Graafisten käyttöliittymien alkeet

Pythonin ja tkinter-kirjaston avulla

Ari Suntioinen, TTY
Essi Isohanni, TTY
Jussi Kasurinen, LTY

Esipuhe

Tämä materiaali on tarkoitettu Tampereen teknillisen yliopiston opintojaksolle TIE-02100 Johdatus ohjelmointiin graafisten käyttöliittymien alkeiden opiskeluun Python-ohjelmointikielen ja tkinter-käyttöliittymäkirjaston avulla. Esitys pohjautuu raskaasti muokattuun Lappeenrannan teknillisen yliopiston vastaavaan Creative Commons 2.5 -lisenssin alaisuudessa julkaistuun materiaaliin [1], jonka lukuja 1–3 olemme hyödyntäneet sen kirjoittajan Jussi Kasurisen ystävällisellä myötävaikutuksella.

Materiaali on esitetty sellaisessa muodossa, että se ei ota kantaa käytettyyn ohjelmointiympäristöön. Kaikki esimerkit on kuitenkin testattu PyCharm-ympäristössä käyttäen Python-kielen versiota 3.x, eli samassa ympäristössä, jota kurssin työskentelyssä käytetään.

Sisältö

1 Tkinter-kirjaston perusteita

1.1 Taustaa: tkinter-kirjasto

Nykyisin useimmat ohjelmat julkaistaan käyttöjärjestelmille, joissa on mahdollisuus käyttää graafista käyttöliittymää. Tämän vuoksi käytännössä kaikki laajemmat ohjelmistot toimivat nimenomaan graafisella käyttöliittymällä, jossa valinnat ja vaihtoehdot esitetään valikkoina, valitsimina tai painikkeina, joiden avulla käyttäjä voi hiiren avulla valita, mitä haluaa tehdä. Monesti aloitteleva ohjelmoija luulee, että tällaisen käyttöliittymän toteuttamista varten tarvitaan jokin monimutkainen tai kallis kehitystyökalu, tai että se olisi erityisen vaikeaa. Ehkä joidenkin ohjelmointikielien yhteydessä tämä voi pitää paikkansa, mutta Pythonissa yksinkertaisten graafisten käyttöliittymien tekeminen onnistuu helposti Tkinter-kirjaston avulla.

Tkinter on alun perin Tcl-nimisen ohjelmointikielen käyttöliittymätyökalusta Tk tehty Python-laajennus, jonka avulla voimme luoda graafisen käyttöliittymän ohjelmallemme. Tkinter-kirjasto poikkeaa siinä mielessä ”perinteisistä” Python-kirjastoista, että sitä käyttävä Python-koodi eroaa jonkin verran tavallisesta Python-koodista ulkonäkönsä puolesta. Kuitenkin Python-kielen syntaksisäännöt sekä rakenne pysyvät edelleen samoina.

Seuraavissa kolmessa luvussa tutustumme siihen, kuinka yksinkertaisten käyttöliittymien ja niiden osien, kuten tekstikenttien, painikkeiden ja valintanappien tekeminen onnistuu. Tämä tietenkin tarkoittaa, että ensimmäiseksi tarvitsemme ikkunan, johon käyttöliittymän rakennamme.

1.2 Yksinkertainen graafinen käyttöliittymä

Esimerkkikoodi

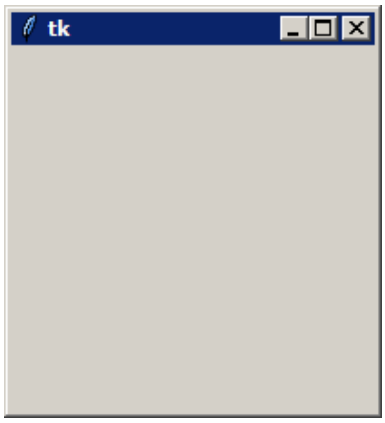
Yksinkertaisin mahdollinen Tkinter-käyttöliittymä saadaan toteutettua suorittamalla Pythonilla käskysarja:

```
1 from tkinter import *

2 def main():
3     pääikkuna = Tk()
4     pääikkuna.mainloop()

5 main()
```

Esimerkki 1: pelkkä pääikkuna

Esimerkkikoodin tuottama tulos

Kun esimerkin 1 koodi suoritetaan, saadaan tuloksena käyttöliittymäikkuna:



Kuva 1: pelkkä pääikkuna (Windows XP)

Kyseessä on tosin niin yksinkertainen käyttöliittymä, että käyttäjä ei voi tehdä sen avulla mitään.

Kuinka koodi toimii

Jotta ohjelmassa voisi muodostaa Tkinter-käyttöliittymiä, rivillä 1 on otettava käyttöön `tkinter`-kirjasto. Periaatteessa rivi 1 voisi olla myös tutumpi:

```
import tkinter
```

Mutta tällöin kaikkien kirjastosta käytettävien valmiiden funktioiden[1] eteen pitäisi aina liittää kirjaston nimi `tkinter`:

```
pääikkuna = tkinter.Tk()
```

mikä olisi vähänkään monimutkaisemman käyttöliittymän toteuttavassa ohjelmassa työlästä. Kun kirjastoa hyödynnetään `from ··· import *`-mekanismilla kuten esimerkissä 1, kirjaston funktioita voi käyttää suoraan niiden nimellä, ilman `tkinter.` -etuliitettä. Tämä sama idea toimii toki myös muiden kirjastojen kanssa.

Rivillä 3 muodostetaan käyttöliittymän *pää-* eli *juuri-ikkuna* (*main window, root window*), eli käytönnössä `Tk`-tyyppinen olio. Kaikissa graafisissa käyttöliittymissä on oltava pääikkuna, sillä muut käyttöliittymän *komponentit* (painonapit, tekstikentät jne.) sijoitellaan pääikkunaan, kuten myöhemmin

[1] Tarkasti ottaen `tkinter`-kirjaston tarjoamat palvelut on toteutettu luokkina (`class`) ja käsky `Tk()` tarkoittaa uuden `Tk`-tyyppisen olion luontia.

nähdään. Ensimmäisessä esimerkkiohjelmassa ei kuitenkaan ole mitään muuta kuin tyhjä pääikkuna, joten se on hiukan tylsä eikä sillä voi tehdä mitään hyödyllistä.

Rivillä 4 käyttöliittymä käynnistetään, mikä tapahtuu kohdistamalla pääikkunaan operaatio[1] `mainloop`. Jos tätä ei tehdä, pääikkuna ei ilmesty näkyviin laisinkaan. Yleisesti käyttöliittymien ohjelmoinnin terminologiassa `mainloop`:ia kutsutaan *tapahtumasilmukaksi* (*event loop*).

Huomaa sellainen merkittävä yksityiskohta, että `mainloop`-metodista ei palata, ennen kuin käyttöliittymä suljetaan tavalla tai toisella. Tätä asiaa tutkitaan hieman syvällisemmin luvussa 2.1.

1.3 Käyttöliittymän toteutus luokkana

Periaatteessa kaikki käyttöliittymät voitaisiin toteuttaa normaalien funktioiden ja muuttujien avulla, kuten esimerkissä 1 tehtiin. Tämä johtaa kuitenkin hiukankaan monimutkaisempien käyttöliittymien kanssa ohjelmointityylillisiin ongelmiin. Ongelmat juontuvat siitä, että ainoa tapa, jolla käyttöliittymän eri toiminnat (siis funktiot) voivat jakaa tietoa keskenään, ovat globaalit muuttujat, joiden käyttö ainakin suuressa mittakaavassa on paheksuttavaa. Syy globaalien muuttujien tarpeelle selviää luvussa 2.1.

Globaaleihin muuttujiin liittyvät ongelmat voidaan pitkälti välttää, jos toteutetaan käyttöliittymä luokan (`class`) avulla. Lisäksi luokkien käytöllä saavutetaan kehittyneemmässä käyttöliittymien ohjelmonnissa etuja,[2] jotka ovat valitettavasti tämän kurssin aihepiirien ulkopuolella.

Esimerkkikoodi

Seuraavassa listauksessa esimerkin 1 alkeellinen käyttöliittymä on toteutettu luokkana.

---

[1] Eli metodi, jos ja kun halutaan käyttää luokkien ja olioiden terminologiaa, sillä onhan `Tk` luokka ja `pääikkuna` puolestaan `Tk`-tyyppinen olio.

[2] Koska `tkinter`:in valmiit *komponentit* on toteutettu luokkina, niistä voidaan *periyttää* uusia tiettyyn käyttötarkoitukseen paremmin soveltuvia versioita. *Periyttäminen* on olio-ohjelmoinnissa käytetty tekniikka, jolla olemassa olevasta luokasta voidaan muodostaa muokattuja versioita.

```
1 from tkinter import *

2 class Käyttöliittymä:
3     def __init__(self):
4         self.__pääikkuna = Tk()
5         self.__pääikkuna.mainloop()

6 def main():
7     käli = Käyttöliittymä()

8 main()
```

Esimerkki 2: pelkkä pääikkuna luokan avulla toteutettuna

Esimerkkikoodin tuottama tulos

Esimerkki 2 tuottaa täsmälleen saman pääikkunan, kuin mitä kuvassa 1 oli havainnollistettu esimerkistä 1.

Kuinka koodi toimii

Jos ei huomioida luokkamäärittelyn syntaksiin liittyviä lisärivejä, huomataan, kuinka koodirivit 4–5 ovat lähes samat kuin esimerkin 1 rivit 3–4. Pääikkunaoliota ei nyt kuitenkaan sijoiteta minkään funktion paikalliseen muuttujaan, vaan rivillä 4 se menee talteen muodostettavana olevan `Käyttöliittymä`-tyyppisen olion sisäiseen muuttujaan (*attribuuttiin*) `self.__pääikkuna`. Rivillä 5 käyttöliittymä käynnistetään normaaliin tapaan kohdistamalla pääikkunaolioon `self.__pääikkuna` metodi `mainloop`.

Uuden käyttöliittymän muodostaminen tarkoittaa tässä muutetussa versiossa `Käyttöliittymä`-tyyppisen olion muodostamista, mikä tapahtuu konkreettisesti pääohjelmassa rivillä 7. Tämän jälkeen luokan `__init__`-metodi huolehtii kulissien takana automaattisesti alustustoimenpiteistä ja `mainloop`:in käynnistämisestä.

1.4 Komponenttien lisääminen

Kuten esimerkistä 1 havaittiin, käyttöliittymä ei ole erityisen hyödyllinen, mikäli se muodostuu pelkästä pääikkunasta. Tämän vuoksi pääikkunaan halutaan aina lisätä *komponentteja* (*widgets*), joiden avulla käyttäjä voi harrastaa interaktiota käyttöliittymän kanssa. Yksinkertaisimpia käyttöliittymäkomponentteja ovat erilaiset *painonapit* (`Button`), *tekstikentät* (`Label`) ja *syötekentät* (`Entry`).

Esimerkkikoodi

Lisätään aiempien esimerkkien alkeelliseen pääikkunaan lyhyt teksti.

```
1  from tkinter import *

2  class Käyttöliittymä:
3      def __init__(self):
4          self.__pääikkuna = Tk()

5          self.__tekstikenttä = Label(self.__pääikkuna, text="Hello World!")
6          self.__tekstikenttä.pack()

7          self.__pääikkuna.mainloop()

8  def main():
9      käli = Käyttöliittymä()

10 main()
```

Esimerkki 3: `Label`-tyyppinen komponentti

Esimerkkikoodin tuottama tulos

Kun esimerkin 3 koodi suoritetaan, ilmestyy näytölle ikkuna:

Kuva 2: `Label`-komponentti (Linux/GNOME)

Edelleenkään käyttäjä ei voi tehdä käyttöliittymässä muuta kuin sulkea sen ×-nappia painamalla, koska `Label`-komponentti ei ole interaktiivinen, vaan sen avulla voidaan pelkästään näyttää staattista tekstiä osana käyttöliittymää.

Kuinka koodi toimii

Esimerkissä on käytetty `Label`-tyyppistä *komponenttia*, jonka pääasiallinen käyttötarkoitus on staattisen tekstin esittäminen käyttöliittymässä. Kun tutkitaan esimerkin 3 koodilistausta, huomataan sen olevan rivejä 5–6 lukuunottamatta identtinen esimerkin 2 listaukseen verrattuna. Tämä johtuu tietysti siitä, että kaikissa kaäyttöliittymissä tarvittavan pääikkunan alustusoperaatio ja *tapahtumasilmukan* `mainloop` käynnistäminen etenee aina

samaa rataa. Lisärivit 5–6 liittyvät siihen, kuinka `Label`-komponentti saadaan lisättyä osaksi käyttöliittymää.

Rivillä 5 luodaan `Label`-tyyppinen *käyttöliittymäkomponentti* ja talletetaan se luokan attribuuttiin `self.__tekstikenttä`. `Label` on itsessäänkin luokka, joten uusia sen tyyppisiä *komponentteja* luodaan käyttämällä tyypin nimeä `Label` ikään kuin se olisi funktio. *Komponentin* tyypistä riippuen sitä luotaessa on myös annettava parametreja, jotka asettavat komponentin arvon halutuksi. Esimerkiksi `Label`:in tapauksessa parametri `text` määrää, minkä tekstin luotava `Label` näyttää käyttöliittymässä. Kaikilla komponenttityypeillä (`Label`, `Button`, `Entry` jne.) **ensimmäisen parametrin on oltava aina** `Tk`-tyyppinen[1] pääikkunaolio, jonka sisään uusi *komponentti* halutaan sijoittaa. Tätä kutsutaan usein komponentin *vanhemmaksi* (*parent*). Kun käyttöliittymä aikanaan ilmestyy näytölle, uusi komponentti sijaitsee *vanhemman* määräämän alueen sisällä. Esimerkissä 3 siis `Label`-komponentin teksti `"Hello World!"` pysyy pääikkunan määräämällä alueella. Yleensä pääikkunan koko myös mukautuu siten, että kaikki sen *lapset* (*child*) ovat kokonaisuudessaan näkyvissä sen sisällä.

Voisi kuvitella, että käyttöliittymä on nyt valmis, mutta käytönnössä haluttu teksti (`Label`-komponentti) ei ilmesty käyttöliittymään, mikäli rivi 6 puuttuu. Käyttöliittymän on nimittäin tiedettävä, kuinka komponentti halutaan sijoittaa *vanhempansa* sisällä (keskelle, vasempaan laitaan, oikeaan yläkulmaan jne.), ennen kuin se osaa näyttää komponentin. Sijoittelu saadaan aikaan *geometrianhallintaoperaatioilla* (*geometry manager*). Esimerkissä 3 käytetty `pack` on niistä yksinkertaisin ja sen avulla komponentteja voidaan sijoitella käyttöliittymään rinnakkain tai alekkain. Jokaiseen käyttöliittymän komponenttiin kohdistettava jokin *geometrianhallintaoperaatio*, mikäli sen halutaan näkyvän käyttöliittymässä. Näin on toimittava silloinkin, kun käyttöliittymässä on vain yksi komponentti kuten esimerkissä 3.

Geometrianhallintaoperaatioille voi antaa parametrejä, jotka ohjaavat sitä, kuinka komponentti sijoitetaan *vanhempansa* sisään. Esimerkiksi `pack`:in toimintaa voidaan säätää määräämällä, mitä *vanhemman* laitaa lähimmäksi komponentti halutaan sijoittaa. Voit kokeilla tätä itse muokkaamalla esimerkin 3 rivin 6 paikalle seuraavia variaatioita:

```
self.__tekstikenttä.pack(side=LEFT)
self.__tekstikenttä.pack(side=RIGHT)
self.__tekstikenttä.pack(side=TOP)
self.__tekstikenttä.pack(side=BOTTOM)
```

Muuta tämän jälkeen pääikkunan kokoa ja seuraa, kuinka `Label`-komponentti käyttäytyy. Jos `pack`:ille ei anneta ollenkaan parametreja, sekä käyttäytyy

[1] Myös `Frame`-tyyppinen komponentti käy vahemmaksi kaikille muille komponenteille. Itse asiassa on olemassa joitain muitakin mahdollisuuksia, mutta ohitetaan ne kaikki tässä yhteydessä.

samoin, kuin jos olisi annettu `pack(side=TOP)`. Jos yksinkertaiseen käyttöliittymään halutaan asettaa useita komponentteja rinnakkain, niin kaikkiin kannattaa kohdistaa `pack(side=LEFT)`-operaatio. Jos taas komponentteja halutaan alekkain, operaatio `pack(side=TOP)` tuottaa toivotun tuloksen (tämä on myös oletustoiminnallisuus).

Jos haluttu käyttöliittymä on monimutkainen ja koostuu useista riveistä ja sarakkeista, geometrian hallinta `pack`:in avulla on yleensä työlästä. Tällaisissa tilanteissa `grid`-operaatio osoittautuu yleensä käyttökelpoisemmaksi. Sen avulla komponentteja voidaan sijoittaa virtuaalisiin matriisin halutulle riville ja haluttuun sarakkeeseen. Tähän palataan myöhemmissä esimerkeissä, joissa käyttöliittymä muodostuu useista komponenteista, jotka halutaan sijoitella hyvin täsmällisesti toistensa suhteen.

Komponenttien ja geometrianhallintaoperaatioiden parametreista

Kuten edellä on käynyt ilmi, sekä käyttöliittymäkomponetteja luotaessa että niiden sijoittelua ohjaaville geometrianhallintaoperaatioille voidaan käyttää erilaisia parametreja, jotka vaikuttavat siihen, miten komponentti lopulta esitetään käyttöliittymässä. Kaikkia mahdollisia vaihtoehtoja ei ole järkevää käydä läpi tämän monisteen kaltaisessa alkeisoppaassa. Niitä tulee kuitenkin jonkin verran esiin myöhemmissä esimerkeissä, joissa niiden tarkoitus on selitetty osana esimerkkiä. Asiasta kiinnostuneille verkosta on löydettävissä suuret määrät materiaalia ja esimerkkejä Tkinterin käytöstä. Yksi hyvä ja kattava (joskin jossain määrin keskeneräinen) käsikirjamainen esitys on effbot.org:in An Introduction To Tkinter [2].

Seuraavassa kuitenkin lyhyt esimerkki `Label`:in ja `pack`:in parametrien käytöstä: jos esimerkin 3 rivit 5–6 muutetaan seuraaviksi ohjelman pysyessä muuten samana:

```
self.__tekstikenttä = Label(self.__pääikkuna, text="Hello World!",
                            background="green", foreground="red",
                            padx=30, pady=10,
                            relief=RAISED, borderwidth=5)
self.__tekstikenttä.pack(expand=True, fill=BOTH)
```

näyttää syntyvä käyttöliittymä seuraavalta:

Kuva 3: koristeltu `Label` (Linux/GNOME)

Jos saadun pääikkunan kokoa nyt muutetaan hiirellä, `Label`-komponentti kasvaa niin, että se täyttää aina koko pääikkunan.

Kuva 4: koristeltu ja hiirellä suurennettu `Label` (Linux/GNOME)

2 Interaktiivisuutta painikkeilla

2.1 Taustaa: tapahtumapohjainen ohjelmointi

Käyttöliittymien ohjelmointi eroaa perinteisestä ohjelmointitavasta siksi, että käyttöliittymän kontrolli sijaitsee *tapahtumasilmukassa* (`mainloop`), eikä ohjelmoija kutsu käyttöliittymänsä funktioita itse. Käytännössä tämä tarkoittaa sitä, että kun käyttöliittymän looginen rakenne on valmis (pääikkuna ja tarvittavat käyttöliittymän komponentit alustettu), ohjelmoija käynnistää *tapahtumasilmukan*, joka alkaa kontrolloida kaikkea käyttöliittymän toimintaan liittyvää (hiiren napin painallukset, käyttöliittymän päivitys, ikkunan koon muutokset jne.).

Jos ohjelmoija esimerkiksi haluaa, että tiettyä funktiota kutsutaan aina, kun hiirellä painetaan tiettyä käyttöliittymän painonappia, hän määrittelee käyttöliittymää luodessaan yhteyden: kun painonappi X aktivoidaan hiirellä, kutsu funktiota Y. Varsinainen kutsu tapahtuu automaattisesti kulissien takana *tapahtumasilmukassa*, joka huomaa, että hiiren nappi painettin pohjaan kursorin ollessa napin X päällä, ja kutsuu käyttäjän määräämää funktiota Y. Tällaisia funktioita kutsutaan *tapahtumakäsittelijöiksi* (*callback*).

Tapahtumasilmukan toimintaa voidaan havainnollistaa hyvin yleisellä tasolla seuraavan kaltaisella algoritmilla:

```
while True:
   (1) Odota, että käyttäjä tekee jotain, esimerkiksi
       aktivoi hiirellä painonapin.
   (2) Reagoi ohjelmoijan määräämällä tavalla käyttäjän
       aiheuttamaan ärsykkeeseen (eli napin painallukseen).
```

Ohjelmointitapaa, joka perustuu siihen, että ohjelma reagoi ulkoisiin ärsykkeisiin *tapahtumakäsittelijöiden* avulla, kutsutaan *tapahtumapohjaiseksi ohjelmoinniksi* (*event driven programming*). Käytännössä kaikki graafiset käyttöliittymät toimivat tällä tavoin.

Koska *tapahtumakäsittelyfunktioiden* kutsu tapahtuu automaattisesti jossain *tapahtumasilmukan* (`mainloop`) syövereissä, ei ohjelmoija pysty vaikuttamaan siihen, mitä funktioille välitetään parametreina tai mitä mitä niiden paluuarvoille tehdään, jos mitään. Tämän vuoksi on kätevää, jos käyttöliittymä on toteutettu luokkana: *tapahtumankäsittelijä* saa tarvitsemansa lähtötiedot käyttöliittymäolion attribuuteista, ja jos se tuottaa tuloksenaan jotain hyödyllistä tietoa, sekin voidaan tallentaa attribuutteihin myöhempää käyttöä varten.

2.2 Painonappikomponentti

Esimerkkikoodi

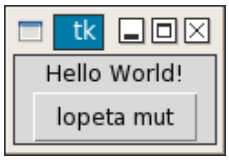
Lisätään esimerkin 3 kolme käyttöliittymään painonappi, jota hiirellä painamalla ohjelman suoritus saadaan päättymään.

```
1  from tkinter import *

2  class Käyttöliittymä:
3      def __init__(self):
4          self.__pääikkuna = Tk()

5          self.__tekstikenttä = Label(self.__pääikkuna, text="Hello World!")
6          self.__tekstikenttä.pack()

7          self.__lopetusnappi = Button(self.__pääikkuna, text="lopeta mut",
8                                       command=self.lopeta)
9          self.__lopetusnappi.pack()

10         self.__pääikkuna.mainloop()

11     def lopeta(self):
12         self.__pääikkuna.destroy()


13 def main():
14     käli = Käyttöliittymä()

15 main()
```

Esimerkki 4: lopetusnapin toteuttaminen käyttöliittymään

Esimerkkikoodin tuottama tulos

Kun ohjelmakoodi suoritetaan, on tuloksena seuraavan näköinen käyttöliittymä:



Kuva 5: painonapillinen käyttöliittymä (Linux/GNOME)

Jos painaa hiirellä `lopeta mut` -nappia, käyttöliittymä sulkeutuu ja ohjelman suoritus päättyy.

Kuinka koodi toimii

Esimerkin 4 ohjelmakoodi pohjautuu edelleen samaan runkoon kuin edelliset esimerkit. Siihen on lisätty painonapin (`Button`) vaatimat toiminnallisuudet riveille 7–9 ja 11–12, mutta muutoin kaikki on säilynyt ennallaan.

Rivillä 7 muodostetaan uusi painonappikomponentti. Kaksi ensimmäistä parametria ovat logiikaltaan samat kuin aiemmin `Label`-komponentin yhteydessä: komponentin vanhemman ja tekstisisällön määrittely. Kolmas parametri `command` rivillä 8 on `tapahtumakäsittelijä`, jota `tapahtumasilmukan` halutaan kutsuvan, kun painonappi aktivoidaan hiirellä: `command`-parametrin arvona on siis aina oltava jokin funktio. Se voi olla ja on tällä kurssilla lähes aina[1] jokin käyttöliittymäolion oma metodi eli siis muotoa `self.`*`jokin_metodin_nimi`*. On tärkeää ymmärtää, että *tapahtumakäsittelijää* ei kutsuta heti sillä hetkellä, kun `Button`-komponenttia ollaan luomassa, vaan kutsu tapahtuu myöhemmin `mainloop`:ista, kun nappia painetaan hiirellä. Tämä selittää myös sen, miksi *tapahtumakäsittelijän* nimen perässä ei ole sulkeita:

```
# EI NÄIN!!!  TÄMÄ ON KERTAKAIKKISEN VÄÄRIN!!!       ↓
self.__lopetusnappi = Button(···, command=self.lopeta())
```

Jos Pythonissa kirjoittaa kaarisulkeet `()` funktion nimen perään, funktiota kutsutaan heti, kun ohjelman suoritus etenee kyseiseen kohtaan. Tämä ei missään tapauksessa ole se, mitä `Button`-komponentin määrittelykohdassa halutaan tapahtuvan.

Rivillä 9 painonapille on suoritettava `pack`-operaatio, jotta se ilmestyisi näkyviin käyttöliittymässä. Kuten esimerkin 3 yhteydessä sivulla 9 todettiin, kaikille komponenteille on määriteltävä, minne ne halutaan käyttöliittymässä sijoittaa.

Riveillä 11–12 on määritelty käyttöliittymäluokalle metodi, jota `mainloop` voi kutsua, kun `"lopeta mut"`-nappia painetaan. Huomaa, että kyseessä on aivan tavallinen metodifunktio, jonka rungossa voi olla mitä tahansa Pythön-käskyjä, ja myös kaikki olion attribuutit ovat funktion rungossa normaalisti käytettävissä. Parametrejä *tapahtumakäsittelyyn* tarkoitetuilla metodeilla ei kuitenkaan `self`:in lisäksi ole.

Pääikkunaolioon voidaan kohdistaa metodi `destroy`, kuten rivillä 12 on tehty. Tämä saa aikaan sen, että pääikkuna sulkee itsensä ja `mainloop`-metodista palataan riville 10, joka on luokan `__init__`-metodin viimeinen käsky. Suoritus jatkuu siis pääohjelman riviltä 14, jossa `Käyttöliittymä`-luokan `__init__`:iä kutsuttiin kulissien takana olion alustamiseksi. Koska rivi 14 on pääohjelman viimeinen rivi, ohjelman suoritus päättyy.

---

[1] Joissain viikkoharjoitusten ∗-tehtävissä tapahtumakäsittelijänä saatetaan käyttää nk. `lambda`-funktiota, mutta yleisesti ottaen se on ylikurssimateriaalia.

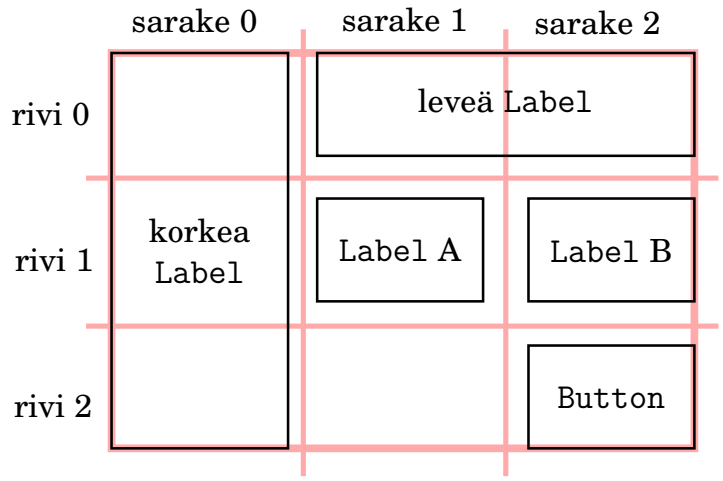
3 Monimutkaisemmat käyttöliittymät

3.1 Taustaa: geometrianhallinta `grid`-mekanismin avulla

Pariatteessa kaikkien toteutettavien käyttöliittymien *geometrianhallinta* voidaan toteuttaa `pack`-mekanismilla. Käytännössä tämä kuitenkin muodostuu monimutkaiseksi, jos *komponentteja* on paljon ja ne halutaan asetella siististi riveille ja sarakkeisiin. Tällaisissa tilanteissa `grid`-operaation käyttäminen saattaa osoittautua `pack`:iä selkeämmäksi ja intuitiivisemmaksi.

Toimintaperiaatteeltaan `grid`-geometrianhallintamekanismi käyttäytyy siten, että jokaisesta käyttöliittymään lisättävästä komponentista kerrotaan virtuaalisen ruudukon (eli matriisin) rivi ja sarake, johon kyseinen komponentti halutaan käyttöliittymässä sijoittaa. Mekanismi on joustava myös siinä suhteessa, että yksi komponentti voidaan levittää useamman kuin yhden rivin ja/tai sarakkeen alueelle.

Jos käyttöliittymä on niin monimutkainen, että sen toteuttamisessa koetaan `grid`-mekanismi tarpeelliseksi, sitä kannattaa hahmotella etukäteen kynällä ja paperilla. Tässä luvussa toteutetaan esimerkkinä käyttöliittymä, jonka hahmotelma on seuraavan kuvion kaltainen.



Kuva 6: hahmotelma 3×3 -ruudukkoon rakennetusta käyttöliittymästä

3.2 Esimerkki komponentien sijoittelusta `grid`:in avulla

Esimerkkikoodi

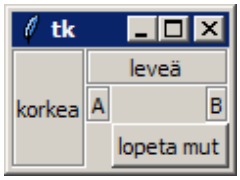
Kun kuvan 6 hahmotelma toteutetaan Tkinter-käyttöliittymänä, syntyy seuraavan näköinen koodi:

```
1  from tkinter import *

2  class Käyttöliittymä:
3      def __init__(self):
4          self.__pääikkuna = Tk()

5          self.__labelA = Label(self.__pääikkuna, text="A",
6                                borderwidth=2, relief=GROOVE)
7          self.__labelA.grid(row=1, column=1)

8          self.__labelB = Label(self.__pääikkuna, text="B",
9                                borderwidth=2, relief=GROOVE)
10         self.__labelB.grid(row=1, column=2, sticky=E)

11         self.__korkealabel = Label(self.__pääikkuna, text="korkea",
12                               borderwidth=2, relief=GROOVE)
13         self.__korkealabel.grid(row=0, column=0, rowspan=3, sticky=N+S)

14         self.__leveälabel = Label(self.__pääikkuna, text="leveä",
15                               borderwidth=2, relief=GROOVE)
16         self.__leveälabel.grid(row=0, column=1, columnspan=2, sticky=E+W)

17         self.__lopetusnappi = Button(self.__pääikkuna, text="lopeta mut",
18                                      command=self.lopeta)
19         self.__lopetusnappi.grid(row=2, column=2)

20         self.__pääikkuna.mainloop()


21     def lopeta(self):
22         self.__pääikkuna.destroy()


23 def main():
24     käli = Käyttöliittymä()

25 main()
```

Esimerkki 5: Käyttöliitymän rakentaminen grid-mekanismin avulla

Esimerkkikoodin tuottama tulos

Esimerkkiohjelman 5 suorittaminen tuottaa seuraavan hiukan mielenkiintoisemman käyttöliittymän:



Kuva 7: Esimerkin 5 `grid`-käyttöliittymä (Windows XP)

Kuinka koodi toimii

Vaikka esimerkkiohjelma 5 on pidempi kuin aiemmat esimerkit, se on itse asiassa melko suoraviivainen, jos on ymmärtänyt kuvion 6 idean virtuaalisesta matriisista, johon käyttöliittymän komponentteja sijoitetaan `grid`:in avulla.

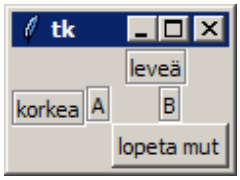
Riveillä 6, 9, 12 ja 15 `Label`-komponenteille on lisätty kahden pikselin levyinen kehys ihan vain sen vuoksi, että kuvasta 7 olisi helpompi nähdä, kuinka `grid` on sijoitellut komponentit pääikkunaan. Lisäparametreilla `borderwidth` ja `relief` ei siis muuten ole mitään tekemistä `grid`:iin perustuvan geometrianhallinnan kanssa.

Kaikkein yksinkertaisin `grid`:in käyttötapa käy ilmi riviltä 19, jossa `row`- ja `column`-parametrien avulla määrätään, että `self.__lopetusnappi` sijoitetaan matriisin riville 2 ja sarakkeeseen 2.

Rivillä 13 lisäparametri `rowspan=3` kertoo, että vaikka `self.__korkealabel` sijoitetaan riville 0 ja sarakkeeseen 0, sen on todellisuudessa kolmen komponenttirivin korkuinen. Aivan vastaavasti rivillä 16 käytetään `columnspan=2` -parametria levittämään `self.__leveälabel` kahden sarakkeen levyiseksi.

Lisäparametri `sticky` riveillä 10, 13 ja 16 voi saada arvokseen minkä tahansa `+`-operaattorilla muodostetun yhdistelmän arvoista `N`, `E`, `S`, `W`, `NE`, `SE`, `SW`, `NW`, eli siis englanninkielisistä ilmansuuntien lyhenteistä (North, NorthEast jne.). Käsiteltävänä olevan komponentin reunat kiinnittyvät pysyvästi kaikkiin niihin `grid`-matriisiin rivin ja sarakkeen laitoihin, jotka `sticky`-parametrissa on mukana. Konkreettisesti siis esimerkissä 5 ja kuvassa 7 `__labelB` on kiinni sarakkeensa oikeassa laidassa (`E` eli East), `__leveälabel` on kiinni sekä sarakkeen 1 vasemmassa laidassa (`W`) että sarakkeen 2 oikeassa laidassa (`E`) jne.

Lisäksi sellainen erityishuomio, että jos `sticky`-parametrille ei anneta arvoa ollenkaan, `grid` sijoittaa komponentin keskelle matriisin solua sekä vaaka- että pystysuunnassa. Jos parametrissa taas on mukana vastailmansuuntia kuten `__leveälabel`:in ja `__korkealabel`:in tapauksessa (vaakasuunassa `E+W` ja pystysuunnassa `N+S`), komponentti venytetään täyttämään koko matriisin solu kyseisessä suunnassa. Tätä toimintaa voidaan kokeilla muokkaamalla esimerkin 5 koodia siten, että kaikki `sticky`-parametrit poistetaan, mutta pidetään kaikki muu ennallaan. Tuloksena saadaan seuraavan näköinen käyttöliittymä:



Kuva 8: esimerkin 5 käyttöliittymä ilman `sticky`-parametreja (Windows XP)

Sekä `__leveälabel` että `__korkealabel` ovat nyt juuri sen kokoisia, mitä ne luonnostaan ovat, ja `__labelB` on sijoittu keskelle omaa matriisin soluaan. Kannattaa myös panna merkille, että `__leveälabel` vie edelleen tilaa kaksi saraketta ja `__korkealabel` kolme riviä, mutta niitä ei ole venytetty täyttämään koko tilaa, vaan ne on sijoitettu luonnollisen kokoisina sekä vaaka- että pystysuunnassa keskellä omaa aluettaan.

Yleisiä huomioita käyttöliittymän muodostamisesta

Esimerkin 5 käyttöliittymä on jo riittävän monimutkainen, että sen toteutuksesta voi päätellä yleisemmän periaatteen siitä, kuinka käyttöliittymän suunnittelu lopulta muodostuu kolmesta osasta:

- tarvittavien komponenttiolioiden muodostaminen (`Label`, `Button` jne.),
- komponenttien sijoittelu oikeille paikoille geometrianhallintatyökaluilla (`pack` tai `grid`) ja
- *tapahtumakäsittelijöiden* määrittely tarvittaville komponenteille.

Kaikki edellä esitellyt osat eivät välttämättä ole erillisiä. Esimerkiksi *tapahtumakäsittelijän* määrittely yhdistetään usein komponentin muodostamisen kanssa `command`-parametrin avulla.

Lopuksi vielä pari nyrkkisääntöä:

- Tkinter-käyttöliittymässä ei suositella käytettäväksi samanaikaisesti sekä `pack`- että `grid`-mekanismeja. Tuloksena on nimittäin usein käyttöliittymä, joka jumiutuu täysin.
- Jos käyttöliittymässä ei ole kuin muutama komponentti, jotka voidaan sijoittaa päällekkäin tai rinnakkain, se saadaan yleensä riittävän hyvin aikaan `pack`-operaatiolla. Monipuolisempi `grid` kannattaa varata tilanteisiin, jossa komponentteja on paljon ja/tai niiden asettelu on tehtävä hyvin tarkasti, jotta saadaan juuri sen näköinen käyttöliittymä kuin oli visioitu.

4 Syötteiden vastaanottaminen

4.1 Taustaa: `Entry`-komponentti

Tässä luvussa keskitytään pohtimaan sitä, kuinka käyttöliittymä voi vastaanottaa käyttäjältä syötteitä. Syötteillä tarkoitaan tässä yhteydessä tietoa, joka voidaan esittää merkkijonona.[1]

Tkinter-kirjastossa on syötteiden lukemiseen `Entry`-tyyppinen komponentti, jonka avulla käyttöliittymään voidaan luoda nk. *syötekenttiä*, joihin käyttäjä voi sitten syöttää näppäimistöltä haluamansa merkkijonon. Tkinter tarjoaa myös mekanismeja, joilla `Entry`-komponentin voi tyhjentää tai siihen voi lisätä tekstiä. Näitä voi käyttää esimerkiksi käyttöliittymän tuottamien tulosten esittämiseen.

4.2 Esimerkki: kellonaikamuunnin

Toteutetaan käyttöliittymä, jonka avulla käyttäjä voi muuttaa mannermaisessa muodossa olevia kellonaikoja (esim. 21.47) "imperialistiseen" muotoon (esim. 9:47 PM). Esimerkissä on myös ensimmäistä kertaa toteutettu *tapahtumakäsittelijään* `muunna` jonkin verran monimutkaisempaa toiminnallisuutta kuin aiemmissa esimerkeissä on ollut. Se ei kuitenkaan ole mitenkään yllätyksellistä, sillä kuten jossain vaiheessa todettiin, *tapahtumakäsittelijät* ovat normaaleja luokan metodeja, joihin voi kirjoittaa mitä tahansa tarvittavaa Python-koodia.

Tilan säästämiseksi esimerkkiohjelmasta on jätetty virhetarkistukset ja pääohjelma pois.

[1] Väärinkäsitysten välttämiseksi kannattaa korostaa, että sekä kokonaisluvut (`int`) että desimaaliluvut (`float`) voidaan esittää myös merkkijonoina, joista ne on tarpeen vaatiessa muunnettavissa lukuarvoiksi `int()`- ja `float()`-funktioiden avulla esimerkiksi laskutoimistusten suorittamaiseksi (vrt. esimerkki 6 rivi 21).

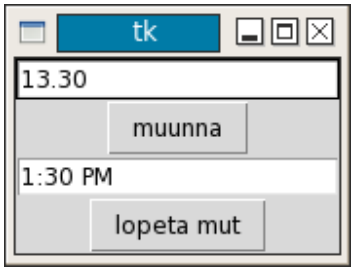
Esimerkkikoodi

```
from tkinter import *

class Käyttöliittymä:
    def __init__(self):
        self.__pääikkuna = Tk()

        self.__lähtöarvo = Entry()
        self.__lähtöarvo.pack()

        self.__muunnosnappi = Button(self.__pääikkuna, text="muunna",
                                     command=self.muunna)
        self.__muunnosnappi.pack()

        self.__tulosarvo = Entry()
        self.__tulosarvo.pack()

        self.__lopetusnappi = Button(self.__pääikkuna, text="lopeta mut",
                                     command=self.lopeta)
        self.__lopetusnappi.pack()

        self.__pääikkuna.mainloop()

    def muunna(self):
        aika_mantereella = self.__lähtöarvo.get()

        # Tilan säästämiseksi kellonajalle tehtävävät
        # virhetarkistukset poistettu tästä kohdasta.

        (tunnit, minuutit) = aika_mantereella.split(".")
        tunnit = int(tunnit)

        if tunnit < 1:           # 00.00-00.59 -> 12.00 AM-12.59 AM
            tunnit += 12
            pääte = "AM"
        elif tunnit < 12:        # 01.00-11.59 ->  1.00 AM-11.59 AM
            pääte = "AM"
        elif 12 <= tunnit < 13:  # 12.00-12.59 -> 12.00 PM-12.59 PM
            pääte = "PM"
        else:                    # 13.00-23.59 ->  1.00 PM-11.59 PM
            tunnit -= 12
            pääte = "PM"

        tulos = "{:d}:{:s} {:s}".format(tunnit, minuutit, pääte)

        self.__tulosarvo.delete(0, END)
        self.__tulosarvo.insert(0, tulos)

    def lopeta(self):
        self.__pääikkuna.destroy()
```

Esimerkki 6: `Entry`-komponentin lukeminen ja asettaminen

Esimerkkikoodin tuottama tulos

Kun esimerkin 6 ohjelmakoodi suoritetaan (`main` muistettava lisättävä), saadaan kuvan 9 käyttöliittymä, jossa käyttäjä voi kirjoittaa ylempään `Entry`-syötekenttään mannermaisessa muodossa olevan kellonajan. Kun hiirellä painetaan `muunna`-nappia, tulos ilmestyy 12-tuntiseen muotoon muunnettuna alempaan kenttään.



Kuva 9: Kellonaikaohjelman käyttöliittymä (Linux/GNOME)

Kuinka koodi toimii

Vaikka esimerkin ohjelmakoodi vaikuttaa melko pitkältä, siinä ei ole uuttaa käyttöliittymäkomponentteihin liittyvää asiaa erityisen paljon. Suurin yksittäinen osa koodia on nimittäin rivien 18–32 muodostama kellonajan muuttamiseen liittyvä laskulogiikka, joka on aivan tavallista käyttöliittymän käsittelyyn liittymätöntä Python-koodia.

Riveillä 5–6 ja 10–11 muodostetaan tarvittavat kaksi `Entry`-komponenttia, joista ensimmäistä (`__lähtöarvo`) käytetään käyttäjän syöttämän kellonajan lukemiseen ja jälkimmäistä (`__tulosarvo`) lasketun 12-tuntisen tulosarvon näyttämiseen. Kaikkien komponenttien asemointi on tehty yksinkertaisesti `pack`-menetelmällä, koska tässä esimerkissä riitti, että komponentit saatiin käyttöliittymään alekkain.

Rivillä 17 nähdään, kuinka `self.__muunnosnappi`:iin rivillä 8 sidotussa *tapahtumakäsittelijässä* (metodi `muunna`) kohdistetaan komponenttiin `self.__lähtöarvo` `get`-metodi, jonka paluuarvona saadaan komponentin sisältämä teksti (käyttäjän kirjoittama kellonaika) merkkijonona.

Rivit 33 ja 34 ksäittelevät lopputuloksen näyttämisen käyttäjälle. Toimenpide muodostuu kahdesta vaiheesta. Rivi 33:

```
self.__tulosarvo.delete(0, END)
```

huolehtii siitä, että tuloksen näyttämiseen käytetty `Entry`-komponentti `self.__tulosarvo` tyhjennetään kaikesta vanhasta sisällöstä, joka sillä saattaa olla, mikäli käyttäjä suorittaa useita muunnoksia peräkkäin. Jos tyhjennysoperaatiota ei suoriteta, vanhat tulokset jäävät näkyviin

samanaikaisesti uuden tuloksen kanssa, mikä olisi sotkuista ja epäselvää. Koska Tkinter mahdollistaa, että `Entry`-komponentin sisällön voi tuhota myös osittain, `delete` tarvitsee parametrinaan kaksi kohtaa, joiden välinen teksti halutaan tuhota. Esimerkissä koko syötekenttä haluttiin tyhjäksi, joten parametreina käytettiin arvoja 0 ja `END`. Nolla tarkoittaa kentän alkua ja `END` loppua.

Metodi `insert` rivillä 34:

```
self.__tulosarvo.insert(0, tulos)
```

asettaa merkkijonon `tulos` sisältämän tekstin (siis muutetun kellonajan) `self.__tulosarvo`-komponentin sisällöksi, niin että käyttäjäkin näkee sen. Vaikka `insert`:iä esimerkissä käytetään asettamaan uusi arvo `delete`:n aiemmin tyhjentämälle `Entry`-tyyppiselle komponentille, sitä voitaisiin tarvittaessa käyttää myös lisäämään tekstiä sellaiseen kenttään, joka ei ole tyhjä. Tämän vuoksi `insert`:illä on ensimmäisenä parametrina kokonaisluku, joka kertoo, kuinka monennesta merkistä alkaen uusi teksti lisätään vanhan tekstin keskelle. Jos tekstiä ollaan lisäämässä tyhjään `Entry`-kenttään, kuten esimerkissä tapahtuu, parametrin arvon on oltava nolla, koska uusi teksti halutaan lisätä kentän alkuun (ensimmäisestä merkistä eteenpäin). Ohjelmointikielissähän järjestysnumerointi (eli indeksointi) tunnetusti alkaa lähes poikkeuksetta nollasta. Arvoa `END` voi käyttää, jos teksti halutaan lisätä syötekentän loppuun.

Muutamia huomioita esimerkin toteuksesta

Esimerkki 6 on periaatteessa aivan toimiva käyttöliittymä, mutta siinä on muutamia ongelmia.

Jos käyttäjä niin haluaa, hän voi käydä kirjoittamassa itse tekstiä alempaan syötekenttään, joka on varattu lopputuloksen esittämiseen. Eli tavallaan käyttäjä voi väärentää tuloksia.

Toinen ongelma liittyy käyttöliittymän suunnitteluun. Graafisesta käyttöliittymästä pitäisi pyrkiä muodostamaan niin itsestään selvä, että käyttäjä osaa käyttää sitä ilman erillistä ohjeistusta. Esimerkin käyttöliittymässä näin ei ole: siitä ei esimerkiksi käy mitenkään ilmi, että ylempään syötekenttään on tarkoitus kirjoittaa kellonaika ja painaa sen jälkeen `muunna`-painonappia.

4.3 Laajempi esimerkki: yksinkertainen nelilaskin

Tutkitaan esimerkki, joka ei sisällä kovin paljoa uusia asioita, mutta kokoaa yhteen kaiken tähän mennessä käsitellyn.

Esimerkkikoodi

```
1  from tkinter import *

2  NaN = float('NaN')    # Not A Number: vakio jonka avulla voidaan
3                        # esittää määrittelemättömiä desiaalilukuja.
4  class Käyttöliittymä:
5     def __init__(self):
6        self.__float_tulos = NaN

7        self.__pääikkuna = Tk()

8        # Luodaan käyttöliittymän komponentit
9        self.__entryA       = Entry(self.__pääikkuna)
10       self.__entryB       = Entry(self.__pääikkuna)

11       self.__labelA       = Label(self.__pääikkuna, text="A:")
12       self.__labelB       = Label(self.__pääikkuna, text="B:")

13       self.__tulosotsikko = Label(self.__pääikkuna, text="Tulos:")
14       self.__tulosarvo    = Label(self.__pääikkuna, text=NaN)  # ◁◁◁

15       self.__summanappi   = Button(self.__pääikkuna, text="A + B", command=self.summa)
16       self.__miinusnappi  = Button(self.__pääikkuna, text="A - B", command=self.miinus)
17       self.__kertonappi   = Button(self.__pääikkuna, text="A * B", command=self.kerto)
18       self.__jakonappi    = Button(self.__pääikkuna, text="A / B", command=self.jako)
19       self.__siirtonappi  = Button(self.__pääikkuna, text="tulos→A", command=self.siirto)
20       self.__vaihtonappi  = Button(self.__pääikkuna, text="A↔B", command=self.vaihto)
21       self.__lopetusnappi = Button(self.__pääikkuna, text="lopeta", command=self.lopeta)

22       # Sijoitellaan komponentit grid-mekanismin avulla
23       self.__labelA.grid(row=0, column=0, sticky=E)
24       self.__entryA.grid(row=0, column=1, columnspan=2)
25       self.__labelB.grid(row=0, column=3, sticky=E)
26       self.__entryB.grid(row=0, column=4)

27       self.__tulosotsikko.grid(row=1, column=3, sticky=E)
28       self.__tulosarvo.grid(row=1, column=4)

29       self.__summanappi.grid(row=2, column=0, sticky=E+W)
30       self.__miinusnappi.grid(row=2, column=1, sticky=E+W)
31       self.__kertonappi.grid(row=3, column=0, sticky=E+W)
32       self.__jakonappi.grid(row=3, column=1, sticky=E+W)
33       self.__siirtonappi.grid(row=2, column=2, sticky=E+W)
34       self.__vaihtonappi.grid(row=3, column=2, sticky=E+W)

35       self.__lopetusnappi.grid(row=4, column=4, sticky=E)

36       # Käynnistetään käyttöliittymä
37       self.__pääikkuna.mainloop()
```

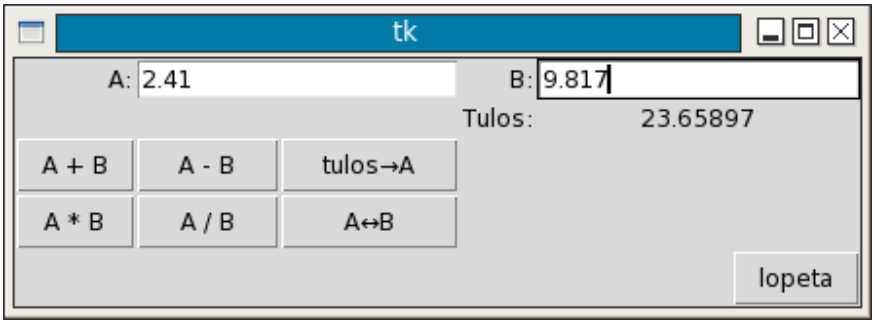
Esimerkki 7: yksinkertainen nelilaskin

```
39      # Tapahtumakäsittelijät
40      def summa(self):
41          (a, b) = self.hae_lähtöarvot()
42          self.__float_tulos = a + b
43          self.aseta_tulosarvo()

44      def miinus(self):
45          (a, b) = self.hae_lähtöarvot()
46          self.__float_tulos = a - b
47          self.aseta_tulosarvo()

48      def kerto(self):
49          (a, b) = self.hae_lähtöarvot()
50          self.__float_tulos = a * b
51          self.aseta_tulosarvo()

52      def jako(self):
53          (a, b) = self.hae_lähtöarvot()
54          self.__float_tulos = a / b
55          self.aseta_tulosarvo()

56      def vaihto(self):
57          (a, b) = self.hae_lähtöarvot()
58          self.__entryA.delete(0, END)
59          self.__entryA.insert(0, b)  # ◁◁◁
60          self.__entryB.delete(0, END)
61          self.__entryB.insert(0, a)  # ◁◁◁

62      def siirto(self):
63          self.__entryA.delete(0, END)
64          self.__entryA.insert(0, self.__float_tulos)  # ◁◁◁

65      def lopeta(self):
66          self.__pääikkuna.destroy()

67      # Normaalitmetodit (ei-tapahtumakäsittelijämetodit)
68      def hae_lähtöarvot(self):
69          try:
70              a_arvo = float(self.__entryA.get())
71          except ValueError:
72              a_arvo = NaN

73          try:
74              b_arvo = float(self.__entryB.get())
75          except ValueError:
76              b_arvo = NaN

77          return (a_arvo, b_arvo)

78      def aseta_tulosarvo(self):
79          self.__tulosarvo.configure(text=self.__float_tulos)  # ◁◁◁


80  def main():
81      käli = Käyttöliittymä()

82  main()
```

Esimerkki 7: yksinkertainen nelilaskin (jatkoa)

Esimerkkikoodin tuottama tulos

Kun esimerkkiohjelma suoritetaan saadaan seuraava käyttöliittymä:



Kuva 10: esimerkin 7 nelilaskinkäyttöliittymä (Linux/GNOME)

Kuinka koodi toimii

Ensimmäinen asia, joka käyttöliittymän toteuttavasta Python-koodista pistää silmään, on se että komponenttien muodostaminen `__init__`-metodissa on hyvin mekaanista, lähes leikkaa–liimaa tyylistä, koodausta. Tämä on valitettava ja tunnettu ongelma käyttöliittymiä käsin koodattaessa: jokainen komponentti pitää luoda ja asetella käsin, mikä johtaa siihen, että koodissa saattaa monimutkaisessa käyttäliittymässä olla kymmeniä tai satoja hyvin vähän toisistaan poikkeavia rivejä. Vertaa toisiinsa vaikkapa esimerkin rivejä 15–21 tai 29–34: aika tylsää ja itseään toistavaa. Tämä on kuitenkin käyttöliittymien ohjelmoinnin ominaisuus, josta ei pääse eroon, jos koko ohjelma kirjoitetaan käsin.

Moderneissa käyttöliittymäympäristöissä on valmiina mukana työkalut, joilla varsinaisen käyttöliittymän muodostaminen (komponenttien valinta ja niiden sijoittelu) tapahtuu graafisella työkalulla. Konkreettisesti tämä tarkoittaa sitä, että käyttöliittymän suunnittelija valitsee ja siirtää hiirellä tarvitsemansa komponentit paikoilleen ilman, että yhtään koodiriviä on tarpeen kirjoittaa. Toki tällaisissakin työympäristöissä käyttöliittymän toiminnallisuuden (mitä tapahtuu, kun vaikka hiirellä painetaan jotain painonappia) toteuttaminen vaatii koodaamista.

Loppupeleissä esimerkin 7 toteutuksessa ei ole montaa uutta asiaa. Tutkitaan ne lyhyesti läpi.

Rivillä 2 määritellään desimaalilukuvakio `NaN` (Not A Number), jota voidaan käyttää esittämään ohjelmassa sellaisia lukuarvoja, joita ei ole määritelty. Vakiolla ei ole mitään tekemistä käyttöliittymien kanssa sinällään, mutta sille on ohjelmassa tarvetta, koska käyttäjän toiinnasta riippuen saataa tulla vastaan tilanteita, joissa käsiteltävillä luvuilla ei ole järkevää arvoa. Esimerkiksi jos käyttäjä painaa yhteenlaskunappia (`A + B`) ennen kuin hän on kirjoittanut mitään arvoja syötekenttiin `A` ja `B`: tulos määrittelemätön. Jos on koskaan ollut

tekemisissä Matlabin kanssa, NaN on käsitteenä luultavasti tuttu jo entuudestaan.

Toisin kuin esimerkissä 6, jossa lopputulos esitettiin `Entry`-tyyppisessä komponentissa, laskinohjelman tulos näytetäänkin `Label`-tyyppisen komponentin avulla. Konkreetisesti tämä tapahtuu niin, että rivillä 79 muutetaan `self.__tulosarvo`:n `text`-attribuutin arvo halutuksi. Muutos tapahtuu kohdistamalla komponenttiin `configure`-metodi, joka saa parametrinaan attribuutin nimen ja sen uuden arvon. Huomaa, että atribuutti `text` on tismalleen se sama, jolle komponenttia luotaessa rivillä 14 annettiin alkuarvo `NaN`:

```
self.__tulosarvo = Label(self.__pääikkuna, text=NaN)
```

Kun tuloksen esittämiseen käytetään `Label`-tyyppistä komponenttia, vältetään esimerkin 6 ongelma, jossa käyttäjä pystyi itse muokkaamaan lopputuloksen arvoa.

Vaikka periaatteessa on niin, että sekä `Entry`- että `Label`-komponenttien arvot ovat merkkijonoja (`str`), tarkkasilmäinen lukija on saattanut huomata jotain outoa riveillä, jotka on merkitty komenteissa ◁◁◁-merkeillä. Kaikilla niillä on komponentin tai sen attribuutin arvoksi asetettu desimaaliluku (`float`). Yleensähän tämä ei ole sallittua, siis se, että merkkijonon paikalla yrittetään käyttää desimaalilukua. Tkinter on kuitenkin tässä suhteessa joustava, ja muuttaa "väärän tyyppisen" arvon automaattisesti merkkijonoksi ennen kuin se asetetaan komponentin/attribuutin arvoksi.

Pieni huomion arvoinen seikka ehkä vielä se, että `Käyttöliittymä`-luokan atribuutti `self.__float_tulos` on normaali `float`-tyyppinen muuttuja, jota käytetään viimeisimpänä suoritetun laskutuloksen arvon tallentamiseen. Sillä ei siis ole mitään tekemistä käyttöliittymän komponenttien kanssa. Tai jos asian haluaa ilmaista toisin: käyttöliittymän toteuttavan luokan olioilla voi olla attribuutteja aivan normaalisti: sekä komponenttien toteutukseen liittyviä että muuhin olion sisäiseen kirjanpitoon tarkoitettuja.

5 Kuvien esittäminen käyttöliittymässä

5.1 Taustaa: Tkinter ja kuvaformaatit

Kaikista käyttöliittymäkirjastoista löytyy jokin mekanismi, jolla käyttöliittymässä voidaan esittää kuvia. Tämä pätee myös Tkinteriin, mutta sen iän vuoksi käsittelykelpoiset kuvaformaatit ovat hyvin rajoituneita ja vanhanaikaisia. Käytännössä vain kuvat, joiden pääte on `.gif`, soveltuvat hyödynnettäviksi Tkinter-käyttöliittymissä. Niin kauan kun kuva on oikeassa formaatissa, sitä voidaan käyttä joidenkin Tkinter-komponenttien kanssa.

5.2 Esimerkki

Toteutetaan yksinkertainen ohjelma, jossa käytetään GIF-formaatissa olevia kuvia `Label`- ja `Button`-tyyppisten komponettien sisältönä. Esimerkissä oletetaan, että kuvatiedostot

`bang.gif` 

`bomb.gif` 

`boom.gif` 

sijaitsevat samassa kansiossa kuin ohjelman lähdekoodi. Kaikki kolme kuvaa ovat 100 pikseliä leveitä. Tämä ei tarkasti ottaen ole välttämätöntä.

Esimerkkiin on upotettu myös toinen uusi asia, jolla ei ole mitään tekemistä kuvien kanssa. Painonappien (`Button`), joilla pommi räjäytetään ja alustetaan, tilaa (*state*) vaihdellaan siten, että vain toinen niistä on kerralla aktiivinen: ei olisi esimerkiksi järkevää, että käyttäjä voisi painaa räjäytysnappia, jos pommi on jo räjäytetty.

Esimerkkikoodi

```
1  from tkinter import *

2  class Pommikäyttöliittymä:
3      def __init__(self):
4          self.__pääikkuna = Tk()

5          self.__bang_kuva = PhotoImage(file="bang.gif")
6          self.__bomb_kuva = PhotoImage(file="bomb.gif")
7          self.__boom_kuva = PhotoImage(file="boom.gif")

8          self.__pommi_label = Label(self.__pääikkuna, image=self.__bomb_kuva,
9                                     height=110, background="white")
10         self.__räjäytä_button = Button(self.__pääikkuna, image=self.__bang_kuva,
11                                        command=self.räjäytä)
12         self.__lataa_button = Button(self.__pääikkuna, text="lataa",
13                                      command=self.lataa, state=DISABLED,
14                                      background="white")
15         self.__lopeta_button = Button(self.__pääikkuna, text="lopeta",
16                                       command=self.lopeta, background="white")

17         self.__pommi_label.pack(fill=BOTH)
18         self.__räjäytä_button.pack()
19         self.__lataa_button.pack(fill=BOTH)
20         self.__lopeta_button.pack(fill=BOTH)

21         self.__pääikkuna.mainloop()

22     def räjäytä(self):
23         self.__pommi_label.configure(image=self.__boom_kuva)
24         self.__räjäytä_button.configure(state=DISABLED)
25         self.__lataa_button.configure(state=NORMAL)

26     def lataa(self):
27         self.__pommi_label.configure(image=self.__bomb_kuva)
28         self.__räjäytä_button.configure(state=NORMAL)
29         self.__lataa_button.configure(state=DISABLED)

30     def lopeta(self):
31         self.__pääikkuna.destroy()

32 def main():
33     käyttöliittymä = Pommikäyttöliittymä()

34 main()
```

Esimerkki 8: pommiohjelma

Esimerkkikoodin tuottama tulos

Kun esimerkin 8 koodi suoritetaan, saadaan tuloksena käyttöliittymäikkuna:

Kuva 11: pommikäyttöliittymä ennen räjäytystä (Linux/GNOME)

Kun käyttäjä painaa "BANG"-kuvalla varustettua nappia, käyttöliittymä vaihtuu seuraavan näköiseksi:

Kuva 12: pommikäyttöliittymä räjäytettynä (Linux/GNOME)

Kuinka koodi toimii

Varsinainen kuvan asettaminen `Label`- tai `Button`-elementin arvoksi ei ole kovin monimutkaista. Tkinter ei kuitenkaan osaa käsitellä GIF-tyyppisiä kuvia suoraan, vaan niistä on ensin muodostettava `PhotoImage`-tyyppisiä olioita. Esimerkissä tämä tapahtuu riveillä 5–7. Kun `PhotoImage`-oliot on muodostettu, ne voidaan asettaa `Label`- tai `Button`-komponentin arvoksi niiden

luomisvaiheessa käyttämällä parametria `image`, kuten riveillä 8 ja 10 on toimittu.

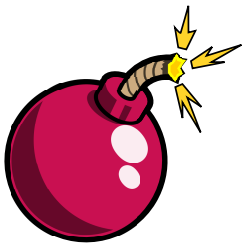
Jos halutaan, komponenttien arvona olevaa kuvaa voidaan myöhemmin muuttaa samalla mekanismilla kuin komponentin muitakin attribuutteja eli `configure`-metodin avulla:

```
self.__jokukomponentti.configure(image=self.__jokuphotoimage)
```

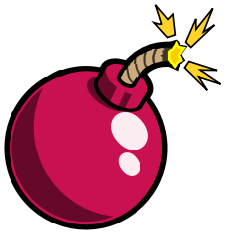
5.3 Taustaa: yksinkertaiset animaatiot

Periaatteessa Tkinterissä voidaan toteuttaa myös yksinkertaisia animaatioita, jos vaihdetaan `Label`in tai `Button`in arvona olevaa kuvaa sopivin aikavälein. Tähän sisältyy kuitenkin pieni haaste: kaikki käyttöliittymään tehdyt muutokset näkyvät käyttäjälle vasta siinä vaiheessa, kun *tapahtumakäsittelijästä* palataan `mainloop`:iin. Sillä kuten aiemmasta muistetaan, `mainloop` kontrolloi kaikkea käyttöliittymässä tapahtuvaa: myös muutosten näyttämistä päivittymistä näytölle. Käytännössä animaatiota voidaan toteuttaa kahdella tavalla: *Ajastimien* (*timer*) tai `update_idletasks`-metodin avulla.

`cherrybomb-1.gif`



`cherrybomb-2.gif`



Esimerkkikoodi

```
1  from tkinter import *

2  class Pommianimaatio:
3      def __init__(self):
4          self.__päivityskerta = 0

5          self.__pääikkuna = Tk()

6          self.__pommikuvat = []
7          for kuvatiedosto in ["cherrybomb-1.gif", "cherrybomb-2.gif"]:
8              self.__pommikuvat.append(PhotoImage(file=kuvatiedosto))

9          self.__pomminappi = Button(self.__pääikkuna, command=self.lopeta)
10         self.__pomminappi.pack()

11         self.päivitä_kuvaa()

12         self.__pääikkuna.mainloop()

13     def lopeta(self):
14         self.__pääikkuna.destroy()

15     def päivitä_kuvaa(self):
16         self.__pomminappi.configure(image=self.__pommikuvat[self.__päivityskerta % 2])
17         self.__päivityskerta += 1
18         self.__pääikkuna.after(200, self.päivitä_kuvaa)

19 def main():
20     käyttöliittymä = Pommianimaatio()

21 main()
```

Esimerkki 9: ajastimen avulla animoitu sytytyslanka

Esimerkkikoodin tuottama tulos

Kuva 13: pommianimaation vaihe 1 (Windows XP)

Kuva 14: pommianimaation vaihe 2 (Windows XP)

Viitteet

[1] Jussi Kasurinen, *Python–Tkinter ja graafinen käyttöliittymä*,
Lappeenrannan teknillinen yliopisto, Tietojenkäsittelytekniikan laboratorio.
2007. ISBN 978–952–214–401–0.
http://www2.it.lut.fi/project/MASTO/material/Python-Tkinteropas_LTY2007.pdf

[2] Fredrik Lundh, *An Introduction To Tkinter*,
effbot.org. 2005. http://effbot.org/tkinterbook/tkinter-index.htm.