Programmēšanas valoda

iesācējiem

3. daļa. *Grafiskā lietotāja interfeisa programmēšana ar Tkinter*

Saturs

Ievads

Pēc ilgāka laika lasītāju uzmanībai tiek piedāvāta nākamā - jau trešā - daļa grāmatai "Programmēšanas valoda *Python* iesācējiem".

Grāmatas "Programmēšanas valoda *Python* iesācējiem" galvenais uzdevums ir sniegt lasītājiem ieskatu programmēšanā, izmantojot valodu *Python*. Šajā grāmatas daļā aplūkota grafiskā lietotāja interfeisa (*GUI*) veidošana, izmantojot *Python* standartkomplektācijā iekļauto bibliotēku `Tkinter`. Aplūkoti dažādi `Tkinter` atbalstītie grafiskā lietotāja interfeisa elementi, kā arī pamati darbam ar tiem.

Kas ir grafiskais lietotāja interfeiss?

Ļoti iespējams, lasot šīs grāmatas pirmās divas daļas, lasītājiem, kuri nekad iepriekš nebija mēģinājuši *piespiest* datoru veikt viņu norādītās darbības jeb programmēt to, radās daudz jautājumu. Lasītāji, iespējams, bija cerējuši, ka šajās grāmatas daļās tiks aplūkoti krāsainu multivides aplikāciju vai jaudīgu datu bāžu izveides principi. Nekā - tā vietā tika piedāvāts izveidot vien dažas programmiņas, kuras veica visnotaļ triviālus matemātiskus aprēķinus vai jocīgas darbības ar teksta rindām. Kādēļ tā? Vienkārši - lai apgūtu programmēšanas pamatus (un bieži vien ne tikai), programmas saziņai ar lietotāju pilnīgi pietiek, ja lietotājs ievada komandas, izmantojot tastatūru un saņem programmas paziņojumus par darba rezultātiem. Pat mūsdienās Unix tipa serveros tiek uzskatīts par pilnīgi normālu, ja programma ar lietotāju sazinās, izmantojot līdzīgas metodes, jo bieži vien svarīgāks ir programmas paveiktais darbs, nevis veids, kā programmu komandēt. Šādu lietotāja-datora saziņas veidu sauc par *teksta režīma interfeisu*.

Savulaik - līdz pat 20.gadsimta 80.gadiem - šāds interfeiss bija praktiski vienīgais pieejamais, jo datorus izmantoja pārsvarā matemātiskiem aprēķiniem un skaitļu izvadei nav nepieciešamas ne krāsas, ne arī augstas izšķirtspējas grafiskais displejs.

Attīstoties datorizētām rasēšanas sistēmām, 1960.gadu otrajā pusē parādījās pirmie grafiskie displeji. Tiesa, sākotnēji tie tika izmantoti vienīgi rasējumu attēlošanai un tikai pamazām radās nojausma, ka ar to palīdzību iespējams radīt lietotājam draudzīgu interfeisu. 1968.gadā Dougs Engelbarts Stenfordas pētījumu institūtā radīja pirmo *grafiskā lietotāja interfeisa* prototipu, kurā tika izmantotas tekstuālas saites, kuras apzīmēja veicamās darbības un ar kurām lietotājs darbojās, izmantojot jaunievedumu - peli. Grafiskā lietotāja interfeisa ideja pakāpeniski tika attīstīta tālāk, ieviešot tādus jēdzienus kā izvēlnes, ikonas, kā arī daudzus citus, līdz esam nonākuši mūsdienās ar modernajiem, lietotājam draudzīgajiem grafiskajiem interfeisiem.

Kas ir Tkinter?

Vispirms nedaudz jāatkāpjas un jāaplūko grafiskā interfeisa uzbūves principi. Kā zināms, grafiskais lietotāja interfeiss sastāv no daudziem elementiem - izvēlnes, ekrānpogas, ikonas, saraksti, tekstu ievades lauki u.c. Šos elementus nebūt nav jāattēlo pašai programmai - to var "uzticēt" veikt kādai bibliotēkai. Viena no šādām bibliotēkām ir `Tk`. Tā tika izveidota kā paplašinājums skriptu valodai `Tcl`, tomēr tās vienkāršības dēļ tiek plaši izmantota arī kā papildinājums citām programmēšanas valodām. Programmēšanas valodā *Python* šo sasaisti veic bibliotēka `Tkinter`.

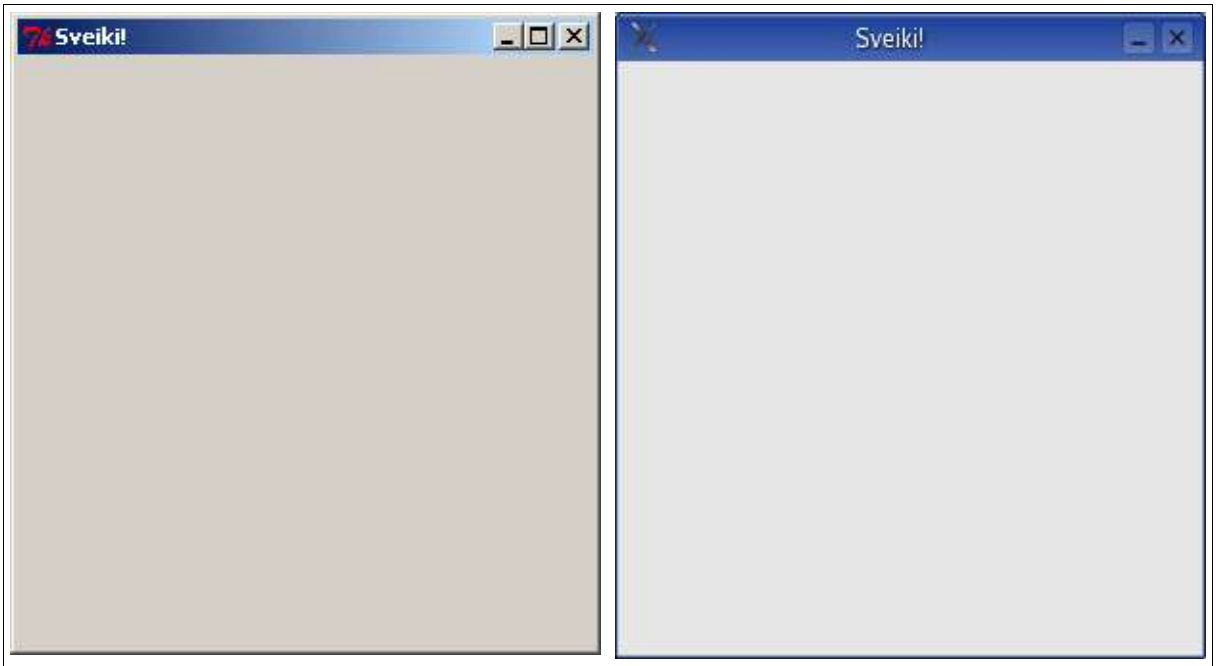
Lai arī `Tkinter` trūkst vairāku mūsdienīgam lietotāja interfeisam nepieciešamu elementu, tomēr, tā kā tā ir iekļauta *Python* standartkomplektācijā un ir vienkārši programmējama, joprojām tiek plaši izmantota.

Logs...

Sāksim ar vienkāršas programmas ievadi un izpildi.

```
from Tkinter import *

# Tukss logs
Logs = Tk()
Logs.title("Sveiki!")
Logs.geometry("300x300")
Logs.mainloop()
```

Programmu izpildot, tā attēlos tukšu logu (1.attēls).



1.attēls. Tukšs logs Microsoft Windows un GNU/Linux operētājsistēmās

Aplūkosim katru šīs programmas rindiņu un ko tā veic.

```
from Tkinter import *
```

Grāmatas pirmajā daļā mēs jau apguvām bibliotēku importēšanu. Šī rindiņa importē bibliotēku `Tkinter`.

Jebkurai programmai ar grafisko lietotāja interfeisu galvenais elements ir **logs**. Visi pārējie interfeisa elementi tiek izvietoti šajā logā. Bibliotēkā `Tkinter` visi elementi ir realizēti kā *Python* klases (atceramies grāmatas otro daļu par objektorientēto programmēšanu!) un logs ir realizēts kā klase `Tk`. Tā kā mūsu programmā logs tiks glabāts objektā `Logs`, tad to vispirms ir jāinicializē kā klases `Tk` objektu. To veic ar rindiņu

```
Logs = Tk()
```

Tagad mūsu rīcībā ir `Tkinter` objekts - `Logs`. Šim objektam varam noteikt tā īpašības, izsaucot atbilstošās objekta metodes. Tā piemēram, loga virsrakstu uzstāda ar metodi `title`, tai norādot nepieciešamo virsraksta tekstu:

```
Logs.title("Sveiki!")
```

Loga sākotnējos izmērus iespējams definēt ar metodi `geometry`, tai norādot loga horizontālos un vertikālos izmērus pikseļos:

```
Logs.geometry("300x300")
```

Uzdevums: Kā norādīt loga izmērus 600 pikseļu horizontāli un 200 pikseļu vertikāli?

Objektā `Logs` ir pieejama vēl kāda metode - `mainloop` (angļu val. *main loop* - galvenais cikls), kura uzsāk loga ciklu. Kas tas? Lieta tāda, ka mūsu programma nav vienīgā, kura vēlas darboties mūsu datorā. Tā kā jebkurā mūsdienu grafiskajā operētājsistēmā nepārtraukti notiek dažādi notikumi (lietotājs mainījis peles kursora pozīciju, lietotājs nospiedis ekrānpogu, lietotājs...), tad, lai nodrošinātu, ka uz šiem notikumiem reaģē "pareizā" programma, t.i., tā, kurai uz konkrēto notikumu ir jāreaģē, sistēma pati veic notikumu apstrādi un sadala tos, katrai programmai nosūtot tikai tai pienākošos notikumus. Katra programma tos apstrādā tālāk un atbilstoši reaģē. Šādai notikumu apstrādei ļoti noder objektorientētā programmēšana - ja ir noteikta klase, kas apraksta, piemēram, logu, un tajā ir realizētas "tukšas" metodes katra ziņojuma apstrādei, mēs varam izveidot klases apakšklasi vai objektu un šīs metodes "papildināt", lai nodrošinātu, ka programma darbojas tā, kā mēs to vēlamies. Gadījumā ar `Tkinter` pēc loga konfigurēšanas (t.i., atbilstošo objekta metožu izsaukumiem) un `mainloop` metodes izsaukuma viss pārējais programmas teksts ir ietverts šī objekta vai tam piesaistīto objektu metodēs. Tātad, mūsu programmas pēdējā rindiņa ir

```
Logs.mainloop()
```

Tkinter lietotāja interfeisa plānošana

`Tk` grafiskā lietotāja interfeisa bibliotēka ir pieejama daudzām operētājsistēmām - gan *GNU/Linux* un citām *Unix* tipa sistēmām, gan *Microsoft Windows*, gan *MacOS*... Katrā no tām interfeisa elementi izskatās "citādi" - tā piemēram, ekrānpogu augstums un platums var atšķirties, tekstu fonti jeb šrifti arī... Ja, piemēram, mēs vēlētos noteikt, ka mūsu logā jāatrodas ekrānpogai, pie kam tās platums ir 60 pikseļi un tai jāsatur tekstu "`Labilabi`", *Microsoft Windows* operētājsistēmā tā izskatīsies kā visnotaļ normāla ekrānpoga, savukārt *GNU/Linux* operētājsistēmā teksts "`Labilabi`" būs "apgriezts", jo neietilps noteiktajā ekrānpogas platumā. Par laimi, `Tkinter` atbalsta iespēju noteikt, ka interfeisa elementu izmēri tiek automātiski pielāgoti videi, kurā tiek izpildīta mūsu programma. Tas nozīmē, ka *GNU/Linux* operētājsistēmā ekrānpoga, iespējams, būs lielāka, nekā *Microsoft Windows* vidē. Tas rada nākamo problēmu - kā, piemēram, "blakus" novietot divas ekrānpogas? Precīzi nosakot katras atrašanās vietu, tās vai nu atradīsies tālu viena no otras, vai arī "pārklāsies".

Lai novērstu šādas problēmas, kļūst skaidrs, ka "jāatbrīvojas" no interfeisa elementu precīza izvietojuma. `Tkinter` atbalsta šādu iespēju, izmantojot t.s. **rāmi** (*frame*). To visvieglāk iztēloties, iedomājoties tabulu, kuras katrā rūtiņā tiek izvietots tieši viens interfeisa elements (2.attēls).

	0	**1**
0	[TEKSTA IEVADES LAUKS 1]	[TEKSTA IEVADES LAUKS 2]
1		[EKRĀNPOGA "`Labi`"]

2.attēls: grafiskā lietotāja interfeisa elementu izvietojums

Katrai rūtiņai ir sava "adrese" - rinda un kolona. Tās numurē, sākot ar 0. Tā piemēram, teksta ievades lauka 1 "adrese" ir (0,0) un teksta ievades lauka 2 - (0,1). Ekrānpoga "`Labi`" atrodas nākamajā tabulas rindiņā un tās adrese ir (1,1).

Mainoties tabulas jeb rāmja izmēriem, elementi tiks "attālināti" vai "satuvināti". Mainoties elementu izmēriem, tiks palielinātas vai samazinātas rāmja rūtiņas. Tas notiks automātiski, līdz ar to programmētājam vairs nav precīzi jārēķina katra interfeisa elementa atrašanās pozīciju.

Kopumā kļūst skaidrs, ka, plānojot `Tkinter` lietotāja interfeisu, ir jāizvēlas interfeisa elementus, to secību un katram elementam jānosaka rāmja rindiņas un kolonas numuru. Par pārējo parūpēsies `Tkinter`.

Runāsim latviski!

Līdz šim visās *Python* programmās mēs izvairījāmies no latviešu valodas simbolu iekļaušanas programmu tekstos. Programmām, kuras darbojas teksta režīmā un veic aprēķinus, tas ir visnotaļ pieņemami, tomēr, veidojot grafisko lietotāja interfeisu, gribētos, lai programma spētu ar mums "runāt" mūsu dzimtajā valodā.

Lai norādītu *Python*, ka mūsu programmā iekļautie simboli, kas neietilpst standarta `ASCII` kodu tabulā, nav jāuzskata par kļūdu, programmas sākumā jāpievieno rindiņu:

```
# -*- coding: KODĒJUMS -*-
```

Kur `KODĒJUMS` ir mums nepieciešamā simbolu kodējuma nosaukums. Mūsdienās plaši izmanto kodējumu `UTF-8`, kurš ļauj kodēt jebkuras valodas jebkuru simbolu. To arī izmantosim:

```
# -*- coding: utf-8 -*-
```

Turpmāk katras programmas sākumā ievietosim šādu rindiņu, lai gūtu iespēju sazināties ar lietotāju latviski.

Interfeisa elementi

Paziņojumi

Kamēr mēs vēl "neprotam" izveidot pilnvērtīgu grafisko interfeisu, mums nav nekādu iespēju, kā paziņot lietotājam "sakarīgu" informāciju. `Tkinter` bibliotēka nodrošina informācijas sniegšanu, izmantojot paziņojumus. Lai izvadītu paziņojumu, jāizmanto bibliotēku `tkMessageBox`. To izmanto šādi:

```
# -*- coding: utf-8 -*-
from Tkinter import *
import tkMessageBox

# Tukss logs ar pazinojumu
Logs = Tk()
Logs.title("Sveiki!")
Logs.geometry("300x300")
tkMessageBox.showinfo("Paziņojums", "Šis ir testa paziņojums...")
Logs.mainloop()
```

Uzdevums: Kas notiek, izpildot šo programmu?

Rāmis

Kā jau vienojāmies iepriekšējā nodaļā, lietotāja interfeisu veidosim, izmantojot rāmi. Rāmis `Tkinter` bibliotēkā ir realizēts kā klase `Frame`. Lai to iekļautu mūsu programmas logā, definēsim to pirms loga `mainloop` metodes izsaukuma:

```
Ramis = Frame(Logs)
Ramis.pack()
```

Pirmajā rindiņā tiek norādīts, ka objekts `Ramis` pieder pie klases `Frame` un tiek piesaistīts objektam `Logs`. Otrajā rindiņā tiek izpildīta objekta metode `pack`, kura veic jaunizveidotā rāmja izvietošanu logā.

Starp citu, rāmi ir iespējams piesaistīt arī citam rāmim, norādot to kā objektu, kuram šo rāmi piesaistīsim.

Uzdevums: Kā izskatīsies pilns programmas teksts?

Ekrānpogas

Ekrānpogas ir interfeisa elementi, kas vizuāli atgādina kādas iekārtas vadības pogas vai slēdžus, uz kuriem ir to nozīmi aprakstošs teksts un kuras ir iespējams "nospiest", izmantojot peles kursoru.

Bibliotēkā `Tkinter` ekrānpogas ir realizētas klasē `Button`. Tā piemēram, izveidosim ekrānpogu ar uzrakstu "`Darbība 1`", kuru nospiežot, tiks izvadīts paziņojums.

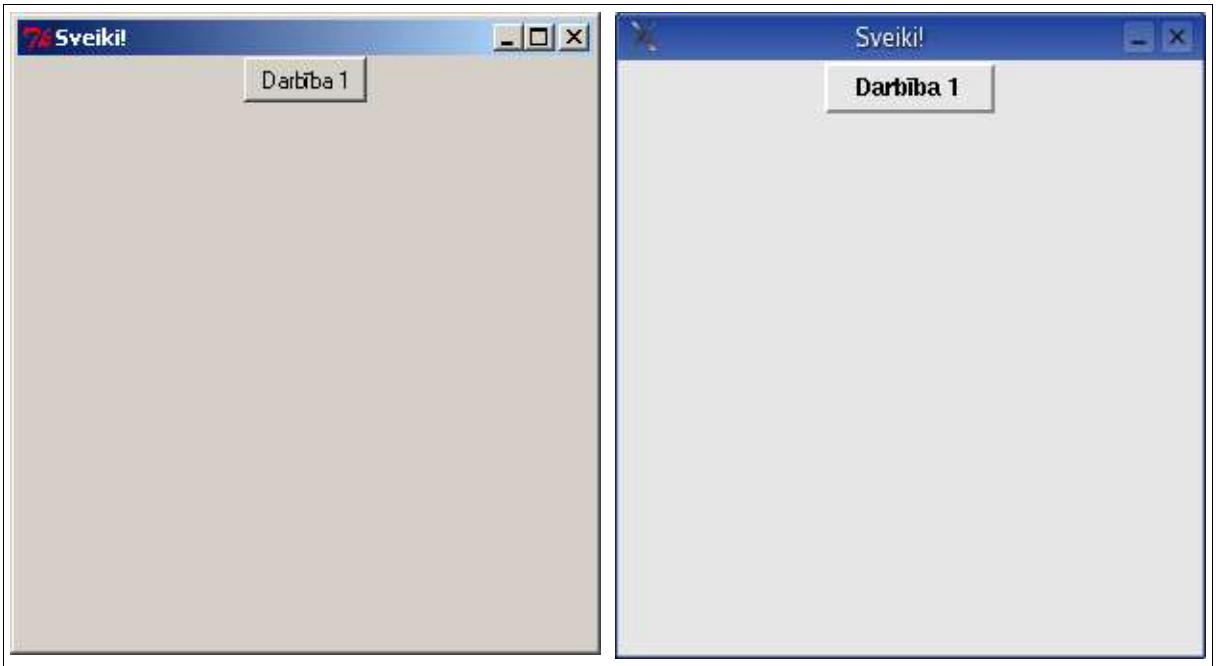
```
  # -*- coding: utf-8 -*-
  from Tkinter import *
  import tkMessageBox

  # Ekranpogas apstrade
  def Darbiba1():
      tkMessageBox.showinfo("Paziņojums", "Veicām darbību 1...")

  # Galvenais logs
  Logs = Tk()
  Logs.title("Sveiki!")
  Logs.geometry("300x300")
  Ramis = Frame(Logs)
  Ramis.pack()
  PogaDarbiba1 = Button(Ramis, text="Darbība 1", command=Darbiba1, width=9,
height=1)
  PogaDarbiba1.grid(row=0, column=0)
  Logs.mainloop()
```

Aplūkosim šo programmu soli pa solim.

Pirmais jaunums ir funkcija `Darbiba1`, kura izvada paziņojumu. Lai gan... patiesībā nekāds jaunums tas nav - funkcija neveic neko nesaprotamu.

Vispirms tiek izveidots klases `Tk` objekts `Logs`, kuram tiek uzstādīti atbilstoši parametri. Tad tam tiek piesaistīts klases `Frame` objekts `Ramis`. Nākamā darbība ir visinteresantākā - tiek veidots klases `Button` objekts `PogaDarbiba1`. Klases konstruktoram ir jānorāda vairākus parametrus. Pirmais no tiem ir objekts, kuram poga tiks "piesaistīta". Mūsu gadījumā tas ir objekts `Ramis`. Otrs parametrs nosaka pogas uzrakstu - "`Darbība 1`". Trešais parametrs nosaka funkciju, kura tiks izpildīta, nospiežot ekrānpogu - mūsu gadījumā tā ir funkcija `Darbiba1`, kuru definējām jau iepriekš. Pēdējie divi parametri nosaka ekrānpogas platumu un augstumu. Šeit jābūt uzmanīgam, jo šie parametri tiek noteikti nevis pikseļos, bet gan simbolos. Tā kā teksts "`Darbība 1`" satur deviņus simbolus, tad pogas platums simbolos ir 9, un augstums ir 1. Nākamā koda rindiņa nosaka, ka jaunizveidotā poga atradīsies rāmja rūtiņā rindiņā ar adresi 0 un kolonā ar adresi 0. Pēc tā seko loga `mainloop` metodes izsaukums.

Programmu izpildot, tiks attēlots šāds logs (3.attēls):



3.attēls: ekrānpogas Microsoft Windows un GNU/Linux vidēs

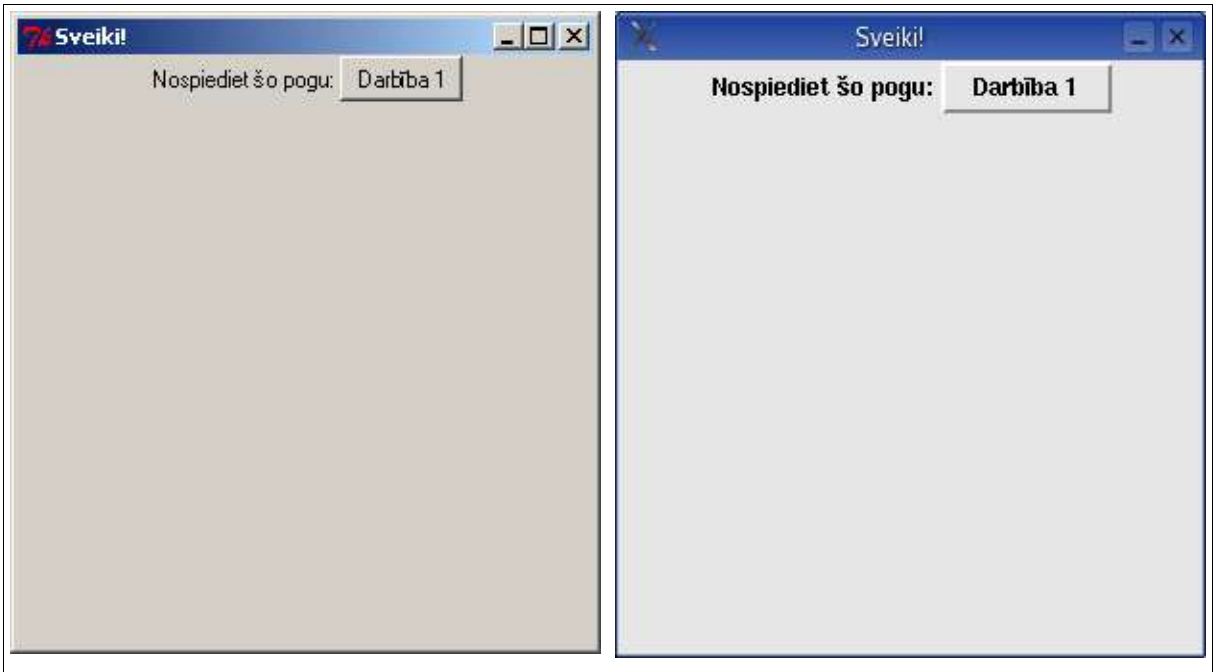
Uzdevums: izveidojiet otru ekrānpogu ar virsrakstu "`Darbība 2`" un izpildes funkciju `Darbiba2`, kura izvada atbilstošu paziņojumu! Pogai rāmī jāatrodas rūtiņā, kas atrodas rindiņā 0 un kolonā 1. Kāds būs programmas izpildes rezultāts?

Statiskie teksti

Neraugoties uz to, ka grafiskais lietotāja interfeiss ir daudz draudzīgāks par tekstuālo, bieži vien programmās nepieciešams attēlot statiskus paziņojumus. Tie var saturēt gan īsu informāciju par veicamajām darbībām (piemēram, uzraksts "`Funkcijas argumenti:`" pirms teksta ievades lauka), gan saturēt dažādu statusa informāciju (piemēram, "`Veicu apstrādi. 10% pabeigti.`"). `Tkinter` bibliotēkā **statiskie teksti** realizēti klasē `Label`, pie kam paši teksti tiek glabāti objektā, kas pieder klasei `StringVar`. Aplūkosim piemēru:

```
  # -*- coding: utf-8 -*-
  from Tkinter import *
  import tkMessageBox

  # Ekranpogas apstrade
  def Darbiba1():
      tkMessageBox.showinfo("Paziņojums", "Veicām darbību 1...")

  # Galvenais logs
  Logs = Tk()
  Logs.title("Sveiki!")
  Logs.geometry("300x300")
  Ramis = Frame(Logs)
  Ramis.pack()
  Teksts = StringVar()
  Pazinojums = Label(Ramis, textvariable=Teksts, fg="black")
  Pazinojums.grid(row=0, column=0)
  Teksts.set("Nospiediet šo pogu:")
  PogaDarbiba1 = Button(Ramis, text="Darbība 1", command=Darbiba1, width=9,
height=1)
  PogaDarbiba1.grid(row=0, column=1)
  Logs.mainloop()
```

Izpildot šo programmu, būs redzams šāds logs (4.attēls):



4.attēls: statiskā teksta elements Microsoft Windows un GNU/Linux vidēs

Principā programmas teksts mums jau ir pazīstams, taču tajā ir veiktas vairākas izmaiņas. Ekrānpoga atrodas rāmja rūtiņā ar kolonas adresi 1, nevis 0, jo kolonas adresē 0 atradīsies statiskais teksts. Teksta elementu realizējam ar vairākām darbībām. Vispirms tiek izveidots jauns klases `StringVar` objekts `Teksts`. Pēc tam tiek izveidots jauns klases `Label` objekts `Pazinojums`. Klases `Label` konstruktoram tiek norādīts objekts, pie kā jāpiesaista jauno objektu (`Ramis`), klases `StringVar` objekts, kurš satur informāciju par tekstu (`Teksts`), kā arī teksta krāsa (`fg` - *foreground*). Krāsu iespējams norādīt gan pēc tās `RGB` vērtībām, gan arī kādu no

iepriekš noteiktajām krāsām, piemēram, `black` (melns), `white` (balts), `red` (sarkans), `yellow` (dzeltens), `blue` (zils), `green` (zaļš) u.c. Nākamajā rindiņā izveidotais teksta elements tiek piesaistīts rāmim, pēc kā tiek uzstādīts objekts `Teksts`, lai tas saturētu mums vēlamo tekstu. Šo tekstu iespējams mainīt jebkurā brīdī, ja tas nepieciešams. Ja teksta elementā attēlotajam tekstam nav jāmainās, ir iespējams iztikt arī bez `StringVar` objekta, teksta elementa konstruktoram `textvariable` vietā uzstādot `text` un norādot attēlojamo tekstu, piemēram, šādi:

```
  text="Nospiediet šo pogu:"
```

Uzdevums: pārveidojiet augstāk redzamo programmu, lai atbrīvotos no StringVar objekta lietojuma!

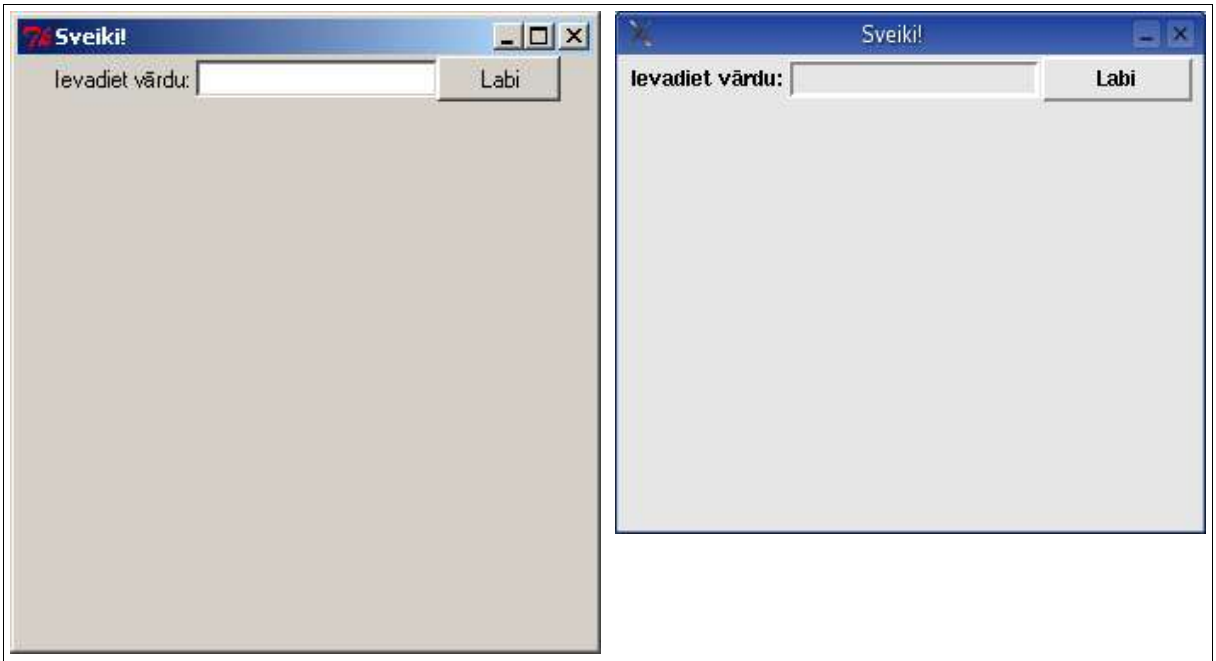
Tekstu ievades lauki

Diemžēl nav iespējams paredzēt visas darbības un visus parametrus, kādus lietotājs vēlēsies paziņot mūsu programmai. Veidojot programmas teksta režīmā, mēs varējām lietotājam pieprasīt ievadīt mums nepieciešamo informāciju, izmantojot tastatūru. Arī `Tkinter` ir iespējams kas līdzīgs, izmantojot **tekstu ievades laukus**. Tekstu ievades lauki ir realizēti klasē `Entry`. Ievades lauka izveide ir pavisam vienkārša - nepieciešams izveidot jaunu klases `Entry` objektu, tā konstruktoram norādot objektu, pie kura tas tiks piesaistīts, kā arī "iekārtot" šo objektu rāmī līdzīgi, kā ekrānpogu vai statiskā teksta objektu. Aplūkosim programmas tekstu:

```
  # -*- coding: utf-8 -*-
  from Tkinter import *
  import tkMessageBox

  # Ekranpogas apstrade
  def Darbiba1():
      tkMessageBox.showinfo("Paziņojums", "Jūsu vārds ir "+Vards.get())

  # Galvenais logs
  Logs = Tk()
  Logs.title("Sveiki!")
  Logs.geometry("300x300")
  Ramis = Frame(Logs)
  Ramis.pack()
  Teksts = StringVar()
  Pazinojums = Label(Ramis, textvariable=Teksts, fg="black")
  Pazinojums.grid(row=0, column=0)
  Teksts.set("Ievadiet vārdu:")
  Vards = Entry(Ramis)
  Vards.grid(row=0, column=1)
  PogaDarbiba1 = Button(Ramis, text="Labi", command=Darbiba1, width=9,
height=1)
  PogaDarbiba1.grid(row=0, column=2)
  Logs.mainloop()
```

Ir veiktas izmaiņas funkcijā `Darbiba1`. Redzams, ka tiek izsaukta objekta `Vards`

metode `get`. Šīs metodes rezultāts ir teksta ievades laukā ievadītais teksts. Programmas darbības rezultātā redzams šāds logs (5.attēls):



5.attēls: teksta ievades lauki Microsoft Windows un GNU/Linux operētājsistēmās

Uzdevums: izskaidrojiet programmas darbību!

Izvēlnes

Lai nodrošinātu lietotājam draudzīgu darbu, programmas ar grafisko lietotāja interfeisu parasti piedāvā jebkuru ar programmu veicamo darbību aktivizēt, to izvēloties ar peles kursora palīdzību. Diemžēl bieži vien iespējamo darbību skaits ir pārāk liels, lai tās izvietotu programmas galvenajā logā. Tad talkā nāk **izvēlnes**. Darbības tiek sagrupētas pēc to nozīmes (piemēram, "`Fails`", "`Labot`", "`Palīgs`") un katrai grupai tiek piesaistītas tikai tai atbilstošās darbības (piemēram, grupai "`Fails`" varētu piesaistīt darbības "`Jauns`", "`Atvērt`", "`Saglabāt`" u.c.). Logā tiek attēlotas tikai darbību grupas, kas ļauj ietaupīt vietu uz ekrāna. Izvēloties jeb aktivizējot kādu darbību grupu, saraksta veidā tiek attēlotas tai piesaistītās darbības.

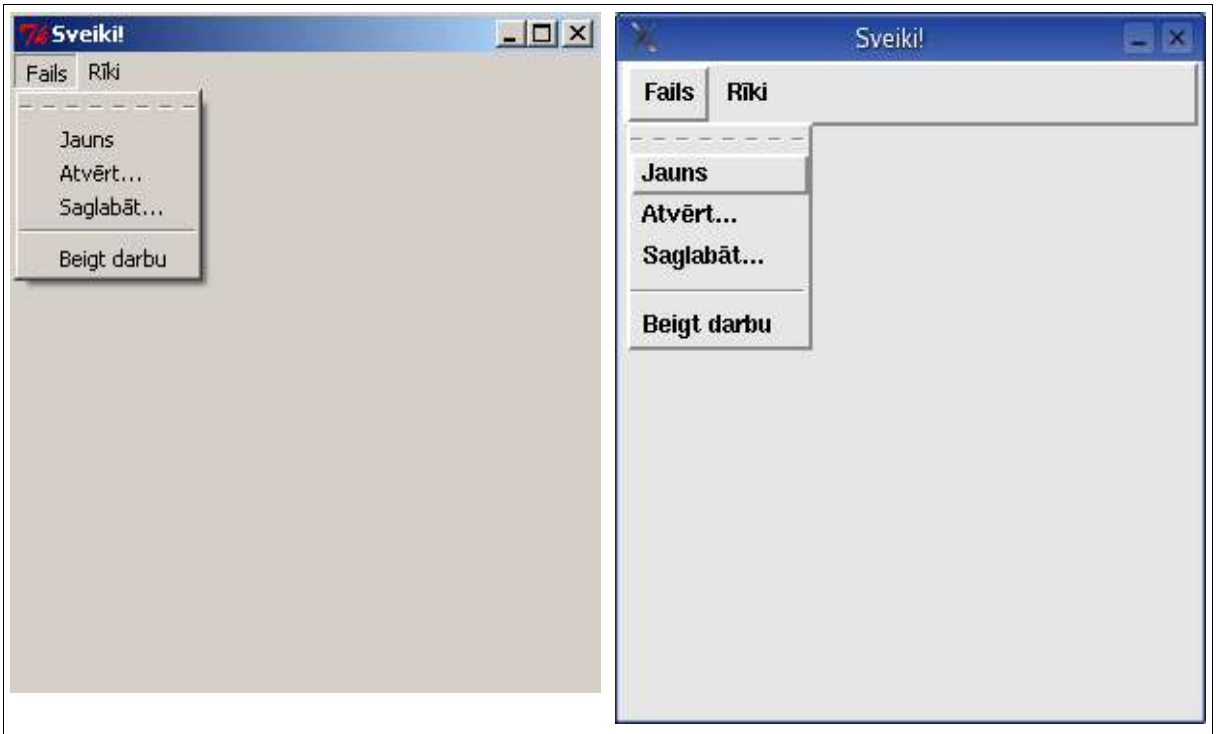
`Tkinter` bibliotēkā izvēlnes ir realizētas klasē `Menu`. Izvēlnes grupas tiek realizētas kā klases `Menu` objekti, tos piesaistot "galvenajam" objektam. Tālāk jau ir iespējams izvēlnēm pievienot komandas. Visbeidzot jāveic loga objekta konfigurēšanu, nosakot, ka tajā ir realizēta izvēlne. Aplūkosim piemēru:

```
# -*- coding: utf-8 -*-
from Tkinter import *
import tkMessageBox

# Izvelnes punktu apstrade
def Komanda():
    tkMessageBox.showinfo("Paziņojums", "Izvēlēta komanda no izvēlnes")

# Galvenais logs
Logs = Tk()
Logs.title("Sveiki!")
Logs.geometry("300x300")

# Izvelne
Izvelne = Menu(Logs)
FailsIzvelne = Menu(Izvelne)
RikiIzvelne = Menu(Izvelne)
Izvelne.add_cascade(label="Fails", menu=FailsIzvelne)
Izvelne.add_cascade(label="Rīki", menu=RikiIzvelne)
FailsIzvelne.add_command(label="Jauns", command=Komanda)
FailsIzvelne.add_command(label="Atvērt...", command=Komanda)
FailsIzvelne.add_command(label="Saglabāt...", command=Komanda)
FailsIzvelne.add_separator()
FailsIzvelne.add_command(label="Beigt darbu", command=Logs.destroy)
RikiIzvelne.add_command(label="Iestatījumi...", command=Komanda)
Logs.config(menu=Izvelne)

# Loga cikls
Logs.mainloop()
```

Lai programmas teksts būtu vieglāk saprotams, no tā ir dzēsti fragmenti, kas darbojās ar iepriekš aplūkotajiem interfeisa elementiem - arī rāmi. Jāatceras, ka izvēlni **NAV** jāsasaista ar rāmi!

Vispirms tiek izveidots klases `Menu` objekts `Izvelne`. Ar to tiek sasaistīti divi citi klases `Menu` objekti - `FailsIzvelne` un `RikiIzvelne`. Tad objektam `Izvelne` tiek izsaukta metode `add_cascade`, kas pievieno jaunu izvēlnes elementu grupu. Vienu no tām nosaucam par "`Fails`" un norādām, ka tai atbilstošais izvēlnes objekts ir `FailsIzvelne`, bet otru - par "`Rīki`" un norādām, ka tai atbilst izvēlnes objekts `RikiIzvelne`. Tālāk tiek pievienoti izvēlnes punkti objektiem `FailsIzvelne` un `RikiIzvelne`. To veic, izmantojot metodi `add_command`. Šai metodei jānorāda izvēlnes elementa virsrakstu, kā arī funkciju, kuru jāizpilda, kad tiek aktivizēts atbilstošais izvēlnes punkts. Gandrīz visiem izvēlnes punktiem šoreiz izmantosim vienu funkciju - `Komanda`. Izņēmums ir izvēlnes punkts "`Beigt darbu`", kurš izpilda objekta `Logs` metodi `destroy`, kura pārtrauc loga ciklu un beidz programmas darbu. Redzams, ka tiek izsaukta arī metode `add_separator`, kura ievieto izvēlnē atdalošo līniju. Tā ļauj padarīt izvēlni vizuāli pievilcīgāku, ja izvēlnē ir iekļautas darbības, kuru nozīme atšķiras. Visbeidzot tiek iestatīti loga parametri, izsaucot objekta `Logs` metodi `config` un norādot tai, ka loga izvēlne atrodama objektā `Izvelne`.

Izpildot programmu, redzams, ka logā atrodama šāda izvēlne (6.attēls):



6.attēls: izvēlne Microsoft Windows un GNU/Linux operētājsistēmās

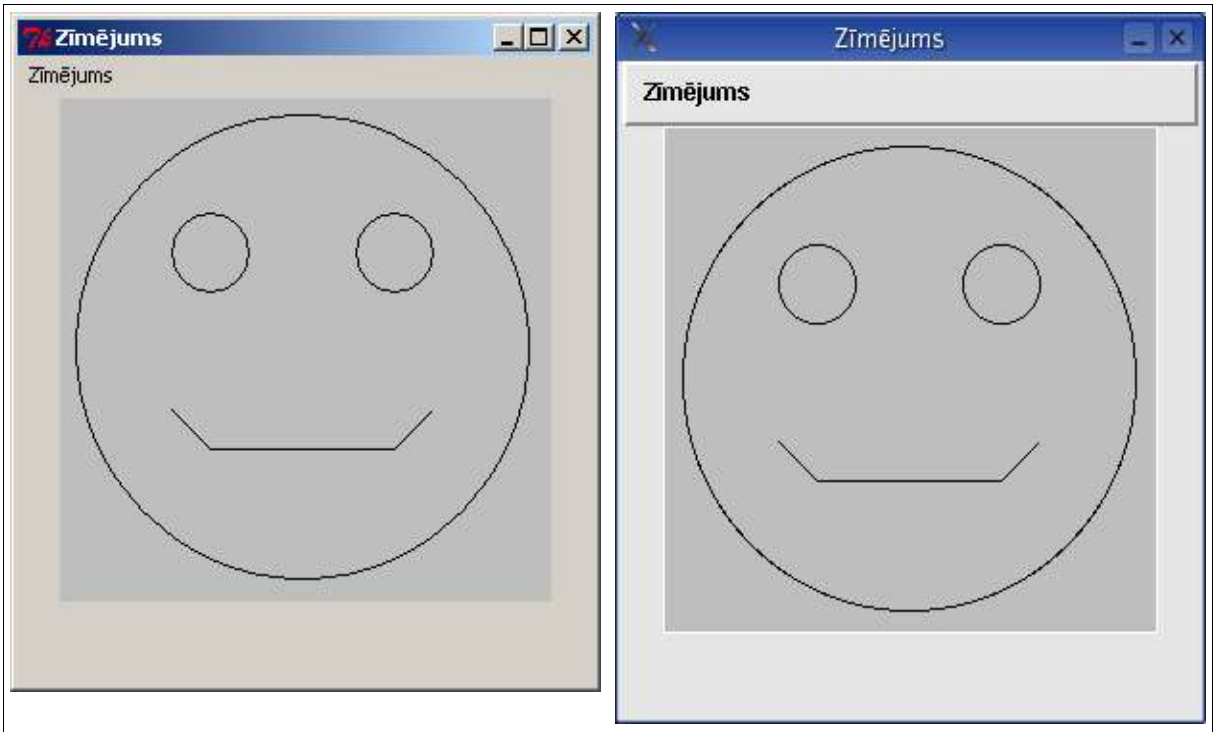
Zīmēšanas virsma

Bieži vien nepietiek, ja programma attēlo tikai ekrānpogas, tekstus un tekstu ievades laukus. Galu galā, līdzīgu lietotāja interfeisu ir iespējams realizēt arī teksta režīmā! Galvenā datora grafiskā displeja priekšrocība ir spēja attēlot detalizētus grafikas elementus. Tie var būt gan vienkārši atsevišķi pikseļi, gan līnijas, gan riņķa līnijas un, galu galā - arī attēli. Vienkāršāko grafisko primitīvu attēlošanai bibliotēkā `Tkinter` tiek izmantota **zīmēšanas virsmas** klase `Canvas`. Klases `Canvas` elementus varam izvietot arī ārpus rāmja objekta, jo parasti grafisko primitīvu attēlošanai, atšķirībā no interfeisa vadības elementiem (ekrānpogām, tekstu ievades laukiem) tiek izmantota ievērojama loga daļa. Papildus tam bieži vien zīmēšanas virsmas ir ar fiksētiem izmēriem, jo dažādās operētājsistēmās (*GNU/Linux*, *Microsoft Windows*) to izmēri neatšķiras. Virsmai ir iespējams piesaistīt dažādus objektus, kuri apraksta grafiskos primitīvus. Aplūkosim piemēru:

```
# -*- coding: utf-8 -*-
from Tkinter import *
import tkMessageBox

# Izvelnes punktu apstrade
def Sejina():
    Virsma.delete("zimejums")
    Virsma.create_oval(10, 10, 246, 246, tags="zimejums")
    Virsma.create_oval(60, 60, 100, 100, tags="zimejums")
    Virsma.create_oval(156, 60, 196,100, tags="zimejums")
    Virsma.create_line(60, 160, 80, 180, tags="zimejums")
    Virsma.create_line(80, 180, 176, 180, tags="zimejums")
    Virsma.create_line(176, 180, 196, 160, tags="zimejums")

# Galvenais logs
Logs = Tk()
Logs.title("Zīmējums")
Logs.geometry("300x300")

# Izvelne
Izvelne = Menu(Logs)
ZimejumsIzvelne = Menu(Izvelne)
Izvelne.add_cascade(label="Zīmējums", menu=ZimejumsIzvelne)
ZimejumsIzvelne.add_command(label="Sejiņa", command=Sejina)
ZimejumsIzvelne.add_separator()
ZimejumsIzvelne.add_command(label="Beigt darbu", command=Logs.destroy)
Logs.config(menu=Izvelne)

# Virsma
Virsma = Canvas(Logs, width=256, height=256, bg="gray")
Virsma.pack()

# Loga cikls
Logs.mainloop()
```

Pēc galvenā loga un izvēlnes sagatavošanas tiek sagatavota zīmēšanas virsma, izveidojot klases `Canvas` objektu `Virsma`. Objekta konstruktoram tiek norādīts, ka tas tiek piesaistīts objektam `Logs`, virsmas platums un augstums pikseļos ir 256, kā arī virsmas fona krāsa ir pelēka (`gray`). Pēc tam tiek izpildīta objekta `Virsma` metode `pack`, kas iekārto objektu logā.

Izpildot programmu un izvēlnē izvēloties komandu `Zīmējums -> Sejiņa`, tiek izveidots vienkāršs attēls (7.attēls):



7.attēls: zīmēšanas virsma Microsoft Windows un GNU/Linux operētājsistēmās

Zīmēšana notiek funkcijā `Sejina`, kura tiek izsaukta, lietotājam izvēloties atbilstošo izvēlnes punktu. Šeit jāiegaumē, ka katram grafiskajam primitīvam, kurš tiek izveidots objektā `Virsma`, ir jāpiešķir t.s. **birku** (tag). Birkas ļauj identificēt grafisko primitīvu grupas, lai, ja nepieciešams, tos varētu dzēst vai veikt ar tiem citas manipulācijas. Funkcijas `Sejina` pirmais operators dzēš visus objektam `Virsma` piesaistītos grafiskos primitīvus ar birku `zimejums`. Pārējie operatori veido primitīvus, kas kopā izveido zīmējumu. Klasē `Canvas` iespējams izveidot šādus grafiskos primitīvus:

create_arc

Metode izveido elipses segmentu. Metodei norāda augšējo kreiso un apakšējo labo stūri apgabalam, kurā tiks veidots segments (formā x1, y1, x2, y2), kā arī papildparametrus:

`extent` - segmenta platums grādos. Segmenta zīmēšana tiek uzsākta ar parametra start noteikto leņķi un tiek turpināta pretēji pulksteņa rādītāju virzienam extent grādus.

`fill` - ja šo parametru nenorāda, segments netiek aizkrāsots. Šajā parametrā iespējams norādīt krāsu, ar kuru segments tiks aizkrāsots, piemēram, `fill="black"`.

`outline` - iespējams norādīt segmenta ārējā apveida krāsu, piemēram, outline="red". Pēc noklusējuma tā ir melna.

`start` - segmenta sākumleņķis.

`style` - segmenta stils. Iespējams norādīt vairākus stilus - `PIESLICE`, lai zīmētu visu segmentu (noklusētais), `ARC`, lai zīmētu tikai riņķa līnijas fragmentu un `CHORD`, lai zīmētu riņķa līnijas fragmentu, tā galapunktus savienojot ar taisnu līniju. Piemērs: `style=CHORD`.

`tags` - objekta birka, piemēram, `tags="zimejums"`.

`width` - segmenta ārējā apveida platums. Noklusētā vērtība ir 1 pikselis.

create_line

Metode izveido līniju. Metodei norāda līnijas galapunktus formā (x1, y1, x2, y2), kā arī papildparametrus:

`arrow` - nosaka, vai līnijas galapunkti tiks zīmēti kā bultiņas (pēc noklusējuma nē). `arrow=FIRST` nosaka, ka bultiņa tiks zīmēta līnijas pirmajā galapunktā, `arrow=LAST` - ka pēdējā, un `arrow=BOTH` - abos.

`fill` - nosaka līnijas krāsu, piemēram, `fill="blue"`. Pēc noklusējuma tā ir melna.

`width` - nosaka līnijas platumu. Pēc noklusējuma tas ir 1 pikselis.

`tags` - objekta birka, piemēram, `tags="zimejums"`.

create_oval

Metode izveido ovālu jeb elipsi. Metodei norāda ovālu ietverošā taisnstūra augšējā kreisā un apakšējā labā stūra koordinātas formā (x1, y1, x2, y2), kā arī papildparametrus:

`fill` - pēc noklusējuma ovāls netiek aizkrāsots. Šeit iespējams norādīt, kādā krāsā ovālu jāaizkrāso, piemēram, `fill="grey"`.

`outline` - ovāla līnijas krāsa. Pēc noklusējuma melna.

`width` - ovāla līnijas platums. Pēc noklusējuma 1 pikselis.

`tags` - objekta birka, piemēram, `tags="zimejums"`.

Ja ovālu ietverošā taisnstūra malas ir vienādas (t.i., tas ir kvadrāts), tiks iegūta riņķa līnija.

Klase `Canvas` uztur arī citus grafiskos primitīvus un īpašības, kuri šajā grāmatā netiks aplūkoti. Vairāk informācijas par iespējams iegūt *Python* projekta mājaslapā: `http://www.python.org/`.

Uzdevums: Papildiniet šo programmu ar vēl kādu izvēlnes punktu, kuru izvēloties tiek izveidots cits attēls!

Dialoglogi

Bieži vien, veicot kādu darbību, nepieciešams no lietotāja iegūt papildinformāciju. Piemēram, ja tekstu procesorā (piemēram, *OpenOffice.org Writer* vai *Microsoft Word*) izvēlnē izvēlamies darbību `File -> Print`, atveras dialoglogs, kurā tiek pieprasīts izvēlēties printeri, uz kuru drukāt dokumentu, kā arī iespējams izvēlēties dažādus drukāšanas parametrus. Šī informācija nav nepieciešama, kamēr rakstām dokumentu, tādēļ tai nav jāatrodas galvenajā logā. Arī `Tkinter` piedāvā dialoglogu veidošanas iespēju.

Tā kā dialoglogs ir un paliek logs, tā apstrādei jāizveido atsevišķu klasi. Parasti dialoglogam iespējami divi darbības rezultāti: lietotājs izvēlējies nepieciešamos konfigurācijas parametrus un nospiedis ekrānpogu "`Labi`", tā apliecinot, ka vēlas veikt izvēlēto darbību, vai arī nospiedis ekrānpogu "`Atcelt`", tā paziņojot, ka ir pārdomājis un darbību veikt nevēlas. Tas nozīmē, ka dialogloga klases struktūra ir šāda:

- konstruktors - izveido dialoglogu
- ekrānpogas "`Labi`" apstrāde
- ekrānpogas "`Atcelt`" apstrāde

Mēģināsim izveidot šādu dialoglogu. Tā uzdevums būs ļaut lietotājam ievadīt krāsu, kādā attēlosim mūsu zīmējumu.

```
  class ColorInput(Toplevel):
     def __init__(self, parent):
        Toplevel.__init__(self, parent)
        self.transient(parent)
        self.focus_set()
        self.title("Krāsas izvēle")
        self.parent = parent
        self.result = None
        Ramis = Frame(self)
        Ramis.pack()
        Uzraksts = Label(Ramis, text="Krāsa:", fg="black")
        Uzraksts.grid(row=0, column=0)
        self.Krasa = Entry(Ramis)
        self.Krasa.grid(row=0, column=1)
       LabiPoga = Button(Ramis, text="Labi", width=8, height=1,
command=self.labi)
        LabiPoga.grid(row=1, column=0)
       AtceltPoga = Button(Ramis, text="Atcelt", width=8, height=1,
command=self.atcelt)
        AtceltPoga.grid(row=1, column=1)
        self.wait_window(self)
     def labi(self, event=None):
        self.result = self.Krasa.get()
        self.parent.focus_set()
        self.destroy()
     def atcelt(seft, event=None):
        self.result = None
        self.parent.focus_set()
        self.destroy()
```

Aplūkosim šīs klases izveidi soli pa solim.

Klases nosaukums būs `ColorInput`. Klasei tiek norādīts tās virsklases objekts, kuru nosauksim par `Toplevel`. Tālāk tiek veidots klases konstruktors. Dialoglogiem ir svarīgi konstruktora sākumā izpildīt virsklases konstruktoru. To izpilda ar operatoru

```
  Toplevel.__init__(self, parent)
```

Tālāk tiek uzstādītas vairākas topošā dialogloga īpašības - tādas, kā pakļaušana izsaucošajam logam (`transient`), fokusa uzstādīšana jeb vizuālā aktivizēšana (`set_focus`), nosaukuma uzstādīšana (`title`), vecāka norādīšana (`parent`), kā arī rezultāta inicializēšana (`result=None`). Pēc tam tiek izveidoti interfeisa elementi (ar tiem jau esam pazīstami) un tiek izsaukta metode `wait_window`, kura iedarbina dialogloga ciklu un neļauj pārslēgties uz galveno logu, pirms dialoglogs nav aizvērts.

Pēc konstruktora tiek definētas metodes `labi` un `atcelt`, kuras darbojas, nospiežot atbilstošās ekrānpogas. Tās aizpilda dialogloga īpašību `result`, kura satur loga darbības rezultātu, aktivizē izsaucošo logu un izsauc metodi `destroy`, kura likvidē dialogloga objektu un aizver dialoglogu.

Funkciju `Sejina` pārveidosim tā, lai tiktu veikts mūsu jaunā dialogloga izsaukums:

```
def Sejina():
    Dialogs = ColorInput(Logs)
    if Dialogs.result:
        Krasa = Dialogs.result
        Virsma.delete("zimejums")
        Virsma.create_oval(10, 10, 246, 246, tags="zimejums", outline=Krasa)
        Virsma.create_oval(60, 60, 100, 100, tags="zimejums", outline=Krasa)
        Virsma.create_oval(156, 60, 196,100, tags="zimejums", outline=Krasa)
        Virsma.create_line(60, 160, 80, 180, tags="zimejums", fill=Krasa)
        Virsma.create_line(80, 180, 176, 180, tags="zimejums", fill=Krasa)
        Virsma.create_line(176, 180, 196, 160, tags="zimejums", fill=Krasa)
```

Izveidojam jaunu klases `ColorInput` objektu `Dialogs`. Veidojot šo objektu, tiks attēlots dialoglogs un iegūts tā rezultāts. Ja rezultāts ir `None`, varam uzskatīt, ka lietotājs nospiedis ekrānpogu "`Atcelt`" un tālākās darbības nav jāveic. Pretējā gadījumā mainīgajam `Krasa` piešķiram ievadīto krāsu un veicam zīmēšanu, pie kam, veidojot grafiskos primitīvus, jānorāda arī to krāsu. Ovāla gadījumā nepieciešamais parametrs ir `outline`, bet līnijas - `fill`.

Programmu izpildot un tās izvēlnē izvēloties `Zīmējums -> Sejiņa`, atvērsies šāds dialoglogs (8.attēls):

8.attēls: dialoglogs Microsoft Windows un GNU/Linux operētājsistēmās

Teksta ievades lauciņā ievadot krāsas nosaukumu (piemēram, `black`, `red` u.c.) vai tās `RGB` vērtību heksadecimālajā pierakstā (piemēram, `#000000`, `#FF0000` u.c.), "sejiņa" tiks uzzīmēta atbilstošajā krāsā.

Citi interfeisa elementi

Lai arī šīs grāmatas daļas uzdevums nav aplūkot visas `Tkinter` bibliotēkas iespējas un visus tajā iekļautos grafiskā lietotāja interfeisa veidošanas elementus, aplūkosim vēl dažus, kuri noteikti noderēs, izstrādājot programmas.

CheckButton

Klase `CheckButton` ļauj izveidot grafisku pārslēgu. Veidojot objektu, jānorāda objektu, kuram tas tiks piesaistīts, aprakstu, kā arī `IntVar` klases objektu, kurā tiks glabātas vērtības.

Piemērs:

```
ParslegaStatuss = IntVar()
Parslegs = CheckButton(Ramis, text="Pārslēgs", variable=ParslegaStatuss)
Parslegs.grid(row=..., column=...)
Pārslēga vērtību iespējams iegūt un apstrādāt šādi:
Statuss = ParslegaStatuss.get()
if Statuss:
   # darbibas, ko javeic, ja parslegs ir ieslegts
else:
   # darbibas, ko javeic, ja parslegs ir izslegts
```

Uzdevums: Izveidojiet programmu, kuras interfeisā tiktu izmantots pārslēgs!

Failu izvēles dialoglogi

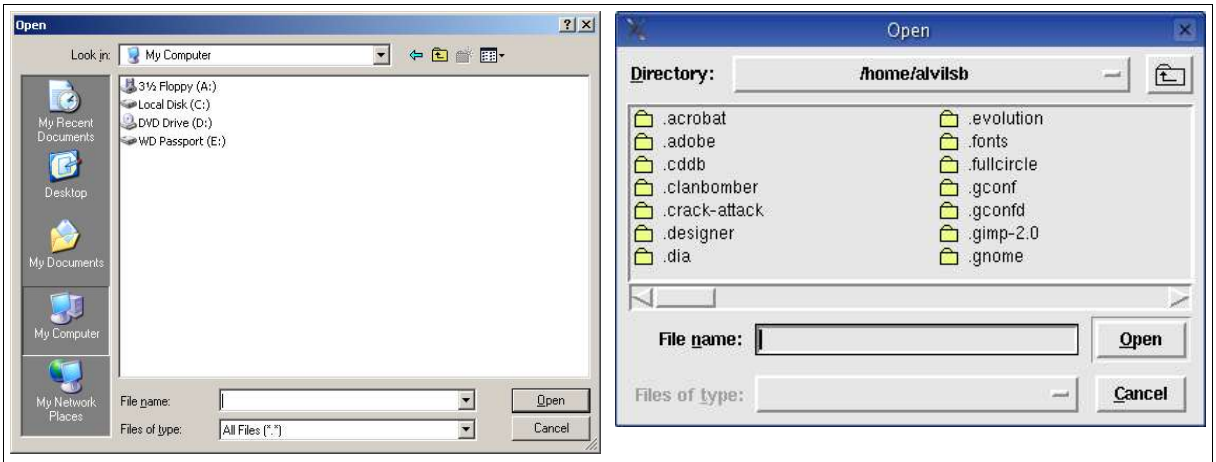
Veidojot programmas, kuras strādā ar failiem, jāizmanto dialoglogus, kuri ļauj lietotājam izvēlēties, kuru failu atvērt, vai tieši otrādi - kādā failā saglabāt datus. Šo iespēju nodrošina klase `tkFileDialog`. Lai to izmantotu, jāimportē bibliotēku `tkFileDialog`, izmantojot operatoru

```
import tkFileDialog
```

Pēc tam iespējams uzzināt, kādu failu lietotājs izvēlējies, izmantojot divas klases `tkFileDialog` metodes:

```
Filename = tkFileDialog.askopenfilename()
```

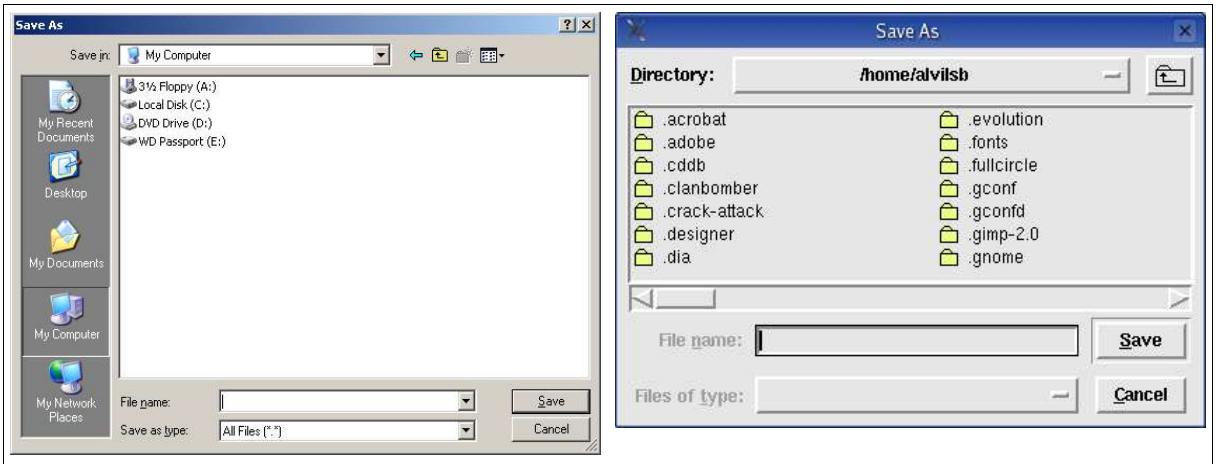
Šī metode izvada standarta failu atvēršanas dialoglogu (9.attēls) un tās rezultāts ir izvēlētā faila nosaukums. Ja lietotājs nospiedis ekrānpogu "`Cancel`" (atcelt), tad rezultāts ir tukša teksta rinda - "".



9.attēls: faila atvēršanas dialoglogs Microsoft Windows un GNU/Linux operētājsistēmās

```
Filename = tkFileDialog.asksaveasfilename()
```

Šī metode izvada standarta failu saglabāšanas dialoglogu (10.attēls) un tās rezultāts ir izvēlētā faila nosaukums. Ja lietotājs nospiedis ekrānpogu "`Cancel`" (atcelt), tad rezultāts ir tukša teksta rinda - "".



10.attēls: faila saglabāšanas dialoglogs Microsoft Windows un GNU/Linux operētājsistēmās

Piemēram, lai veiktu faila atvēršanu, jārīkojas šādi:

```
Filename = tkFileDialog.askopenfilename()
if Filename != "":
   # Veicam faila atversanu
```

To visu saliekot kopā...

Šobrīd mūsu zināšanas par `Tkinter` bibliotēku un *Python* programmēšanas valodu ir pietiekošas, lai izveidotu kādu "īstu" programmu - tādu, no kuras ir kāds praktisks labums.

Izveidosim programmu, kura spēj attēlot kvadrātvienādojuma funkcijas $y=ax^2+bx+c$ grafiku. Programmas galvenajā logā attēlosim izvēlni, kura pagaidām sastāvēs no viena punkta - "`Grafiks`". Šajā izvēlnes punktā būs divi apakšpunkti "`Izveidot`" un "`Beigt darbu`". Izvēloties punktu "`Izveidot`", atvērsies dialoglogs, kurā būs jāievada `a`, `b` un `c` vērtības, pēc kā tiks uzzīmēts funkcijas grafiks, kurš tiks attēlots galvenajā logā izvietotā `Canvas` objektā.

Vispirms izveidosim klasi `ParametruDialoglogs`, kura veiks dialogloga attēlošanu un ļaus ievadīt `a`, `b` un `c` vērtības. Tā kā dialogloga rezultāts ir trīs skaitļi, jāievēro, ka klases rezultātu jāatgriež kā sarakstu - to redzēsim ekrānpogas "`Labi`" apstrādes funkcijā.

```
  class ParametruDialoglogs(Toplevel):
     def __init__(self, parent):
        Toplevel.__init__(self, parent)
        self.transient(parent)
        self.focus_set()
        self.title("Parametri")
        self.parent = parent
        self.result = None
        Ramis = Frame(self)
        Ramis.pack()
        # A ievade
        UzrakstsA = Label(Ramis, text="a:", fg="black")
        UzrakstsA.grid(row=0, column=0)
        self.A = Entry(Ramis)
        self.A.grid(row=0, column=1)
        # B ievade
        UzrakstsB = Label(Ramis, text="b:", fg="black")
        UzrakstsB.grid(row=1, column=0)
        self.B = Entry(Ramis)
        self.B.grid(row=1, column=1)
        # C ievade
        UzrakstsC = Label(Ramis, text="c:", fg="black")
        UzrakstsC.grid(row=2, column=0)
        self.C = Entry(Ramis)
        self.C.grid(row=2, column=1)
        # Ekranpogas
        LabiPoga = Button(Ramis, text="Labi", width=8, height=1,
command=self.labi)
        LabiPoga.grid(row=3, column=0)
        AtceltPoga = Button(Ramis, text="Atcelt", width=8, height=1,
command=self.atcelt)
        AtceltPoga.grid(row=3, column=1)
        self.wait_window(self)
     def labi(self, event=None):
        self.result = int(self.A.get()), int(self.B.get()), int(self.C.get())
        self.parent.focus_set()
        self.destroy()
     def atcelt(self, event=None):
        self.result = None
        self.parent.focus_set()
        self.destroy()
```

Kā redzams, `self.result` tiek piešķirtas vairākas vērtības, atdalot tās ar komatu, tā iegūstot sarakstu.

Tālāk izveidosim programmas "galveno" daļu, kura izveidos galveno logu, izvēlni un zīmēšanas virsmu. Tā kā zīmēšanas virsmā neatkarīgi no funkcijas grafika jāattēlo koordinātu asis, tās varam izveidot "pastāvīgas".

```
  # Galvenais logs
  Logs = Tk()
  Logs.title("Funkcijas grafiks")
  Logs.geometry("300x300")

  # Izvelne
  Izvelne = Menu(Logs)
  GrafiksIzvelne = Menu(Izvelne)
  Izvelne.add_cascade(label="Grafiks", menu=GrafiksIzvelne)
  GrafiksIzvelne.add_command(label="Izveidot", command=GrafiksIzveidot)
  GrafiksIzvelne.add_separator()
  GrafiksIzvelne.add_command(label="Beigt darbu", command=Logs.destroy)
  Logs.config(menu=Izvelne)

  # Virsma
  Virsma = Canvas(Logs, width=256, height=256, bg="black")
  Virsma.pack()

  # Koordinatu asis
  Virsma.delete("asis")
  Virsma.create_line(128, 1, 128, 256, tags="asis", fill="grey", arrow=FIRST)
  Virsma.create_line(1, 128, 256, 128, tags="asis", fill="grey", arrow=LAST)

  # Loga cikls
  Logs.mainloop()
```

Pati svarīgākā mūsu programmas daļa ir tā, kura "zīmē" funkcijas grafiku. Tā kā zīmēšanas virsmas izmēri ir 256x256 pikseļi, varam pieņemt, ka asu vērtības ir no -128 līdz +128. Tālākais jau ir vienkārši - mums jāizveido ciklu no -128 līdz 128, katrā solī jāaprēķina y vērtību, jāaprēķina "ekrāna" koordinātes (128-X un 128-Y) un "jāatliek" šo punktu uz ekrāna. "Atlikšanu" veiksim, zīmējot līniju no "iepriekšējā" punkta līdz "jaunajam". Lai iegūtais rezultāts būtu detalizētāks, samazināsim mērogu, nosakot, ka asu vērtības ir no -32 līdz +32. Tas nozīmē, ka katru iegūto `X` un `Y` vērtību jāreizina ar 4.

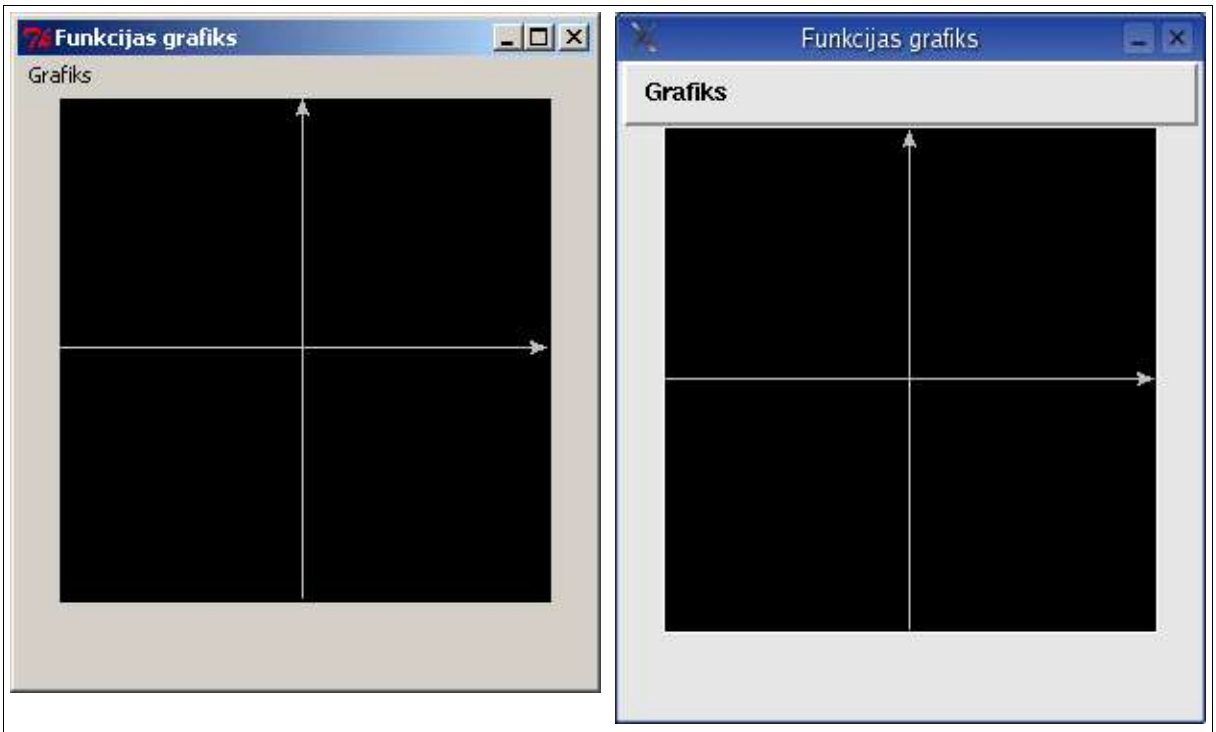
Kad esam vienojušies par algoritmu, ķersimies pie darba. Vispirms pieprasīsim lietotājam ievadīt parametrus `A`, `B` un `C` un tos "iegūsim":

```
  def GrafiksIzveidot():
      Dialogs = ParametruDialoglogs(Logs)
      if Dialogs.result:
          A = Dialogs.result[0]
          B = Dialogs.result[1]
          C = Dialogs.result[2]
  Tā kā dialogloga rezultāts ir saraksts, tad parametrus varam iegūt kā
saraksta elementus.
          Virsma.delete("grafiks")
          EkranaX = 0
          EkranaY = 0
          for X in xrange(-128, 128):
              Y = A*X*X+B*X+C
              JaunaisEkranaX = int(128-(X*4))
              JaunaisEkranaY = int(128-(Y*4))
              Virsma.create_line(EkranaX, EkranaY, JaunaisEkranaX+1,
JaunaisEkranaY+1, tags="grafiks", fill="green")
              EkranaX = JaunaisEkranaX
              EkranaY = JaunaisEkranaY
```

Tālāk jādzēš visus grafiskos primitīvus, kuru birka ir "grafiks". Tas nepieciešams, lai,

veidojot grafiku atkārtoti, tiktu nodzēsts iepriekšējais. Pēc tam noteiksim, ka sākotnējās "saglabātās" ekrāna `X` un `Y` koordinātes ir 0. Kā jau vienojāmies, tās nepieciešamas, lai iegūtais funkcijas grafiks būtu "gludāks". Tad veidojam ciklu no -128 līdz 128, aprēķinām `Y` un "jaunās" ekrāna koordinātes. Tā kā zīmēšanas virsmas koordinātes ir vienādas ar koordinātu ass III kvadrantu, tad, lai no grafika koordinātēm iegūtu ekrāna koordinātes, tās jāatņem no 128. Lai palielinātu mērogu 4 reizes, vienlaikus notiek reizināšana ar 4. Kad tas paveikts, tiek izveidota līnija, kuras galapunkti ir no "saglabātajām" jeb "iepriekšējām" koordinātēm līdz jaunajām. Visbeidzot, "jaunās" koordinātes tiek saglabātas mainīgajos, jo nākamajā cikla iterācijā tās būs "iepriekšējās".

Tas arī viss. Esam ieguvuši programmu, kura spēj uzzīmēt funkcijas $y=ax^2+bx+c$ grafiku. Programmu izpildot, tiek attēlots logs ar koordinātu asīm (11.attēls):

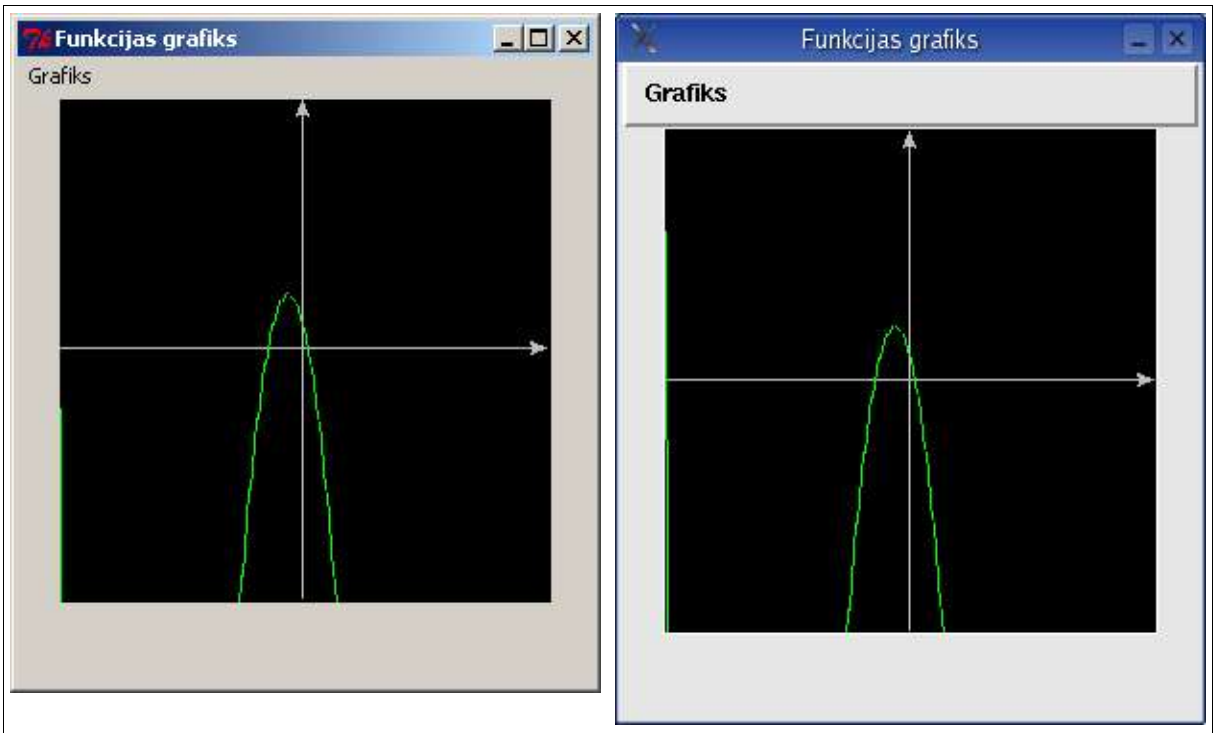


11.attēls: logs ar koordinātu asīm Microsoft Windows un GNU/Linux operētājsistēmās

Izvēloties izvēlnes punktu `Grafiks -> Izveidot`, atveras dialoglogs, kurā tiek piedāvāts ievadīt `A`, `B` un `C` (12.attēls):

12.attēls: parametru ievades dialoglogs Microsoft Windows un GNU/Linux operētājsistēmās

Ievadīsim, piemēram, `-1`, `4` un `3`. Nospiežot ekrānpogu "`Labi`", tiek izveidots funkcijas grafiks (13.attēls):



13.attēls: funkcijas grafiks Microsoft Windows un GNU/Linux operētājsistēmās

Šo programmiņu ir iespējams attīstīt tālāk - piedāvāt lietotājam mēroga izvēli, atbilstoši izvēlētajam mērogam atlikt uz koordinātu asīm vienības, veikt arī matemātisku kvadrātfunkcijas analīzi... Viss jūsu rokās - šobrīd ar jūsu zināšanām pilnīgi pietiek, lai to veiktu. Atliek krāt programmēšanas pieredzi - un to var veikt tikai programmējot.

Nobeigumam

Šīs grāmatiņas apjoms nebūt nav pietiekošs, lai aplūkotu visas `Tkinter` bibliotēkas iespējas, tomēr ir sniegts pietiekošs ieskats grafiskā lietotāja interfeisa veidošanā, lai turpmākās zināšanas būtu iespējams apgūt patstāvīgi, izmantojot *Python* un `Tkinter` dokumentāciju (`http://www.python.org/`). Neraugoties uz to, ir sniegts pietiekošs informācijas apjoms, lai lasītājs spētu veidot vienkāršas, tomēr pabeigtas un praktiskas *Python* programmas, izmantojot grafisko lietotāja interfeisu un bibliotēku `Tkinter`.

Nākamajā grāmatas "Programmēšanas valoda *Python* iesācējiem" daļā tiks aplūkota datorspēļu programmēšana, izmantojot bibliotēku `PyGame`. Šī bibliotēka ļauj veidot modernas multimēdiju programmas un spēles, izmantojot mūsdienīgu grafiku un skaņu.

Līdzīgi, kā iepriekšējās grāmatas daļās, atsauksmes lūdzu sūtīt uz e-pasta adresi `alvilsb@parks.lv`. Šo un citas grāmatas daļas iespējams lejuplādēt autora mājaslapā – `http://alvils.latvietis.com/`.

Vienlaikus brīdinu, ka nekonstruktīvas atsauksmes ("*šitā neder, vajag savādāk*", "*man nepatīk...*" u.c.) netiks izskatītas. Tiks izskatīta konstruktīva kritika, kā arī labprāt pieņemtas pozitīvas atsauksmes. :)