,"=",A+B
A = input()
B = input()
DrukaSummu(A,B)
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Kā redzam, šajā funkcijā nav operatora `return`. Rodas jautājums - kā prototipēt šādu funkciju? Pamēģināsim:

```
#!/usr/bin/python
def DrukaSummu(A,B):
A = input()
B = input()
DrukaSummu(A,B)
```

Mēģinot izpildot šo programmu, tā vietā, lai to godīgi veiktu, *Python* mūs godīgi salamās:

```
There's an error in your program.
Expected an indented block.
```

Problēma rodas tur, ka *Python* valodas sintakse pieprasa, lai jebkurā programmas elementā, kurš tiek pierakstīts ar atkāpi (funkcija, cikls, pārbaude u.c.) būtu vismaz viens operators. Tādiem gadījumiem, kad nav iespējams izmantot nevienu citu operatoru, ir paredzēts operators `pass`. Tas nedara neko "derīgu" un turpina programmas izpildi no nākamās rindiņas. Tomēr tas ir tieši tas, kas mums nepieciešams:

```
#!/usr/bin/python
def DrukaSummu(A,B):
    pass
A = input()
B = input()
DrukaSummu(A,B)
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Vairākmoduļu programmu izstrāde

Izstrādājot apjomīgas programmas, to pieraksts ātri vien kļūst tik garš, ka izsekot tam līdz un, vēl jo vairāk, atrast un novērst iespējamās kļūdas vai papildināt programmas funkcionalitāti kļūst ļoti grūti - pat neiespējami. Tādos gadījumos ir jādomā par to, kā programmu sadalīt vairākos neatkarīgos blokos. Mēs protam veidot funkcijas - gan tādas, kuras atgriež rezultātu, gan bezrezultāta. Iekļaujot tajās dažādu funkcionalitāti, kuru mēs savā programmā bieži izmantojam, ir iespējams būtiski vienkāršot programmas izskatu. Diemžēl programmas teksta garumu tas praktiski neietekmē, lai arī padara to pārskatāmāku. Un, galu galā, ja mēs esam izveidojuši kādu funkciju, mūs principā vairs neinteresē tās teksts, bet gan tas, kādi ir tās argumenti un, ja ir, tad kāds ir rezultāts. Bez tam, ja mēs esam izveidojuši kādu funkciju vai funkcijas, kuras var mums būt noderīgas arī citu programmu izstrādei, diez vai prātīgākais būtu katru reizi šīs funkcijas meklēt un dublēt jaunās programmas tekstā.

Lai risinātu šīs problēmas, programmēšanas valoda *Python* piedāvā programmētājam iespēju sadalīt programmas tekstu vairākos failos. Katru šādu failu sauc par **moduli**. *Python* standartkomplektācijā jau ir iekļauti daudzi moduļi - piemēram, mums jau zināmie `math`, `string`, `re` un daudzi citi. Līdzīgi mēs varam izstrādāt paši savus moduļus.

Pirms moduļa izstrādes jānolemj, kā mēs to sauksim. Moduļa nosaukumā vēlams izmantot latīņu alfabēta burtus un ciparus un to vēlams izvēlēties tā, lai tas raksturotu modulī ietverto funkciju būtību. Piemēram, izveidosim moduli, kurā ietversim divas funkcijas - vienu, kas aprēķina riņķa līnijas garumu, ja zināms tās rādiuss, un otru - kura aprēķina riņķa laukumu. Loģiskākais šāda moduļa nosaukums ir `rinkis`. No šī nosaukuma arī jāizveido faila nosaukumu, pievienojot tam beigās simbolus `.py` - līdzīgi, kā jebkurai *Python* programmai. Tas nozīmē, ka faila nosaukums modulim `rinkis` ir `rinkis.py`. Jāņem vērā, ka, veidojot "savus" moduļus, to failiem jāatrodas vienā direktorijā ar to izsaucošās jeb **importējošās** programmas failu.

Moduļa teksts tiek veidots līdzīgi, kā jebkuras *Python* programmas teksts, vienīgi nav vajadzības pirmajā rindiņā ierakstīt

```
#!/usr/bin/python
```

Piemēram, moduļa `rinkis` teksts izskatīsies šādi:

```
import math
def Garums(R)
    Rezultats = 2 * math.pi * R
    return Rezultats
def Laukums(R)
    Rezultats = math.pi * R * R
    return Rezultats
```

Uzdevums: izskaidrojiet modulī ietverto funkciju darbību!

Līdzīgi, kā izmantojot standartmoduļus, arī pirms mūsu jaunizveidotajā modulī ietverto funkciju izmantošanas mums programmai ir jānorāda, ka šo moduli jāielādē,

izmantojot operatoru `import`:

```
#!/usr/bin/python
import rinkis
print "Ievadiet radiusu"
R = input()
print "Garums ir", rinkis.Garums(R)
print "Laukums ir", rinkis.Laukums(R)
```

Tieši tāpat ir iespējams izmantot otru moduļu ielādes metodi:

```
from rinkis import *
```

Protams, jāņem vērā, ka tādā gadījumā mainīsies funkciju pieraksts (sk. grāmatas pirmo daļu).

Uzdevums: pārrakstiet šo programmu tā, lai tā izmantotu otru moduļu ielādes metodi!

Veidojot moduļus, tie var samērā brīvi izmantot viens otru, kā arī *Python* standartkomplektācijā iekļautos moduļus. Šāda programmēšanas pieeja ļauj būtiski vienkāršot sarežģītu programmu tekstus un padara tos pārskatāmākus. Bez tam, nekas mūs netraucē jau izveidotu moduli ielādēt citā programmā, lai izmantotu to tieši tāpat, kā iepriekš veidotajā.

Kļūdu situāciju apstrāde

Katrs, kam kādreiz ir gadījies dzirdēt *Latvijas radio* 1.programmu, būs saklausījis arī raidījumu "Mūsu valoda", kurā tante aizsmakuši reverberētā balsī bļauj: "Kļūda, kļūda, kļūda!" Diemžēl arī programmu izstrādes procesā gadās kļūdas. Mazākais, ja tā ir sintakses kļūda, t.i., nekorekti pierakstīta izteiksme vai operators. Tādā gadījumā *Python* mūs, kā nākas, izlamās un ļaus kļūdu izlabot. Ir vēl arī otra kļūdu klase - loģiskās kļūdas. Tās parasti parādās dažādās situācijās, kuras mēs vismazāk esam gaidījuši. Svarīgi ir nodrošināt, lai ar tām pēc iespējas nebūtu jāsaskaras gala lietotājam. Piemēram:

```
#!/usr/bin/python
# Laujam ievadit divus skaitlus un izvadam to dalijumu
A = input()
B = input()
print A/B
```

Pamēģināsim ievadīt divus skaitļus: `2` un `0`... Kā jūs domājat, kāds ir programmas izpildes rezultāts?

Tā kā dalīt ar nulli nedrīkst, varam programmu pierakstīt šādi:

```
#!/usr/bin/python
A = input()
B = input()
if B==0:
    print "Dalisana ar nulli!"
else:
    print A/B
```

Uzdevums: izskaidrojiet šīs programmas darbību!

Veicot aprēķinus daudz sarežģītākām izteiksmēm, šāda pieļaujamo vērtību pārbaude var būt visnotaļ apgrūtinoša un sarežģīta. Piemēram, ja jāaprēķina izteiksmes `A/B/C` vērtību, jau ir jāpārbauda gan `B`, gan `C`. Un tas nebūt nav sarežģītākais gadījums...

Uzdevums: izveidojiet programmu, kura aprēķina izteiksmes `A/B/C` vērtību, korekti pārbaudot ievaddatus!

Lai vienkāršotu kļūdu situāciju apstrādi, programmēšanas valoda *Python* piedāvā iespēju veikt kļūdu apstrādi un "ķeršanu". To *Python* pieraksta šādi:

```
try:
    operatori, kurus mēģina izpildīt
except:
    operatori, kurus izpilda kļūdas gadījumā
```

Ja, izpildot operatorus, kuri ierakstīti `try` blokā, kļūdas nerodas, tad `except` blokā ierakstītie operatori tiek ignorēti. Savukārt, ja `try` blokā rodas kļūda, tad šī bloka izpilde tiek pārtraukta un tiek izpildīti operatori, kas ierakstīti `except` blokā. Tiesa, šāds `try/except` apraksts ir nedaudz nepilnīgs un neaplūko `except` bloka papildiespējas kļūdu identificēšanā, tomēr trūkstošo informāciju iespējams atrast *Python* projekta mājaslapā `http://www.python.org`.

Lai ilustrētu `try/except` izmantošanu, pārveidosim mūsu iepriekš izveidoto

programmu tā, lai kļūdu pārbaude notiktu ar šīs konstrukcijas palīdzību.

```
#!/usr/bin/python
A = input()
B = input()
# Meginam dalit
try:
    print A/B
# Kluda!
except:
    print "Dalisana ar nulli!"
```

`try/except` ir iespējams izmantot visur, kur potenciāli eksistē kļūdas situācijas iespēja. `try` blokā jāievieto "aizdomīgos" operatorus un kļūdas situācijas apstrādi jāveic `except` blokā.

Objektorientētā programmēšana

Mūsdienās bieži dzirdētais termins "objektorientētā" programmēšana daudziem var izraisīt neizpratni vai tieši otrādi - var sadzirdēt atvieglotu uzelpu - beidzot! Jā, arī *Python* piedāvā programmētājam plašas objektorientētās programmēšanas iespējas. Objektorientētā programmēšana valodā *Python* ir realizēta, minimāli paplašinot esošo valodas sintaksi, tādēļ to apgūt nevajadzētu sagādāt sevišķas problēmas. Protams, daudziem rodas jautājums: kas tā ir - objektorientētā programmēšana?

"Parastajā" jeb **procedurālajā** programmēšanā programma darbojas, pēc kārtas izpildot operatorus. Ir iespējams izsaukt funkcijas, kurām iespējams nodot argumentus un, iespējams, saņemt rezultātus. Tomēr programmēšanas centrā paliek algoritms, nevis apstrādājamie dati. Objektorientētā programmēšana ļauj apstrādājamos datus apvienot ar to apstrādes algoritmiem, izveidojot objektu klases. Pēc tam programmā ir iespējams izveidot objektus, kuri katrs pieder pie noteiktas klases. Objektu klases arī var būt savā starpā saistītas - piemēram, viena klase var no citas pārmantot abām raksturīgās īpašības. Objektorientētā programmēšana, to pareizi pielietojot, var būtiski atvieglot lielu programmu izstrādi. To visu mēģināšu izskaidrot šajā nodaļā.

Tātad, kas ir **objekti** un to **klases**? Iedomāsimies situāciju no reālas dzīves - kas ir lopkopības saimniecības raksturīgākie objekti? Laikam jau lopi... Noteiksim, ka "Lops" ir **klase**, pie kuras piederēs visi mūsu virtuālās saimniecības lopi. Tiesa, no tik vispārīgas klases lielas jēgas nav, jo lopi var būt dažādi - govis, zirgi, cūkas, aitas u.c. Tiem ir kopīgas īpašības - piemēram, tie visi ēd. Tiem ir arī īpašības, kuru realizācija var atšķirties - piemēram, visi lopi prot izdvest skaņas, tomēr katrai lopu apakšklasei tās būs citas - govis mauj, zirgi zviedz, cūkas rukšķ, aitas blēj u.c. Un, protams, ir arī īpašības, kuras katrai klasei būs pilnīgi atšķirīgas - piemēram, govis badās, bet nespārdās (pieņemsim, ka tā), zirgi toties nebadās (jo nav ar ko), toties mēdz spārdīties u.c. Šos "atklājumus" mēģināsim pierakstīt strukturēti:

1. Klase "Lops":
   - ēd
   - izdod skaņas (tiesa, vispārīgā gadījumā nav zināms, kādas)
2. Apakšklase "Govs":
   - ēd (kā jau visi lopi)
   - izdod skaņas (precīzi - mauj)
   - badās
3. Apakšklase "Zirgs":
   - ēd
   - izdod skaņas (zviedz)
   - spārdās
4. Apakšklase "Cūka":
   - ēd
   - izdod skaņas (rukšķ)

5. Apakšklase "Aita":

- ēd
- izdod skaņas (blēj)
- mēdz dumji skatīties

Katrai klasei, izņemot klasi "Lops" (jo tā ir pārāk nekonkrēta), mēs varam noteikt jau konkrētus objektus, piemēram "Govs Raibaļa", "Zirgs Melnītis", "Cūka Aigars", "Aita Mikumeku" u.c. Ja vēlamies, varam noteikt, ka mums ir vairāki objekti, kas pieder pie vienas klases, piemēram, papildus mums var būt arī "Govs Brūnaļa". Redzam, ka katram šim objektam ir arī īpašība - vārds.

Bet nu pietiks filozofēt par abstraktām klasēm - mūsu mērķis ir iemācīties tās pierakstīt programmēšanas valodā *Python*. Pamatsintakse klases definīcijai izskatās šādi:

```
class Nosaukums:
    klases metodes
```

Hmm, bet kas ir metodes? Dažādo īpašību realizācijas *Python* tiek sauktas par **metodēm**. Tās tiek definētas kā vienkāršas funkcijas. Īpaša metode ir `__init__`. Tā tiek saukta par **konstruktoru** un tiek automātiski izpildīta, kad tiek radīts jauns klases objekts. Konstruktors var veikt objektā saglabājamo datu inicializāciju. Lai labāk izprastu klašu definēšanu, aplūkosim piemēru:

```
class Lops:
    def __init__(self, Vards):
          self.Vards = Vards
    def Est(self):
          print "Njam njam"
    def Skana(self):
          pass
```

Kas ir `self`? Tas ir mainīgais, kurš jau konkrēta objekta gadījumā nozīmēs "es pats". Pagaidām par to varam aizmirst, jo to mēs aplūkosim vēlāk, bet jāatceras to, ka, izsaucot objekta metodi, `self` kā argumentu ir jāizlaiž. Piemēram, klases "Lops" metodi `Skana(self)` būtu jāizsauc kā

```
Lops.Skana()
```

nevis

```
Lops.Skana(self)
```

Ņemot vērā to, ka mēs iepriekš vienojāmies, ka klase "Lops" mums ir vispārīga jeb abstrakta klase, tad definēsim arī jau konkrētas klases, kuras "mantos" vairākas klases "Lops" īpašības:

```
class Govs(Lops):
    def Skana(self):
          print "Muuu"
    def Badities(self):
          print "Buc"
class Zirgs(Lops):
```

```
    def Skana(self):
          print "Iii-aaa"
    def Spardities(self):
          print "Bam"
class Cuka(Lops):
    def Skana(self):
          print "Rukruk"
class Aita(Lops):
    def Skana(self):
          print "Mekmek"
```

Definējot apakšklases kādai citai klasei, jāņem vērā, ka:

1) Ja metode nav definēta virsklasē (raugoties no apakšklases viedokļa), tad tā eksistēs tikai konkrētajā apakšklasē;

2) Ja metode ir definēta virsklasē, bet tiek definēta arī apakšklasē, tad konkrētajā situācijā tās izpilde atšķirsies no atbilstošās virsklases metodes;

3) Ja metode ir definēta tikai virsklasē, tad tā identiski darbosies arī apakšklasē.

Jūs jautāsiet - bet kāda jēga no visa šī ārprāta? Jēga ir tāda, ka tagad mēs varam definēt veselu lopu saimniecību, tādējādi demonstrējot objektorientētās programmēšanas būtību.

```
Lops1 = Govs("Raibala")
Lops2 = Govs("Brunala")
Lops3 = Zirgs("Beritis")
Lops4 = Cuka("Aigars")
Lops5 = Aita("Mekameka")
Lops3.Est()
Lops1.Skana()
Lops2.Skana()
Lops3.Skana()
Lops4.Skana()
Lops5.Skana()
Lops2.Badities()
Lops3.Spardities()
```

Apvienosim visu iepriekš uzrakstīto vienā programmā:

```
#!/usr/bin/python
class Lops:
    def __init__(self, Vards):
          self.Vards = Vards
    def Est(self):
          print "Njam njam"
    def Skana(self):
          pass
class Govs(Lops):
    def Skana(self):
          print "Muuu"
    def Badities(self):
          print "Buc"
class Zirgs(Lops):
    def Skana(self):
          print "Iii-aaa"
    def Spardities(self):
          print "Bam"
class Cuka(Lops):
    def Skana(self):
          print "Rukruk"
class Aita(Lops):
    def Skana(self):
          print "Mekmek"
Lops1 = Govs("Raibala")
Lops2 = Govs("Brunala")
Lops3 = Zirgs("Beritis")
Lops4 = Cuka("Aigars")
Lops5 = Aita("Mekameka")
Lops3.Est()
Lops1.Skana()
Lops2.Skana()
Lops3.Skana()
Lops4.Skana()
Lops5.Skana()
Lops2.Badities()
Lops3.Spardities()
print Lops1.Vards
print Lops2.Vards
print Lops3.Vards
print Lops4.Vards
print Lops5.Vards
print "Un tagad interesantakais..."
Lops = Lops1
Lops.Skana()
Lops = Lops5
Lops.Skana()
```

Lai labāk izprastu objektorientētās programmēšanas principus, aplūkosim šo programmu tuvāk.

Vispirms tiek definēta klase "Lops". Tai tiek definēts konstruktors, kurš klases "iekšpusē" izveidos jaunu **mainīgo** jeb **īpašību** `Vards`, kuram tiks piešķirts atbilstošs konstruktora arguments. Tā kā metode `Est` būs kopīga visām klases "Lops" apakšklasēm, tad arī to definējam šajā klasē, lai izvairītos no liekas programmēšanas vēlāk. Metode `Skana` eksistēs visās klases "Lops" apakšklasēs, tomēr katrā apakšklasē tās veiktā darbība būs cita, tādēļ pašu metodi ir jādefinē, bet pagaidām atstāsim to "tukšu", izmantojot iepriekšējā sadaļā apgūto operatoru `pass`.

Lai definētu klasi, kura ir apakšklase citai klasei, klases definīciju jāpieraksta šādi:

```
class Apaksklase(Virsklase):
...
```

Tā piemēram, definējot klasi "Govs", kura būs apakšklase klasei "Lops", pieraksts izskatās

```
class Govs(Lops):
...
```

Klasei "Govs" jau ir skaidrs, kādu darbību jāveic metodei `Skana`, tādēļ programmā tā ir pārdefinēta. Klasei "Govs" ir arī viena papildus metode `Badities`, kura arī tiek definēta.

Līdzīgi tiek definētas arī klases "Lops" apakšklases "Zirgs", "Cuka" un "Aita".

Pašā programmā tiek definēti pieci objekti, kas pieder pie dažādām klasēm. Tā kā visas šīs klases ir apakšklases klasei "Lops", tad jāņem vērā klases "Lops" konstruktoram nododamais arguments `Vards`. Līdz ar to objekts tiek radīts šādi:

```
Lops1 = Govs("Raibala")
```

Objekta metodes tiek izsauktas šādi:

```
Objekta_nosaukums.Metode(argumenti)
```

Piemēram:

```
Govs.Est()
```

Viena no interesantākajām objektorientētās programmēšanas iespējām ir izveidot atsauces uz konkrētu objektu. Tā piemēram:

```
Lops = Lops1
```

Tādā gadījumā `Lops` būs objekts, kurš būs identisks objektam `Lops1`, t.i., tas piederēs pie tās pašas klases un būs iespējams izmantot visas šīs klases objektu metodes un īpašības.

Uzdevums: kāds būs šīs programmas darbības rezultāts? Izskaidrojiet šīs programmas darbību!

Līdz ar to, pateicoties šādai objektorientētās programmēšanas pieejai, ir iespējams efektīgi papildināt jau iepriekš izveidotu klašu funkcionalitāti, saglabājot jaunizveidoto klašu kopīgo funkcionalitāti un nodrošinot atšķirīgās funkcionalitātes ieviešanu. Kur tas praktiski noder programmēšanā? Iedomāsimies, ka mēs veidojam klašu bibliotēku dažādu grafisko primitīvu (līniju, apļu, punktu u.c.) "zīmēšanai". Visiem šiem objektiem (patiesībā to klasēm) būs kopīgas īpašības - piemēram, krāsa un sākumkoordinātes. Tomēr būs arī atšķirīgas īpašības - līnijai tās būs beigu koordinātes, aplim tas būs rādiuss, punktam patiesībā papildīpašību var arī nebūt. Atšķirsies arī katra grafiskā primitīva zīmēšana. Tādā gadījumā mēs varam izveidot **virsklasi** `GrafiskaisPrimitivs` un realizēt tās apakšklases `Linija`, `Aplis` un `Punkts`. Dažādi šo apakšklašu objekti var piederēt pie dažādām klasēm, tomēr mūs tas maz interesēs - piemēram, metodei, kura "uzzīmēs" šādu primitīvu, argumenti būs vienādi un līdz ar to būs iespējams pierakstīt:

```
Objekts = ...(...)
...
Objekts.Zimet()
...
```

Pie tam nebūs svarīgi, pie kuras konkrētās klases pieder objekts.

Uzdevums: mēģiniet patstāvīgi izveidot klasi *Aritmetika* ar šādām īpašībām un metodēm:
1) klases īpašības ir divi skaitļi - *X* un *Y*
2) klases konstruktors veic šo skaitļu inicializāciju ar vērtībām, kuras tiek "nodotas" kā argumenti
3) klasē pieejamas metodes:
- ***UzstaditX(X)* - uzstāda X vērtību**
- ***UzstaditY(Y)* - uzstāda Y vērtību**
- ***Summa()* - metodes rezultāts ir X un Y summa**
- ***Starpiba()* - metodes rezultāts ir X un Y starpība**

Uzdevums: izveidojiet klases *Aritmetika* apakšklasi *Aritmetika2*, kura papildina klasi *Aritmetika* ar šādām jaunām metodēm:
1) *Reizinajums()* - metodes rezultāts ir *X* un *Y* reizinājums
2) *Dalijums()* - metodes rezultāts ir *X* un *Y* dalījums. NB! Jāņem vērā, ka dalīt ar 0 nedrīkst!

Python standartmoduļu bibliotēka

Abās šīs grāmatas daļās mēs saskārāmies ar *Python* standartmoduļu bibliotēku. Šī bibliotēka satur moduļus, kuri papildina *Python* funkcionalitāti, nodrošinot visbiežāk veicamo darbību vienkāršošanu. Šādi moduļi ir, piemēram, `math`, kurš papildina *Python* ar dažādām matemātiskajām funkcijām, `string`, kurš satur funkcionalitāti teksta rindu apstrādei un daudzi citi. Šīs nodaļas uzdevums ir virspusēji aplūkot standartbibliotēkā iekļautos moduļus un to funkcionalitāti. Netiek aplūkoti reti izmantojami moduļi un arī aplūkojamo moduļu reti izmantojamā funkcionalitāte. Plašāku informāciju vienmēr ir iespējams atrast *Python* projekta mājaslapā `http://www.python.org`.

Modulis `os` *- piekļuve operētājsistēmas resursiem*

Modulis `os` ļauj piekļūt datora operētājsistēmas resursiem. Šo moduli jāielādē ar operatoru

```
import os
```

nevis

```
from os import *
```

Tas tādēļ, ka modulī `os` ir iekļauta funkcija `open()`, kura "pārklāj" *Python* iebūvēto funkciju `open()`. Līdz ar to pie moduļa `os` funkcijām jāpiekļūst šādi:

```
os.funkcija()
```

Modulī `os` iekļautas šādas funkcijas (kā jau rakstīts, aplūkotas tiek tikai visbiežāk izmantotās):

```
environ[mainigais]
```

funkcijas rezultāts ir operētājsistēmas vides mainīgais. Piemēram,

```
environ['PATH']
```

rezultāts ir operētājsistēmas vides mainīgā PATH saturs

```
chdir(cels)
```

nomaina tekošo direktoriju jeb mapi. Piemēram,

```
chdir("/usr/bin")
```

vai

```
chdir("C:\Windows")
```

```
getcwd()
```

funkcijas rezultāts ir tekošā direktorija jeb mape. Piemēram,

```
Mape = getcwd()
print Mape
```

Uzdevums: kāds būs šo divu programmas rindiņu izpildes rezultāts?

```
system(komanda)
```

izpilda ārējo operētājsistēmas komandu, piemēram, kādu programmu:

```
system("dir")
```

vai

```
system("ls")
```

Modulis `sys` *- specifiska sistēmas funkcionalitāte*

Būtībā noderīgākā šī moduļa funkcija ir `argv[]`. Tās saturs ir *Python* programmai nodotie komandrindas argumenti. Pie tam

```
argv[0]
```

satur pašas programmas nosaukumu.

Piemērs:

```
#!/usr/bin/python
if argv[1]:
    print "Atveram failu "+argv[1]
else:
    print "Nav noraditi parametri"
```

Uzdevums: kāds būs šīs programmas izpildes rezultāts?

Modulis `re` *- regulāras izteiksmes*

Šo moduli sīkāk šajā grāmatas daļā neaplūkosim, jo nepieciešamā funkcionalitāte ir aplūkota grāmatas pirmajā daļā "Ievads *Python*".

Modulis `math` *- matemātiskās funkcijas*

```
ceil(x)
```

Funkcijas rezultāts ir x noapaļojums uz augšu, t.i., mazākais veselais skaitlis, kas lielāks vai vienāds ar x.

```
fabs(x)
```

Funkcijas rezultāts ir x absolūtā vērtība, t.i., |x|.

```
floor(x)
```

Funkcijas rezultāts ir x noapaļojums uz leju, t.i., lielākais veselais skaitlis, kas mazāks vai vienāds ar x.

```
modf(x)
```

Funkcijas rezultāts ir rinda, kura satur divus elementus - x daļas un veselo vērtību. Piemēram,

```
modf(2.5)
```

rezultāts ir

```
(0.5, 2.0)
```

```
exp(x)
```

Funkcijas rezultāts ir e^x

```
log(x)
log(x, baze)
```

Funkcijas rezultāts ir x logaritms norādītajai bāzei. Ja bāze nav norādīta, tad rezultāts ir x naturālais logaritms (t.i., logaritms bāzei e).

```
pow(x,y)
```

Funkcijas rezultāts ir x^y

sqrt(x)

Funkcijas rezultāts ir x kvadrātsakne

acos(x)

Funkcijas rezultāts ir x arkkosinuss (izteikts radiānos)

asin(x)

Funkcijas rezultāts ir x arksinuss (izteikts radiānos)

atan(x)

Funkcijas rezultāts ir x arktangenss (izteikts radiānos)

atan2(y,x)

Funkcijas rezultāts ir y/x arktangenss, izteikts radiānos. Rezultāts ir robežās no -pi līdz `pi`. Atšķirība no `atan(x)` ir tāda, ka ir zināmas y un x zīmes, līdz ar to ir iespējams korekti aprēķināt kvadrantu.

cos(x)

Funkcijas rezultāts ir x (izteikts radiānos) kosinuss

sin(x)

Funkcijas rezultāts ir x (izteikts radiānos) sinuss

tan(x)

Funkcijas rezultāts ir x (izteikts radiānos) tangenss

degrees(x)

Funkcijas arguments x ir leņķis, izteikts radiānos, un rezultāts - leņķis, izteikts grādos

radians(x)

Funkcijas arguments x ir leņķis, izteikts grādos, un rezultāts - leņķis, izteikts radiānos

pi

Matemātiskā konstante `pi` (~3,14)

e

Matemātiskā konstante e

Modulis `random` *- gadījuma skaitļu ģenerators*

seed()
seed(x)

Gadījuma skaitļu ģeneratora inicializācija ar skaitli. Ja šai funkcijai netiek norādīts arguments, par argumentu tiek izmantots tekošais sistēmas laiks sekundēs.

randint(a,b)

Funkcijas rezultāts ir tāds gadījuma skaitlis N, ka a<=N<=b, t.i., robežās no a līdz b.

random()

Funkcijas rezultāts ir gadījuma daļskaitlis robežās no 0.0 līdz 1.0 (neieskaitot).

Modulis `string` *- darbs ar teksta rindām*

Teksta rindas tika aplūkotas šīs grāmatas pirmajā daļā "Ievads *Python*". Pēc pirmajā daļā redzamā teksta rindu apstrādes funkciju pieraksta tagad mēs varam noteikt, ka teksta rindas apstrādes funkcijas ir nekas cits, kā... objekts. Attiecīgi pierakstā

```
string.funkcija(argumenti)
```

funkcija ir nekas cits, kā objekta `string` metode. Aplūkosim populārākās šī objekta metodes:

```
capitalize(teksts)
```

Metodes rezultāts ir teksta rinda `teksts`, kurā vienīgais lielais burts ir sākumburts. Pārējie teksta rindā sastopamie burti tiek konvertēti uz mazajiem.

Piemērs:

```
#!/usr/bin/python
import string
teksts = "Alise Aizspogulija"
print string.capitalize(teksts)
```

Uzdevums: kāds būs šīs programmas izpildes rezultāts?

```
center(teksts, platums)
center(teksts, platums, simbols)
```

Metodes rezultāts ir teksta rinda ar garumu `platums`, kurā rinda `teksts` ir izvietota centrēti. Ja ir norādīts `simbols`, tas tiek izmantots "tukšumu" aizpildīšanai `teksts` labajā un kreisajā pusē.

Piemērs:

```
#!/usr/bin/python
teksts = "Alise Aizspogulija"
print string.center(teksts, 30, ".")
```

Uzdevums: kāds būs šīs programmas izpildes rezultāts?

```
count(teksts, apaksteksts)
```

Metodes rezultāts ir skaits, cik reizes teksta rindā `teksts` parādās rinda `apaksteksts`

```
find(teksts, apaksteksts)
```

Metodes rezultāts ir pozīcija, kurā teksta rindā `teksts` parādās rinda `apaksteksts`. Ja apakšrinda nav atrodama, rezultāts ir `-1`.

```
isalnum(teksts)
```

Metodes rezultāts ir `true`, ja visi teksta rindā `teksts` sastopamie simboli ir vai nu burti, vai cipari (t.i., nav speciālie simboli). Pretējā gadījumā rezultāts ir `false`.

```
isalpha(teksts)
```

Metodes rezultāts ir `true`, ja visi teksta rindā `teksts` sastopamie simboli ir burti. Pretējā gadījumā rezultāts ir `false`.

```
isdigit(teksts)
```

Metodes rezultāts ir `true`, ja visi teksta rindā `teksts` sastopamie simboli ir cipari. Pretējā gadījumā rezultāts ir `false`.

`islower(teksts)`

Metodes rezultāts ir `true`, ja visi teksta rindā `teksts` sastopamie burti ir pierakstīti apakšējā reģistrā (mazie burti). Pretējā gadījumā rezultāts ir `false`.

`isupper(teksts)`

Metodes rezultāts ir `true`, ja visi teksta rindā `teksts` sastopamie burti ir pierakstīti augšējā reģistrā (lielie burti). Pretējā gadījumā rezultāts ir `false`.

`split(teksts, atdalitajs)`

Metodes rezultāts ir saraksts, kura elementi tiek iegūti no teksta rindas `teksts`, kā atdalītājšablonu izmantojot `atdalitajs`. Piemērs aplūkots grāmatas pirmajā daļā nodaļā "Saraksti".

`join(saraksts, atdalitajs)`

Metodes rezultāts ir teksta rinda, kura tiek iegūta, apvienojot saraksta `saraksts` elementus, tos atdalot ar teksta rindā `atdalitajs` norādīto simbolu.

`upper(teksts)`

Metodes rezultāts ir teksta rinda, kura tiek iegūta, teksta rindā `teksts` aizstājot visus mazos burtus ar lielajiem.

`lower(teksts)`

Metodes rezultāts ir teksta rinda, kura tiek iegūta, teksta rindā `teksts` aizstājot visus lielos burtus ar mazajiem.

Nobeigumam

Šajā grāmatas "Programmēšanas valoda *Python* iesācējiem" daļā tika aplūkotas tās *Python* valodas papildiespējas, kuru pārzināšana ļauj apgūt sarežģītāku programmu veidošanu un dažādu papildmoduļu izmantošanu. Tipiski piemēri ir grafiskā lietotāja interfeisa izmantošana, darbs ar datu bāzēm un multimēdiju. Šīs iespējas tiks aplūkotas nākamajās grāmatas daļās.

Papildus tam, valodu *Python* ir iespējams izmantot jaudīgu Internet mājaslapu veidošanai - līdzīgi, kā izmantojot valodu *PHP*. Ir plāni arī šādu *Python* pielietojumu aplūkot kādā no nākamajām šīs grāmatas daļām.

Ar plašāku informāciju par nākamajām grāmatas daļām iespējams iepazīties manā mājaslapā – `http://home.parks.lv/alvilsb/`. Vairāk informācijas par *Python* iespējams atrast *Python* projekta mājaslapā – `http://www.python.org`.