6.1 Alamprogramm. Funktsioon

SISSEJUHATUS

Kui on mingi suurem ülesanne lahendada, siis sageli on mõistlik lahendus osadeks jaotada. Alamülesanded on väiksemad ja võib-olla saab mõne nendest hoopis kellelegi teisele teha anda. Kui käsil on maja ehitamine, võib olla mõistlik näiteks kamin lasta ehitada kogenud pottsepal. Kodu koristamise käigus võib näiteks prügiämbri väljaviimise usaldada kellelegi usaldusväärsele inimesele, kes selle töö kiiresti ja korralikult ära teeb ning seejuures ise kaotsi ei lähe.

Programmeerimise juureski on suure ülesande jaotamine osadeks äärmiselt oluline. Ühelt poolt võimaldab see programmi kirjutamise jaotada erinevate inimeste (osakondade, firmade) vahel ja nii ratsionaalsemalt tegutseda. Näiteks saab kogu programm kokkuvõttes siis varem valmis, kui tööd tehakse paralleelselt (tööaega ei pruugi summaarselt küll vähem kuluda). Põhimõtteliselt on olemas võimalused ka programmi enda töö paralleliseerimiseks, aga neid me selles kursuses ei käsitle.

Tegelikult ei huvita meid siin niivõrd, mitu inimest programmi erinevaid osi kirjutab, vaid see võimalus, et juba valmis tehtud alamprogramme saab korduvalt kasutada. Seda võibki lugeda alamprogrammi idee üheks põhialuseks. See ideoloogia on kooskõlas ka nn *DRY*-printsiibiga (ingl k “Don’t repeat yourself” (“Ära korda ennast”)), millega rõhutatakse, et samasuguse koodi mitmekordset kirjapanekut tuleks vältida. Tegelikult oli selle põhimõttega täiesti kooskõlas juba see, et mingite ridade korduva kirjapaneku asemel lasime neid korduvalt täita tsüklil.

OLEME FUNKTSIOONE JUBA KASUTANUD

Erinevates programmeerimiskeeltes (või ka erinevates programmeerimist käsitlevates materjalides) võivad alamprogrammide nimetused ja liigitamine olla mõnevõrra erinevad. Pythonis tegutseme **funktsioonidega**, eespool oleme neid vahel ka käskudeks nimetanud.

Tegelikult oleme funktsioonidega kokkupuutunud juba esimesest nädalast alates. Näiteks juba esimestest programmidest on meil kasutuses funktsioon `print`, millega oleme ekraanile erinevaid suurusi väljastanud. Üsna algusest oleme kasutanud ka funktsiooni `input`, millega oleme kasutajalt infot saanud. Ka funktsioonid `round`, `randint`, `forward`, `back`, `left`, `right`, `color`, `exitonclick` jt on tuttavad. Funktsiooni nime järel on sulud, milles võivad olla (ja sageli ongi) argumendid, nt `forward(100)` viib kilpkonna 100 sammu edasi. Aga oli ka ilma argumentideta funktsioone, nt `exitonclick()`. Funktsiooni `range` puhul on 1, 2 või 3 argumenti.

Mitme argumendiga funktsioonide puhul on väga oluline argumentide järjekord. Näiteks `range(2, 10, 3)` ja `range(3, 10, 2)` tähendavad erinevaid asju.

Eelmainitud funktsioonid on Pythonisse sisseehitatud. Teisisõnu, need on kirjutatud Pythoni arendajate poolt. Paljud funktsioonid on kohekasutatavad, osade puhul on vaja vastav moodul importida. Paraku ei tunne Pythoni arendajad kõikide programmide ja programmeerijate vajadusi, seega oleks päris kasulik, kui saaksime nende poolt kirjutatud funktsioonidele lisaks ka ise meile vajalikke funktsioone juurde kirjutada. See on täiesti võimalik! Tegelikult programmide kirjutamine suuresti just uute funktsioonide loomises seisnebki.

FUNKTSIOONI DEFINEERIMINE

Funktsiooni defineerimiseks ehk kirjeldamiseks kasutatakse võtmesõna *def*. Igal funktsioonil on nimi, mille abil saame teda hiljem kasutada. Järgmises näites on funktsioonile antud nimi *trükiAB*. Põhimõtteliselt on meil nime valikuks üsna vabad käed (sarnaselt muutuja nime valikuga), aga hea programmeerimise stiil on kasutada nime, mis kajastab seda, mida funktsioon teeb. Funktsiooni nimele järgnevad sulud, mille praegu jätame tühjaks (argumente pole). Hiljem vaatame ka seda, mida huvitavat sulgude sisuga teha saab. Pärast sulge tuleb koolon ning kõik järgnevad read, mis antud funktsiooni alla kuuluvad, peavad olema esimese rea (rida, kus asub *def* ja funktsiooni nimi) suhtes taandatud.

```
def trükiAB():
    print("A")
    print("B")
```

Kui meie programm koosneb ainult nendest kolmest reast, siis selle käivitamisel ei juhtu näiliselt midagi. A-d ja B-d ekraanile ei ilmu, kuigi võiks ju! Oleme funktsiooni küll kirjeldanud (defineerinud), aga ei ole seda veel rakendanud (välja kutsunud). Sisuliselt oleme Pythonile "õpetanud" selgeks, kuidas reageerida uuele käsule. Varem sai Python aru näiteks käskudest *print* ja *input*, aga nüüd saab aru ka käsust *trükiAB*!

Kui tahame funktsiooni rakendada, siis saame seda teha samas programmis. Selleks tuleb eraldi reale kirjutada funktsiooni nimi (meil *trükiAB*) ning tühjad sulud. Rakendame funktsiooni lausa kaks korda.

```
def trükiAB():
    print("A")
    print("B")

print("C")
trükiAB()
print("D")
trükiAB()
```

Proovige, mis ilmub ekraanile.

Esimesel real antakse teada, et nüüd hakatakse funktsiooni kirjeldama. Funktsiooni kirjelduse ulatust näitab taane. Teine ja kolmas rida on seega veel funktsiooni osad. Programmi tegelik täitmine algab alles realt `print("C")`, sest eelmised read vaid kirjeldasid funktsiooni. Jätame enne n-ö põhiprogrammi ka ühe tühja rea. See on inimese, mitte Pythoni jaoks. **Funktsiooni kirjeldamine ei too endaga kaasa meie jaoks nähtavaid muutusi.** Küll aga saame võimaluse seda funktsiooni kasutada.

Kui nüüd programmi käivitame, siis ekraanile ilmub esimesena just `print("C")` toimel *C*. Edasi rakendatakse funktsiooni *trükiAB*, mille mõjul ilmuvad ekraanile *A* ja *B*. Seejärel toob rida `print("D")` ekraanile *D*. Pärast seda rakendatakse jälle funktsiooni *trükiAB*.

Ülesanne

Mis on esimene rida, mis ilmub ekraanile?

```
def funktsioon():
    a = 1
    b = 4
    print(a + b)
print(41)
funktsioon()
```

Vali 5

Vali 14

Vali 41

Vali ab

Vali veateade

Veel tuleb tähele panna, et programmis peavad funktsioonid olema kirjeldatud enne nende väljakutsumist. Niisamuti nagu me ei saa inimeselt oodata oskust liita 1-le 2, kui me pole talle liitmist õpetanud, ei saa ka Pythonilt nõuda käskude (*funktsioonide*) täitmist enne, kui need on talle õpetatud (*defineeritud*). Seepärast on tavaks funktsioonide definitsioonid kirjutada kohe programmi algusesse, et neid saaks kogu järgneva programmi jooksul vajadusel välja kutsuda.

Proovige järgmist näidet.

```
trükiAB()
def trükiAB():
    print("A")
    print("B")
```

Mis juhtus? Miks?

Veateateid ei maksa karta, nendega püütakse edasi anda olulist informatsiooni. Programmeerija töö oleks ilma veateadeteta oluliselt keerulisem!

Ülesanne

Programmi käivitamisel ilmus järgmine veateade. Mitmendal programmireal on viga?

```
Traceback (most recent call last):
File "C:\Python33\trüki.py", line 32, in < module >
trükiAB()
NameError: name 'trükiAB' is not defined
```

Sisesta vastus

Funktsiooni võib välja kutsuda ka tsüklis. Näiteks järgmises programmis töötab funktsioon `trükiAB` 10 korda.

```
def trükiAB():
    print("A")
    print("B")

for i in range(10):
    trükiAB()
```

FUNKTSIOONI KIRJELDUSES

Funktsiooni kirjelduses saab kasutada neid funktsioone, millest Python “aru saab”. Sageli kasutatakse neid, mis juba vaikimisi Pythonis olemas on. Kasutasime ju meiegi funktsiooni *print* funktsiooni *trükiAB* kirjelduses. Tegelikult saab kirjelduses kasutada ka funktsioone, mis kellegi enda tehtud on. Nii on järgmises programmis defineeritud funktsioon `funktsioon_a()` ja seda on kohe järgmise funktsiooni kirjelduses kasutatud.

```
def funktsioon_a():
    print("a")
def funktsioon_b():
    funktsioon_a()
    print("b")
```

Katsetage mõlema funktsiooni tööd! Selleks peate need pärast kirjeldust ka välja kutsuma.

Ülesanne

Mitu korda väljastatakse selle programmi käivitamisel ekraanile täht *a*?

```
def funktsioon_a():
    print("a")
def funktsioon_b():
    funktsioon_a()
    print("b")
funktsioon_b()
funktsioon_a()
```

Vali 0

Vali 1

Vali 2

Vali 3

Vali Veateade

Vaata ka kokkuvõtvat videot: http://www.uttv.ee/naita?id=24050

Tegelikult saab funktsioon välja kutsuda ka iseennast. Sellist olukorda nimetatakse rekursiooniks ja see on väga võimas võimalus. Rekursiooni abil saab näiteks sugupuust (või mingist muust puulaadsest struktuurist) konkreetset nime üles otsida. Rekursioon on küllaltki keeruline teema ja algkursuses seda reeglina ei käsitleta. Meie siiski silmaringi materjalis rekursiooni vaatleme.

Funktsiooni kirjelduses saab kasutada ka teisi konstruktsioone peale funktsioonide väljakutsete. Näiteks varasemalt kasutatud "alla lugemise" saame esitada funktsioonina. Näeme, et siin on taane mitmeastmeline, sest tsükkel on funktsiooni sees.

```
from time import sleep

def loe_alla():
    i = 10
    while i > 0:
        print(i)
        i -= 1
        sleep(1)

loe_alla()
```

Funktsioone võib tinglikult jaotada kahte rühma: ühte tüüpi funktsioonide puhul me tahame, et nad midagi ära teeksid ja teiste puhul, et nad midagi arvutaksid ning meile tulemuse tagastaksid. Senised selle osa funktsioonid tegid midagi ära - näiteks väljastasid midagi ekraanile. Tegelikult on selline ka funktsioon *print*, mida oleme korduvalt varem kasutanud. Funktsioonile *print* andsime ikka ette, mida tahtsime ekraanile saada.

```
print("Sisesta PIN-kood:")
```

Funktsioonile etteantavaid suurusi nimetatakse argumentideks ning need pannakse funktsiooni väljakutsel funktsiooni nimele järgnevate sulgude sisse (nii toimime ka ju *print* funktsiooniga). Argumentidest tuleb juttu järgmises peatükis. Samuti tuleb seal juttu funktsioonidest, mille põhiroll ei ole millegi ärategemine, vaid hoopis mingisuguse tulemuse arvutamine.

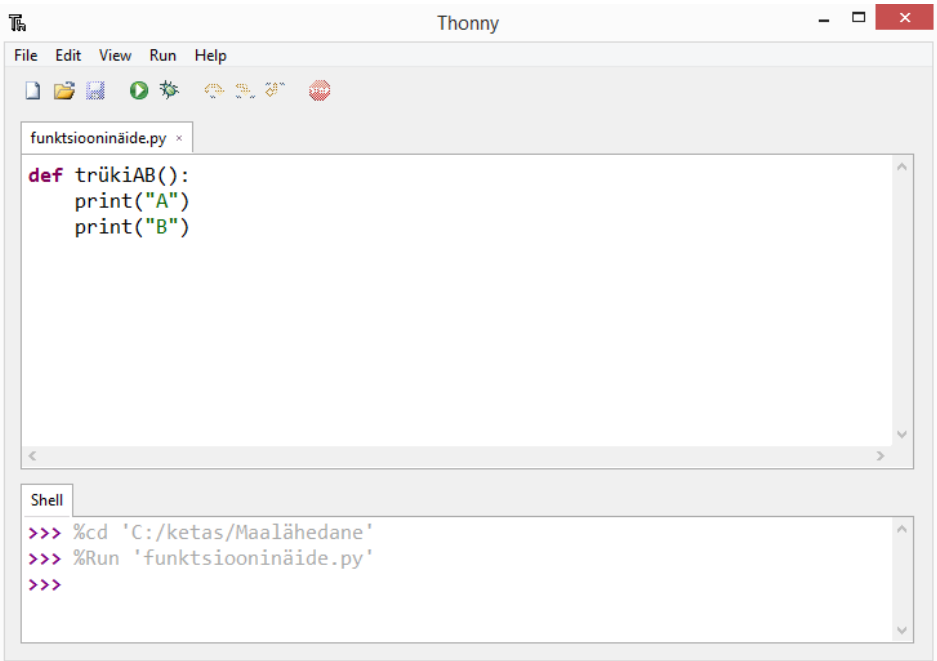
KÄSUREAAKEN

Lisaks sellele, et funktsiooni saab rakendada programmi sees, saab seda teha ka Thonny käsureaaknas. See on Thonny osa, mille tiitelribale on kirjutatud *Shell* ja iga rea ees on **>>>**.

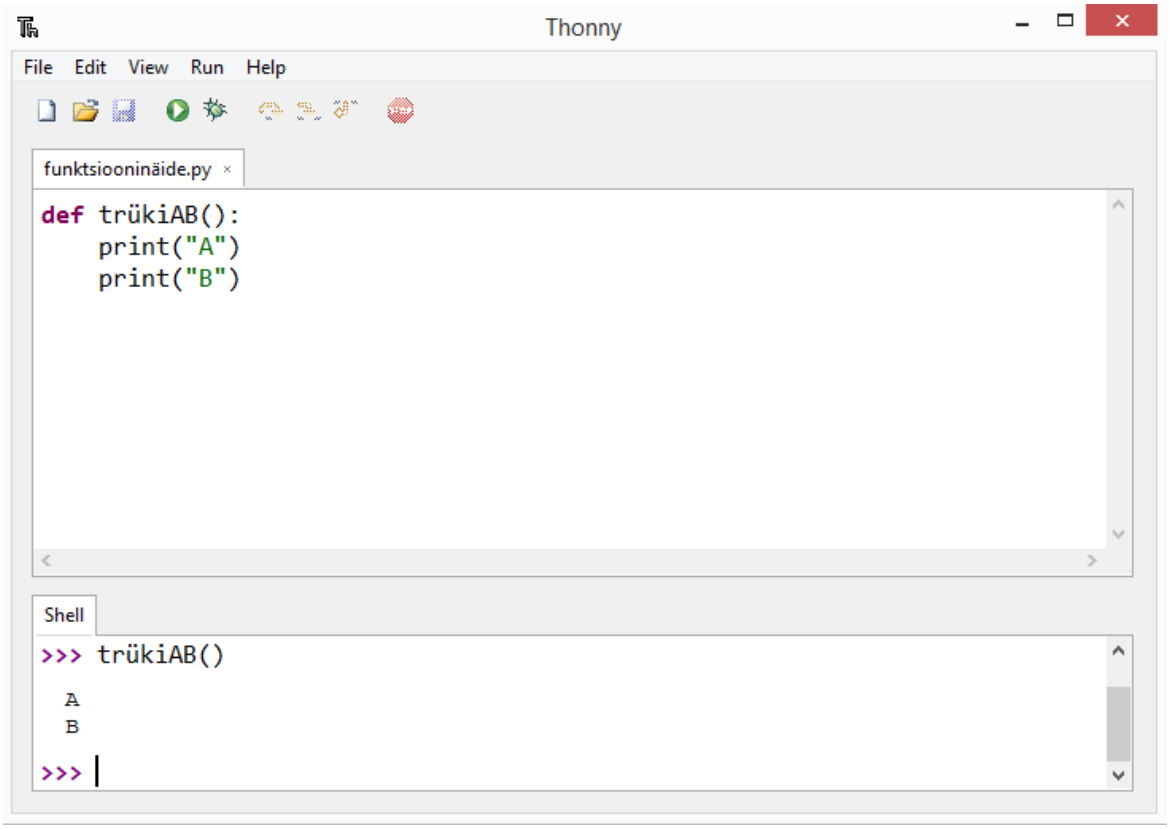
Olgu meil programm, milles on ainult funktsiooni kirjeldus, aga selle funktsiooni rakendamist pole.

```
def trükiAB():
    print("A")
    print("B")
```

Kui nüüd nagu tavaliselt valida roheline nupp, F5 või *Run*-menüüst *Run current script*, siis käsureaaknas näidatakse vaid, mis failis olev programm käivitati.



Kui käsureale kirjutada `trükiAB()` ja reavahetusklahvi vajutada, siis see funktsioon rakendub.



Meie jätkame siiski nii, et kirjutame funktsiooni rakendamised ikka programmiteksti sisse, kuid käsureal saab oma funktsioonide tööd eraldi mugavalt kontrollida.

Vaata ka kokkuvõtvat videot: http://www.uttv.ee/naita?id=24051

6.2 Funktsioonil on väärtus

ÜLDISTAMINE. ARGUMENDID

Üsna sageli on meil vaja peaaegu samasugust tegevust teha mitmes erinevas kohas. Kui eri kohtades on vaja sarnast, kuid teatud variatsiooniga koodi, siis on võimalik kirjeldada funktsioon natuke üldisemalt. Nimelt saab funktsiooni kirjeldada nii, et tema väljakutsumisel antakse ette argumendid ja konkreetne tegevus sõltub nende väärtustest.

Näiteks eelmise allalugemise funktsiooni võime teha üldisemaks nii, et anname ette, millisest arvust tuleb hakata lugema. Hetkel läheb programm sellest isegi lühemaks. Nimelt pole meil enam vaja rida `i = 10`, vaid lisame hoopis muutuja *i* funktsiooni nime järel olevatesse sulgudesse.

```
from time import sleep

def loe_alla(i):
    while i > 0:
        print(i)
        i -= 1
        sleep(1)

loe_alla(4)
```

Nüüd on funktsioon palju paindlikum ja käitub erinevalt olenevalt argumendi väärtusest. Näites on funktsioon *loe_alla* välja kutsutud argumendi väärtusega 4. See tähendab, et *i* saab esialgseks väärtuseks 4. Tsükli jätkamistingimus on täidetud ja nii tehaksegi esimene samm *i* väärtusega 4, mis väljastatakse ekraanile. Järgmine rida muudab *i* väärtuse ühe võrra väiksemaks ja nii edasi.

Proovige funktsiooni välja kutsuda erinevate argumentide väärtustega, nt `loe_alla(8)`, `loe_alla(1)`, `loe_alla(-1)`.

Muudame nüüd ridade `print(i)` ja `i -= 1` järjekorda ja järgmise küsimuse tarbeks kutsume selle funktsiooni välja argumendiga 6.

```
from time import sleep

def loe_alla(i):
    while i > 0:
        i -= 1
        print(i)
        sleep(1)

loe_alla(6)
```

Ülesanne

Milline on esimene arv, mis ekraanile ilmub?

```
from time import sleep

def loe_alla(i):
    while i > 0:
        i -= 1
        print(i)
        sleep(1)

loe_alla(6)
```

Vali 1

Vali 5

Vali 6

Vali Veateade

Kokkuvõttev funktsiooni tutvustav video https://youtu.be/RMYNksCwUPQ

FUNKTSIOON TAGASTAB VÄÄRTUSE

Eelmised näited olid funktsioonidest, mis n-ö tegid midagi ära. Nüüd vaatame funktsioone, mis tagastavad väärtuse või õieti funktsioon ise ongi selle väärtusega. Tegelikult on see väga sarnane funktsiooni mõiste matemaatilise käsitlusega. Nimelt vastab igale argumendile teatud väärtus, mida nimetataksegi selle funktsiooni väärtuseks antud argumendi korral. Näiteks ruutfunktsiooni x^2 korral vastab argumendile 4 funktsiooni väärtus 16, argumendile 5 aga 25. Koosinusfunktsiooni korral vastab argumendile 0 funktsiooni väärtus 1, sest cos(0) = 1. Ruutjuure korral vastab argumendile 9 funktsiooni väärtus 3. Koosinuse ja ruutjuure jaoks on Pythonis funktsioonid olemas, vastavalt `cos` ja `sqrt`. Nende kasutamiseks tuleb enne nad moodulist *math* importida (`from math import cos, sqrt`).

Proovime aga ise kirjeldada funktsiooni, millega saab leida arvu ruudu ehk siis ruutfunktsiooni x^2 väärtuse konkreetse argumendi korral.

```
def ruutu(x):
    return x**2
print(ruutu(4))
```

Pange see programm tööle!

Selliste funktsioonide korral, mis peavad tulemuse tagastama, on olulisel kohal käsk `return`, millega funktsioon väärtuse omandabki. Programmi kolmas rida on juba funktsiooni kirjelduse järel ja rakendab vastloodud funktsiooni argumendiga 4. Nüüd on `ruutu(4)` väärtus juba 16 ja seda saab kasutada nagu arvu ikka. Näiteks saime selle arvu ekraanile väljastada. Saaksime seda ka avaldises kasutada, nt avaldise `ruutu(4) - 17` väärtus oleks -1.

Funktsiooni kirjeldus võib olla pikem, argumente võib olla rohkem (sel juhul eraldatakse need komadega) ja tagastatav suurus võib olla ka muud tüüpi kui arv. Järgmine funktsioon tagastab tõeväärtuse.

```
def kas_raha_jätkub(kghind, kogus, raha):
    hind = kghind * kogus
    if hind <= raha:
        return True
    else:
        return False

if kas_raha_jätkub(2, 4.5, 10):
    print("Ostan")
```

Kuna selle funktsiooni väärtus on tõeväärtustüüpi, siis saab seda kasutada näiteks valikulause tingimuses. Antud juhul on `kas_raha_jätkub(2, 4.5, 10)` väärtuseks `True`.

Kui funktsioon võtab mitu argumenti, siis on oluline nende järjestus. Praegusel juhul antakse funktsiooni väljakutsumisel esimese argumendi väärtus muutujale *kghind*, teise väärtus muutujale *kogus* ja kolmanda väärtus muutujale *raha*. Võiks mõelda nii, et kui funktsioon argumentidega 2, 4.5 ja 10 välja kutsutakse, siis enne funktsioonis olevate lausete juurde minemist tehakse järgmised omistuslaused: `kghind = 2`, `kogus = 4.5` ja `raha = 10`. Tuleb ka märkida, et funktsiooni argumentide nimed on programmi seisukohalt suvalised, neist ei sõltu programmi jaoks mitte midagi (nagu ka muutujate nimede puhul), kuid nimed võiksid ikka olla sellised, mis annaksid programmi lugejale kasulikku infot koodi kohta.

Tegelikult on funktsioon *kas_raha_jätkub* võimalik defineerida kahe reaga (1 rida funktsiooni nime ja argumentide jaoks ning 1 rida *return*-lause jaoks). Soovi korral püüdke funktsiooni kirjeldus nii ümber kirjutada! (Abiks võib olla teadmine, et `return` järel ei pruugi olla ilmutatult `True` või `False`, vaid näiteks avaldis `a * b <= c`.)

Ülesanne

Mis ilmub ekraanile?

```
def reaktsioon(punkte):
    if punkte >= 50:
        return "Olid tubli!"
    else:
        return "Püüa veel!"
print(reaktsioon(50))
```

Vali 50

Vali Olid tubli!

Vali Püüa veel!

Vali Veateade

TAGASTAMINE JA VÄLJASTAMINE

Me oleme nendes materjalides kasutanud kahte küllaltki sarnast terminit: **väljastamine** ja **tagastamine**. Väljastamise all mõtleme põhiliselt millegi ekraanile manamist (eelkõige funktsiooni *print* abil). Tagastamise all aga peame silmas funktsiooni väärtuse tagastamist (käsu *return* abil). (Mingites teistes materjalides võib sõnastus olla teistsugune.) Kui me jaotasime funktsioone sellisteks, mis midagi ära teevad ja sellisteks, mis väärtuse tagastavad, siis kuhu platseerub järgmine funktsioon?

```
def summa(x,y):
    print(x + y)
```

Arvutamine ilmselt toimub. Tulemus aga väljastatakse ekraanile ja seda ei tagastata - `return` puudub. Selline funktsioon kvalifitseerub millegi ära tegijate hulka. Selline liigitamine on oluline seepärast, et erinevatel liikidel on mõnevõrra erinevad rollid. Väärtust tagastavad funktsioonid leiavad kasutust seal, kus saab kasutada arve, tõeväärtusi, sõnesid - oleneb sellest, mis tüüpi väärtus tagastatakse. Näiteks `ruutu(4)` sobib tehniliselt igale poole, kuhu kõlbaks 16. Igale poole, kuhu sobib `True`, sobib tehniliselt ka `kas_raha_jätkub(2, 4.5, 10)`. Nende väärtus ongi igal konkreetsel juhul see, mis vastavalt funktsiooni kirjeldusele ja etteantud argumentidele välja arvutatakse ja *return*-rea abil tagastatakse.

Neid funktsioone, milles *return*-lauset pole ja mis lihtsalt teevad midagi ära ise mingit väärtust saamata, rakendatakse harilikult iseseisvate lausetena. Neid ei saa kasutada avaldistes.

Proovime ülaltoodud funktsiooni summa korral püüda tema väärtuse ekraanile väljastada.

```
def summa(x,y):
    print(x + y)
print(summa(1,3))
```

Proovige see programm tööle panna!

Ekraanile ilmub

```
4

None
```

Mis siis juhtus? Kõigepealt käivitati funktsioon `summa(1,3)` ja väljastati 1 + 3 ehk 4. Seejärel aga püüti ekraanile väljastada funktsiooni `summa(1,3)` väärtus. Kuna aga seal *return*-i sees pole, väärtust pole ja seda sümboliseerib sõna *None* (eesti keeles “mitte miski”). Kui tahame teha funktsiooni, mis annab võimaluse summa arvutada ja siis seda summat avaldistes edasi kasutada, siis peaksime kasutama *return*-lauset.

```
def summa(x,y):
    return x + y
```

Proovige, mis nüüd ekraanile tuleb!

```
def summa(x,y):
    return x + y
print(summa(1,3))
```

Samuti on väljastamise ja tagastamise erinevus märgatav, kui tahame funktsiooni väärtust kasutada mingis avaldises. Näiteks keskmise arvutamisel tahaksime summa jagada veel kahega. Proovime seda mõlema variandiga.

```
def summaReturn(x,y):
    return x + y
print(summaReturn(1,3)/2)

def summaPrint(x,y):
    print(x + y)
print(summaPrint(1,3)/2)
```

Varem väitsime, et iga funktsiooni väljakutsel toimub esmalt argumendiks antavate väärtuste omistamine funktsiooni argumentidele. Funktsiooni argumentidest võib mõelda

kui tavalistest muutujatest, kuid olulise kitsendusega: funktsiooni argumente ei saa kasutada funktsiooni kirjeldusest väljaspool. Proovige järgnevat programmi:

```
def summa(x,y):
    tulemus = x + y
    return tulemus
print(summa(1,3))
print(x)
```

Saate veateate, sest muutujat *x* ei eksisteeri funktsiooni kirjeldusest väljaspool. Täpselt sama lugu on muutujatega, mis on defineeritud funktsiooni kirjelduses. Näiteks kui püüame rea `print(tulemus)` abil väljastada muutuja *tulemus* väärtust, siis seda saame teha vaid funktsiooni kirjelduses, väljaspool funktsiooni kirjeldust muutujat nimega *tulemus* ei eksisteeri. (Lokaalsetest ja globaalsetest muutujatest tuleb põhjalikumalt juttu järgmises peatükis.)

Ülesanne

Mis ilmub ekraanile?

```
def summa(x,y):
    return x + y

a = 4
b = 5
print(summa(a,b))
```

- Vali 9
- Vali 45
- Vali Veateade, sest *summa(a,b)* asemel peab olema *summa(x,y)*
- Vali Veateade, sest *summa(a,b)* asemel peab olema *summa(4,5)*
- Vali Veateade mingil muul põhjusel

PERE SISSETULEKU NÄIDE ILMA FUNKTSIOONITA JA FUNKTSIOONIGA

Vaatleme programmi, mis küsib isa brutopalga, ema brutopalga ning alaealiste laste arvu, ja arvutab selle põhjal pere kuusissetuleku. (Oletame, et iga alaealise lapse kohta makstakse toetust 20€ kuus.)

Esialgu võib eeldada, et mõlema vanema kuupalk on vähemalt sama suur kui maksuvaba miinimum.

Järgnevates videotes on see ülesanne lahendatud vastavalt ilma uut funktsiooni defineerimata ja uue funktsiooni defineerimisega . Videotes ei ole kasutatud keskkonda Thonny, kasutatud on keskkonda IDLE. Kuna videod on valminud juba mõni aasta tagasi, siis rahasummad ei pruugi olla tõepärased.

https://www.uttv.ee/naita?id=16829

https://www.uttv.ee/naita?id=16895

Kontrollülesannetest

Selle materjali põhjal peaks lahendatav olema kontrollülesanne 6.2.

6.3 Argumendid, muutujad

MITU FUNKTSIOONI KOOS

Vaatame näiteid, kus funktsioone kasutatakse nii, et ühe väärtus on teise argumendiks.

```
ümardatud_sisestatud_arv = round(float(input("Sisestage arv ")))
print("See arv ümardatult on " + str(ümardatud_sisestatud_arv))
```

Olge head ja testige seda suhteliselt kokkusurutud programmi.

Selle esimene rida teeb tegelikult palju asju. Tegevus algab “seestpoolt”.

- Funktsiooniga *input* küsitakse kasutaja käest arv. Funktsiooni *input* väärtus on sõne tüüpi.
- Funktsioon *float* võtab argumendiks sõne ja tema enda väärtuseks saab vastav ujukomaarv (murdarv). (Teiste sõnadega: Funktsioon tagastab vastava ujukomaarvu. Või veel kõnekeelsemalt: Funktsioon muudab sõne vastavaks ujukomaarvuks.)
- Funktsioon *round* võtab argumendiks ujukomaarvu ja ümardab selle täisarvuks.

Lõpuks määratakse see täisarv muutuja *ümardatud_sisestatud_arv* väärtuseks.

Niimoodi teise funktsiooni argumendiks saab olla ainult väärtust tagastav funktsioon. Sellisena saab muidugi kasutada ka isekirjeldatud funktsioone, nt

```
def summa(x,y):
    return x + y

a = summa(summa(1, 3)*2, 4)
```

Kokkuvõtva ülevaate eelmisest näiteprogrammist annab järgmine video
https://youtu.be/L02Nzv4AN38

Ülesanne

Mis on muutuja *a* väärtus?

```
def summa(x,y):
    return x + y
a = summa(summa(1,3)*2, 4)
```

Sisesta vastus

ARGUMENTIDEST VEEL

Eelnevates näidetes olid meil argumentideks tavaliselt arvud või sõned, vastavalt konkreetsele funktsioonile muidugi. Olenevalt funktsioonist võivad argumendid olla ka hoopis muud tüüpi. Nagu juba eespool nägime, võib argumentide arv olla funktsioonidel ka erinev. Samuti ei pea argumendid olema omavahel sama tüüpi.

Koostame funktsiooni, mille argumendiks on fail ja tagastatakse sõnede järjend, mille elemendid on read sellest failist.

```
def failistSõnejärjendisse(fail):
    järjend = []
    for rida in fail:
        järjend += [rida.strip()] # järjend.append(rida.strip())
        # strip() võtab lõpust ära reavahetuse
    fail.close()
    return järjend
```

Eks me saime ju nö otse failistki *for*-tsükli abil ridahaaval toimetada. Nüüd aga saame järjendi elemendile (mis vastab faili reale) ka indeksi abil juurde. Samuti oleme reavahetuse ka rea lõpust eemaldanud.

```
def failistSõnejärjendisse(fail):
    järjend = []
    for rida in fail:
        järjend += [rida.strip()] # järjend.append(rida.strip())
        # strip() võtab lõpust ära reavahetused
    fail.close()
    return järjend

failinimi = input("Sisestage failinimi ")
fail = open(failinimi, encoding="UTF-8")
sõned = failistSõnejärjendisse(fail)
print(sõned)
print(sõned[3])
```

See funktsioon tagastas järjendi. See on üheks võimalikuks viisiks mitme väärtuse tagastamiseks. Nimelt saab tegelikult alati tagastada vaid ühe väärtuse. Kui on mitut vaja tagastada, tuleb kasutada sobivat andmestruktuuri, mis siis ühena teisi hõlmab.

Mõnikord on konkreetse argumendi väärtus peaaegu alati sama ja oleks tüütu seda alati uuesti ette anda. Sellisel juhul saame kasutada **vaikeväärtust**. Kui funktsiooni väljakutsel pole seda argumenti näidatud, siis saab argumendi väärtuseks vaikeväärtus.

```
def kasKiiruseÜletamine(kiirus, piirkiirus = 90):
    return kiirus > piirkiirus

print(kasKiiruseÜletamine(100))
print(kasKiiruseÜletamine(90, 70))
```

Vaikeväärtus on ka funktsiooni `print` argumendil `end`. Nimelt vaikimisi vahetab `print` rida, argument `end = "\n"`. Võime aga ka ise selle argumendi väärtuse ette anda.

```
print("Väljastatav tekst ", end = "lõppu juurde")
```

Sõnedega oleme kasutanud ka funktsioone, mis on sõnega ühendatud punktiga.

```
print("tartu".capitalize())
print("Tartu".endswith("tu"))
print("Tartu".upper())
linn = "Tartu"
print(linn.lower())
```

Tundub nagu oleks sellisel juhul argument mitte sulgudes vaid hoopis funktsiooni ees. Põhimõtteliselt see nii ongi. Selliseid funktsioone nimetatakse **meetoditeks** ja need on väga tavalised objektorienteeritud programmeerimises. Meie siin kursusel kasutame meetodeid küll (just eriti sõnede puhul), aga põhjalikum käsitlus jääb selle kursuse raamest välja.

LOKAALSED JA GLOBAALSED MUUTUJAD

Olgu meil programm, milles on defineeritud funktsioon ja siis seda rakendatud.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    return summa

print(summaFunktsioon(1, 2, 3))
```

Pannes programmi tööle saame ekraanile arvu 6, mis muidugi ongi 1 + 2 + 3 korral oodatud.

Lisame programmile rea `print(summa)`.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    return summa

print(summaFunktsioon(1, 2, 3))
print(summa)
```

Saame veateate, mis väidab, et `NameError: name 'summa' is not defined`. Nimi `summa` ei ole defineeritud (kirjeldatud). Kuna funktsiooni sees toimuv on funktsiooni "siseasi", siis tõepoolest muutujat `summa` pole väljaspool funktsiooni olemas. Sellist funktsiooni kehas (kirjelduses) defineeritud muutujat nimetatakse **lokaalseks muutujaks**. Kui tõstame printimise funktsiooni kirjelduse sisse, siis on kõik korras.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    print(summa)
    return summa

print(summaFunktsioon(1, 2, 3))
```

Esimene 6 tuleb ekraanile `print(summa)` toimel ja teine 6 `print(summaFunktsioon(1, 2, 3))` toimel.

Samuti on lokaalsed ka funktsiooni argumendid. Muutuja `b` on funktsiooni sees täiesti olemas.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    print(b)
    return summa

print(summaFunktsioon(1, 2, 3))
```

Funktsioonist väljas aga mitte.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    return summa

print(summaFunktsioon(1, 2, 3))
print(b)
```

Lokaalsed muutujad luuakse funktsiooni igal käivitamisel uuesti ja nad hävivad, kui funktsioon töö lõpetab. Lokaalsed muutujad on funktsiooni siseasi, väljast neid näha pole. Neile saab panna sama nime, mis juba programmi põhiosas kasutuses (või mõnes teises funktsioonis kasutuses olnud).

Näiteks on järgmises programmis nii funktsioonis kui põhiprogrammis muutuja `summa`.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c
    return summa

summa = 10
print(summaFunktsioon(1, 2, 3))
print(summa)
```

Näeme, et viimases reas on muutuja `summa` väärtus just see, mis ta põhiprogrammis sai - funktsioonis toimuv tema väärtust ei muuda.

Põhiprogrammis defineeritud muutujad on **globaalsed muutujad**. Nende väärtusi saab kasutada nii põhiprogrammis kui ka funktsioonide sees. Kui aga funktsioonis on sama nimega lokaalne muutuja, siis globaalset muutujat seal kasutada ei saa. Eelmises näites just nii oligi.

Järgmises näites aga saab globaalset muutujat funktsioonis kasutada, sest sellenimelist lokaalset muutujat funktsioonis pole.

```
def summaFunktsioon(a, b, c):
    summa = a + b + c + gm
    return summa

gm = 17 # globaalne muutuja
print(summaFunktsioon(1, 2, 3))
```

Teatud juhtudel on globaalsete muutujate kasutamine funktsioonis ebasoovitav ja nii võib näiteks ka kontrollülesannete automaatkontrolli tagasiside sellele tähelepanu juhtida.

KOKKUVÕTE

Funktsioonid e. alamprogrammid võimaldavad (sageli küllalt keerulise) programmilõigu panna kirja ühekordselt, aga kasutada seda mitmes erinevas kohas.

Funktsiooni *definitsiooni* (kirjelduse) e *def*-lause kehas olevad laused jäetakse esialgu lihtsalt meelde. Neid saab hiljem käivitada, kirjutades funktsiooni nime koos sulgudega. Sellist tegevust nimetatakse funktsiooni *väljakutseks* e rakendamiseks.

Funktsiooni defineerimisel saab jätta mõned detailid lahtiseks. Täpne töö sõltub etteantud argumentide väärtustest.

Funktsioone võib jaotada kahte gruppi – ühed teevad midagi ära ja teised arvutavad ja tagastavad midagi.

Selleks, et funktsiooni saaks kasutada avaldises, peab ta arvutatud väärtuse tagastama. Väärtuse tagastamiseks kasutatakse võtmesõna `return`.

Programmi põhiosas defineeritud muutujaid nimetatakse *globaalseteks muutujateks*, funktsiooni sees defineeritud muutujaid *lokaalseteks muutujateks*.

Kontrollülesannetest

Nüüd peaks käes olema info kontrollülesannete 6.3 ja ka 6.4a, 6.4b ning 6.4c lahendamiseks.

6.4 Graafika ja funktsioon

KILPKONN ÕPIB FUNKTSIOONE

Varasemates osades oleme tähtsamate programmeerimise konstruktsioonidega (nt valikulause ja tsükliga) tutvumisel abiks võtnud kilpkonna. Nii saame parema visuaalse ettekujutuse. Ka funktsioonide mõistmisel võib kilpkonnast abi olla. Kilpkonnagraafikas on meil läbivaks teemaks olnud ruudu joonistamine, jätkame sellega siingi.

Mäletavasti kasutasime viimati ruudu joonistamisel tsüklit.

```
from turtle import *

i = 0
while i < 4:                # Kilpkonn joonistab tsükli abil ruudu.
Tsükli keha läbitakse neli korda.
    forward(100)
    left(90)
    i = i + 1               # Muutujat i suurendatakse ühe võrra

exitonclick()
```

Ruudu joonistamiseks võib defineerida ka spetsiaalse funktsiooni, näiteks nimega *ruut*. Järgmises näiteprogrammis on see funktsioon ilma argumentideta.

Näiteprogramm. Ruut III

```
from turtle import *

def ruut():                 # Defineerime funktsiooni nimega ruut
    i = 0
    while (i < 4):
        forward(100)
        left(90)
        i = i + 1

ruut()                      # Kutsume funktsiooni ruut välja.
Kilpkonn joonistab ruudu küljega 100 pikslit
right(45)                   # Pöörame paremale 45°
ruut()                      # Kutsume uuesti funktsiooni ruut välja

exitonclick()
```

Eelmise näiteprogrammi puhul joonistab kilpkonn alati ruudu, mille külje pikkus on 100 pikslit. Selleks, et me saaksime muuta ruudu külje pikkust vastavalt soovile, lisame funktsiooni *ruut* argumendi. Nüüd saab seda funktsiooni rakendades joonistada erineva

suurusega ruute. Näiteks `ruut(50)` korral on argumendi väärtuseks 50 ja ruudu külje pikkuseks tulebki 50. Kui argumendiks on 75 (`ruut(75)`), siis tuleb külje pikkuseks 75.

Järgnevas programmis ongi seda demonstreeritud.

Näiteprogramm. Ruut IV

```
from turtle import *

def ruut(külg):             # Defineerime funktsiooni nimega ruut,
mille argumendiks on ruudu külje pikkus
    i = 0
    while (i < 4):
        forward(külg)       # Siin kasutataksegi argumenti
        left(90)
        i = i + 1

ruut(50)                    # Kilpkonn joonistab ruudu küljega 50
pikslit
ruut(75)                    # Kilpkonn joonistab ruudu küljega 75
pikslit

exitonclick()
```

Eelmine kord, kui kilpkonna käsitlesime, olid vaatluse all ka värvimise võimalused. Neid on tegelikult veelgi. Näiteks muudetakse taustavärvi käsuga `bgcolor("värvi_nimetus")`. Teeme programmi, mis muudab ringjoone roheliseks ja lisab punase raamiga ruudu. Tausta värvime helesiniseks. Selleks, et ruutu ei joonistataks ringi peale, kasutame käske:

- `up()` - tõsta pliiats üles;
- `down()` - langeta pliiats vastu "paberit".

Lisaks oskab kilpkonn muuta pliiatsi värvi ja täitevärvi vastavalt käskudega:

- `pencolor("värvi_nimetus")` - muuda pliiatsi värvi (tõlkes pliiatsi värv);
- `fillcolor("värvi_nimetus")` - muuda täitevärvi (tõlkes täitevärv).

Näiteprogramm. Ring ja ruut

```
from turtle import *

def ruut(n):                # Defineerime funktsiooni ruudu
joonistamiseks
    i = 0
    while (i < 4):
        fd(n)
        lt(90)
        i = i + 1

pencolor("#32CD32")         # Kilpkonn muudab pliiatsi värvi
laimiroheliseks
fillcolor("red")            # Kilpkonn muudab täitevärvi punaseks
begin_fill()                # Kilpkonn alustab ringi värvimist
circle(100)                 # Kilpkonn joonistab ringi raadiusega 100
pikslit
end_fill()                  # Kilpkonn lõpetab ringi värvimise

up()                        # Pliiats üles
fd(300)                     # Kilpkonn liigub edasi 100 pikslit
lt(90)                      # Kilpkonn pöörab 90° vasakule
fd(50)
down()                      # Pliiats alla

pencolor("red")             # Kilpkonn muudab pliiatsi värvi
fillcolor("#32CD32")        # Kilpkonn muudab pliiatsi värvi
punaseks, täitevärvi laimiroheliseks
begin_fill()
ruut(100)                   # Kilpkonn joonistab ruudu küljega 100
pikslit
end_fill()

bgcolor("pale turquoise")   # Muudame taustavärvi helesiniseks

exitonclick()
```

Eelnevalt oleme vaadanud programme, mis joonistavad kilpkonnaga ruute või ringe. Kui me tahame, et kilpkonn oskaks luua ka teisi kujundeid, näiteks korrapäraseid hulknurki, siis defineerime selleks vastava funktsiooni.

Kirjutame alamprogrammi `hulknurk(n)`, mis joonistaks soovitud nurkade arvuga hulknurga. Selles funktsioonis on argumendiks *n* hulknurga nurkade arv. Näiteks viisnurga joonistamiseks anname argumendi *n* väärtuseks 5 (`hulknurk(5)`). Selleks, et leida mitme kraadi võrra kilpkonn pöörama peab, kasutame korrapärase hulknurga sisenurga suuruse leidmiseks valemit *(n - 2) * 180 / n*, kus *n* on nurkade arv. Seejärel lahutame saadud väärtuse sirgnurgast (180°). Kui valemit lihtsustada, siis saame *360 / n*.

from turtle import *

```
def hulknurk(n):
    i = 0
    nurk = 360 / n          # Arvutatakse kilpkonna pööramise nurga
suurus
    while (i < n):
        forward(100)
        left(nurk)
        i = i + 1

hulknurk(5)
exitonclick()
```

Saame kirjutada hulknurkade joonistamiseks järgmise programmi, kus hulknurga funktsiooni argumentideks on nurkade arv ja külje pikkus: `hulknurk(n, külg)`. Näiteks kuusnurga joonistamiseks, mille külg on 150 pikslit, kirjutame `hulknurk(6, 150)`.

Näiteprogramm. Hulknurk

```
from turtle import *

# Defineerime funktsiooni nimega hulknurk. Argumendid on nurkade arv
ja külje pikkus
def hulknurk(n, külg):
    i = 0
    nurk = 360 / n          # Arvutatakse kilpkonna pööramise nurga
suurus
    while (i < n):
        forward(külg)
        left(nurk)
        i = i + 1

fillcolor("green")
begin_fill()
hulknurk(6, 150)            # Kilpkonn joonistab rohelise korrapärase
kuusnurga küljega 150 pikslit
end_fill()

fillcolor("red")
begin_fill()
hulknurk(7, 50)             # Kilpkonn joonistab punase korrapärase
seitsenurga küljega 50 pikslit
end_fill()

exitonclick()
```

Kilpkonnaga saab veel igasuguseid trikke teha ning võimalusi on veelgi. Lähemalt saab uurida *Pythoni turtle* mooduli dokumentatsioonist. Näiteks kuidas muuta pliiatsi paksust või kilpkonna kuju. Näiteks `shape("turtle")` muudab kilpkonna kuju. Kilpkonna programmeerimisel laske fantaasial lennata, kasutades kõiki oma teadmisi tsüklite, tingimuslausete, muutujate jne kohta.

Järgnevalt esitame näite, mis joonistab naerunäo. Kui me ei soovi joonistada tervet ringjoont, vaid osa sellest, siis saame kasutada käsu *circle* kahe argumendiga väljakutset `circle(r, d)`. Selles käsus tähistab *r* ringi raadiust ja *d* kesknurka kraadides. Näiteks kui soovime joonistada veerand ringjoonest raadiusega 100 pikslit, siis saame kasutada käsku `circle(100, 90)`.

Näiteprogramm. Naerunägu

```
from turtle import *

def silm():                 # Defineerime funktsiooni silmade
joonistamiseks
    pencolor("#000000")
    fillcolor("#FFFFFF")
    begin_fill()
    circle(25)
    end_fill()

pencolor("#000000")         # Pea
fillcolor("#FFFF00")
begin_fill()
circle(100)
end_fill()

up()
bk(45)
lt(90)
fd(100)
rt(90)
down()

silm()                      # Vasak silm

up()
fd(90)
down()

silm()                      # Parem silm

up()                        # Suu
bk(95)
rt(90)
fd(30)
down()
circle(50,180)              # Pool ringjoonest
bgcolor("#AFEEEE")

exitonclick()
```

Järgnevalt esitame näite, mis joonistab ette antud pikkusega ja värviga tähe. Esitatud programmis on defineeritud kahe argumendiga funktsioon `täht(pikkus, värv)`. Näiteks kollase tähe joonistamiseks pikkusega 100 pikslit kirjutame `täht(100, "yellow")`.

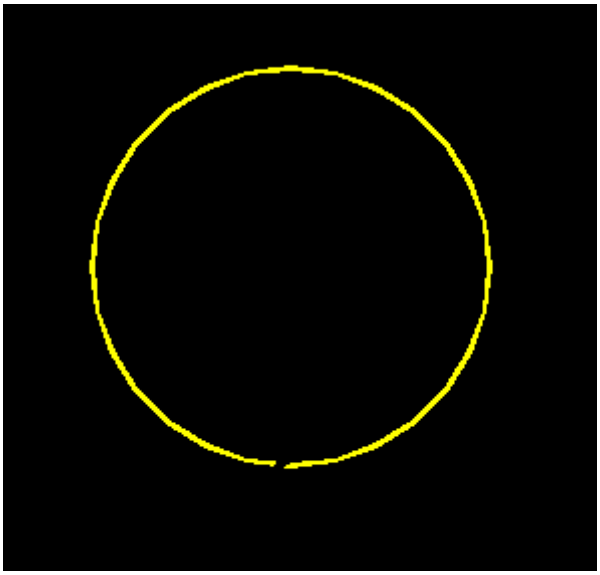
Näiteprogramm. Täht

```
from turtle import *

# Defineerime funktsiooni täht, mis joonistab valitud värvi ja
pikkusega tähe
def täht(pikkus, värv):
    color(värv)
    begin_fill()
    i = 0
    while (i < 5):
        fd(pikkus)
        rt(144)
        i = i + 1
    end_fill()

täht(200, "yellow")         # Kilpkonn joonistab kollase tähe
pikkusega 100 pikslit

exitonclick()
```

Ülesanne. Päikesevarjutus

Kirjutage programm, mis joonistab kilpkonna abil täielikku päikesevarjutust imiteeriva pildi. Pildi taust võiks olla musta või tumesinist värvi.

Tulemus võiks olla midagi sellist. (Tegemist ei ole kohustusliku ülesandega, aga võiksite ikka proovida.)



FUNKTSIOONID KA TKINTERIGA

Üleeelmisel nädalal tegelesime graafikaga ka mooduli Tkinter abil. Siingi saame ise funktsioone defineerida. Teeme näiteks programmi, millega saab kuusiku joonistada. Kuused koosnevad täisnurksetest kolmnurkadest. Defineerime kolm funktsiooni.

- Funktsioon `joonista_tase` joonistab kuuse ühe taseme - täisnurkse kolmnurga, mis on määratud ülatipu ja selle vastaskülje pikkusega (taseme laius). Argumentideks on ülatipu koordinaadid, laius ja värv.
- Funktsioon `joonista_kuusk` joonistab kuuse. Argumentideks on ladva koordinaadid, kuuse laius ja tasemete arv. Kui värvi ei märgita, siis vaikimisi joonistatakse roheline kuusk. Teistvärvi kuuse tegemiseks tuleb värv ette anda, nt `joonista_kuusk(200, 50, 40, 4, "red")`.
- Funktsioon `joonista_mets` joonistab argumendina antud arvu kuuski. Iga kuuse asukoht, tasemete arv ja laius on teatud piirides juhuslikud.

Näiteprogramm. Kuusik

```
from tkinter import *
from random import *

def joonista_tase(ülatipp_x, ülatipp_y, laius, värv):
    tahvel.create_polygon(ülatipp_x, ülatipp_y, ülatipp_x + laius/2,
ülatipp_y + laius/2, ülatipp_x - laius/2,  ülatipp_y + laius/2,
fill=värv, outline=värv)

def joonista_kuusk(ladva_x, ladva_y, laius, tasemeid, värv="green"):
    for tase in range(tasemeid):
        joonista_tase(ladva_x, ladva_y + tase * laius/3, laius,
värv)

def joonista_mets(kuuskede_arv):
    for tase in range(kuuskede_arv):
        joonista_kuusk(randint(1, 600), randint(1, 200),
randint(10, 30), randint(3, 5))

raam = Tk()
# nimeks Kuusik
raam.title("Kuusik")
# loome tahvli laiusega 600px ja kõrgusega 300 px
tahvel = Canvas(raam, width=600, height=300)

joonista_mets(10)
joonista_kuusk(200, 50, 40, 4, "red")
# paigutame tahvli raami ja teeme nähtavaks
tahvel.pack()
# siseneme põhitsüklisse
raam.mainloop()
```

Näiteprogramm. Miiniotsija

Lõpetuseks video miiniotsija mängu tegemisest: http://www.uttv.ee/naita?id=23552

6.5 Silmaring: Rekursioon

MEENUTUSI FUNKTSIOONIST

Programmeerimises on äärmiselt tähtis uute alamprogrammide loomine. Tegelikult suuresti selles programmeerimine seisnebki. Pythonis nimetetakse alamprogramme funktsioonideks. Varasemas oleme funktsioone juba paljudes kohtades rakendanud ehk välja kutsunud. Kui funktsioon on defineeritud, aga seda pole veel rakendatud, siis on tegemist nagu ükskõik millise tarbeeseme või masinaga, mis on küll olemas, aga mida (veel) ei kasutata.

Eespool oleme funktsioone rakendanud n-ö põhiprogrammis, aga ka teiste funktsioonide kirjeldustes. Näiteks järgmises programmis on põhiprogrammis kaks korda rakendatud funktsiooni *ruut*. Funktsiooni *ruut* kirjelduses aga on kasutatud funktsioone *forward* ja *left*.

```
from turtle import *

def ruut():             # Defineerime funktsiooni nimega ruut
    i = 0
    while i < 4:
        forward(100)
        left(90)
        i = i + 1

ruut()                  # Kutsume funktsiooni ruut välja. Kilpkonn
joonistab ruudu küljega 100 pikslit
right(45)               # Pöörame paremale 45°
ruut()                  # Kutsume uuesti funktsiooni ruut välja

exitonclick()
```

Rekursioon

Tegelikult saab funktsiooni välja kutsuda ka selle sama funktsiooni sisemuses. Esialgu võib see tunduda võõras - kuidas siis saab nii olla, et nagu õpetaksime mingit uut asja tegema sellesama asja abil, mida praegu õpetame?! Tegemist on *rekursiooniga* - arvutiteaduse ühe alusmõistega. Reeglina rekursiooni algkursusel ei käsitleta, siingi on see silmaringi materjalide hulgas. Kuna aga tegemist on niivõrd kena ja põneva teemaga, siis ei saa jätta ka selle kursuse huvilisi sellest ilma. Küll aga palun mitte nukrutseda, kui mingid kohad selles osas liiga keerulised tunduvad. Jätke need julgesti vahele! Nädala lõputestis siiski ühtteist on vaja teada.

Rekursioon sobib eriti hästi selliste ülesannete lahendamiseks, kus tervikülesanne koosneb mingis mõttes sarnastest, kuid väiksematest alamülesannetest. Kui põhiülesannet piisavalt kaua alamülesanneteks jagades muutuvad alamülesanded nii väikeseks, et väiksemaks enam minna ei saa või ei taha, siis võiks püüda seda protsessi rekursiivselt esitada.

Klassikaline rekursiooni näide on faktoriaali arvutamine. Positiivse täisarvu n faktoriaal (tähistus n!) on n esimese positiivse täisarvu korrutis. Näiteks 4! = 1 · 2 · 3 · 4 = 24. Eraldi on kokkulepitud, et 0! = 1. Muidugi ka 1! = 1. Rekursiivsena saab faktoriaali leidmist kirjeldada nii, et iga järgmise arvu faktoriaali saame esitada eelmise arvu faktoriaali abil. Näiteks 4! = 4 · 3! ja omakorda 3! = 3 · 2! ning 2! = 2 · 1!. No ja 1! = 1 või kui tahame 0 ka mängu võtta, siis võime öelda ka, et 1! = 1 · 0! ja 0! = 1. Üldistatult saame kaks haru:

- n! = 1, kui n = 0
- n! = n · (n-1)!, kui n > 0

Programm aga on siis selline.

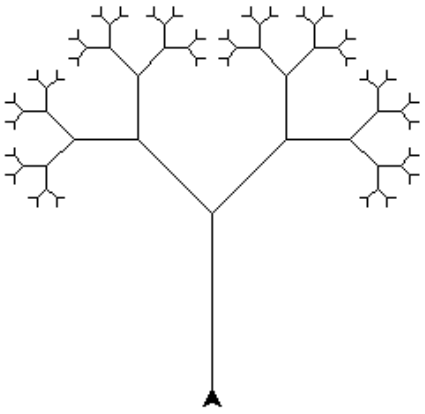
```
def faktoriaal(n):
    if n == 0:              # Rekursiooni baas
        return 1
    else:                   # Rekursiooni samm
        return n * faktoriaal(n-1)

print(faktoriaal(4))
print(faktoriaal(0))
print(faktoriaal(400))
```

Rekursiivsetes programmides ongi alati mitu haru. Selleks, et protsess üldse kunagi lõppeks, peab vähemalt üks haru olema ilma rekursiivse väljakutseta. Seda haru nimetatakse *rekursiooni baasiks*. Rekursiivse väljakutsega (st sellesama funktsiooni väljakutsumisega) haru nimetatakse *rekursiooni sammuks*.

Rekursiooniga on seotud mitmed elulised teemad, näiteks küülikute paljunemise modelleerimine Fibonacci arvude abil jpm. Meie läheme aga visuaalsemate teemade juurde - hakkame vaatlema puid.

Korrapärane puu

Olgu meie eesmärgiks saada selline puu.



Natuke terasemal vaatlemisel märkame, et puu iga haru on ise ka omakorda samasugune puu, aga väiksem. Tolle väiksema puu iga haru on jällegi veel väiksem puu. Selliseid kujundeid, kus osa on terviku sarnane, nimetatakse *fraktaliteks*.

Teoreetiliselt võime lõpmatult joonistada puule väiksemaid oksi, aga praktiliselt poleks sel mõtet ja piiri seadmiseks võtame appi rekursiooni baasi. Näiteprogrammis jõuame rekursiooni baasini, kui järjekordse puu "tüve" pikkus on väiksem kui 5. Sellisel juhul joonistamegi ainult tüve, milleks liigume vastava arvu samme edasi ja kohe tagasi.

Rekursiooni sammu puhul aga joonistame tüve ja kaks haru, mis on omakorda ka puud, aga väiksemad (korrutame teguriga 0,6). Harude joonistamise eel, vahel ja järel tuleb kilpkonna ka sobivalt pöörata.

Palun pange see programm tööle, näete ka, kui kaua kilpkonnal see joonistamine aega võtab. Selleks, et joonis kiiremini tekiks, võib kasutada funktsioone `delay(0)` ja `speed(10)`.

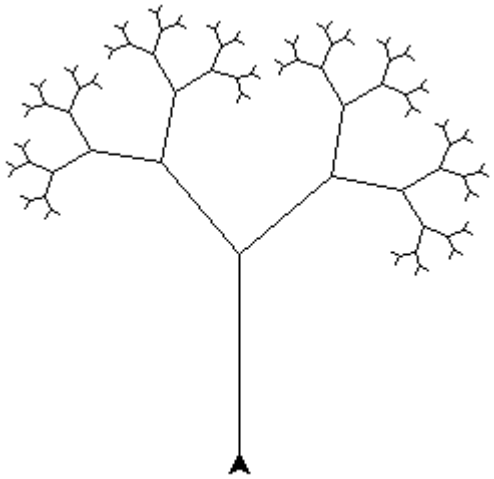
```
from turtle import *

def puu(pikkus):
    if pikkus < 5:          # Rekursiooni baas
        forward(pikkus)     # Ainult tüvi
        back(pikkus)
    else:                   # Rekursiooni samm
        forward(pikkus)     # Tüvi
        left(45)
        puu(0.6 * pikkus)   # Haru, mis on väiksem puu
        right(90)
        puu(0.6 * pikkus)   # Teine haru, mis on ka väiksem puu
        left(45)
        back(pikkus)        # Tüvepidi tagasi

delay(0)
speed(10)
left(90)
puu(100)

exitonclick()
```

Loobume sümmeetriast

Eelmine puu oli meil väga korrapärane ja sümmeetriline. Loobume sellest, et puu peaks sümmeetriline olema. Muudame programmi nii, et enam ei pööratak 45 kraadi vasakule, 90 paremale ja 45 vasakule. Olgu näiteks vasakule pöörded 40 ja 50 kraadi.

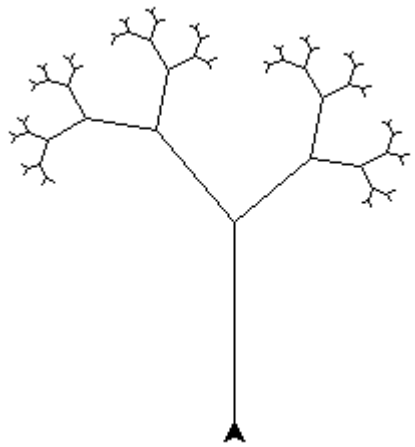
```
from turtle import *

def puu(pikkus):
    if pikkus < 5:
        forward(pikkus)
        back(pikkus)
    else:
        forward(pikkus)
        left(40)            # Pöörame nüüd hoopis 40 kraadi
        puu(0.6 * pikkus)
        right(90)
        puu(0.6 * pikkus)
        left(50)            # ja siin 50 kraadi
        back(pikkus)

delay(0)
speed(10)
left(90)
puu(100)

exitonclick()
```

Näeme, et puu on tõesti natuke ebasümmeetriline.



Päriselt puudel muidugi kõik harud ühepikkused pole. Proovime meiegi nii, et ühes harus oleks tegur endiselt 0,6 aga teisel hoopis 0,5.

```
from turtle import *

def puu(pikkus):
    if pikkus < 5:
        forward(pikkus)
        back(pikkus)
    else:
        forward(pikkus)
        left(40)
        puu(0.6 * pikkus)  # Tegur endiselt 0.6
        right(90)
        puu(0.5 * pikkus)  # Tegur on nüüd 0.5
        left(50)
        back(pikkus)

delay(0)
speed(10)
left(90)
puu(100)

exitonclick()
```



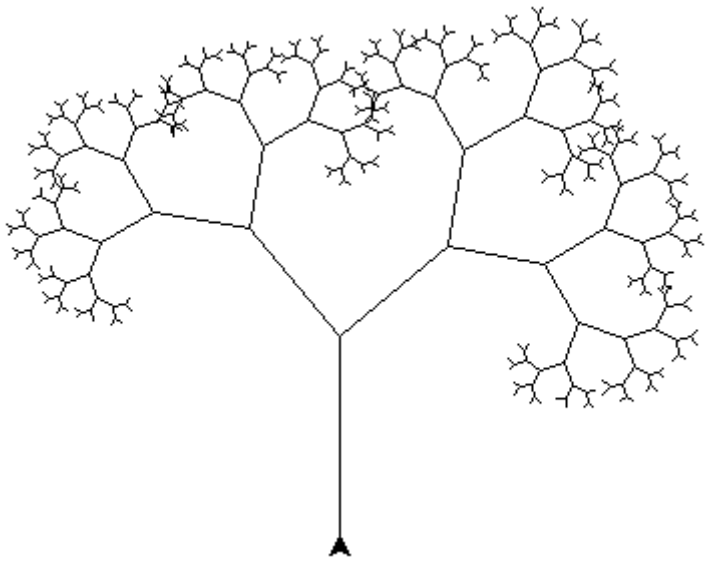
Juhuslik puu

Väga põnevaid võimalusi annab juhuslike arvude kasutamine. Nii annab sama programm erineval korral erinevaid tulemusi. Muudame programmi nii, et iga haru joonistamisel määratakse tegur, millega pikkus korrutatakse, juhuslikult.
Funktsiooniga `randint(6,7)` saame juhusliku arvu 6 või 7. Selleks, et saada 0,6 või 0,7 jagame arvuga 10.

from turtle import *

```
from random import *      # Juhuslike arvude moodul

def puu(pikkus):
    if pikkus < 5:
        forward(pikkus)
        back(pikkus)
    else:
        forward(pikkus)
        left(40)
        puu(randint(6,7) / 10 * pikkus)   # Tegur on 0.6 või 0.7
        right(90)
        puu(randint(6,7) / 10 * pikkus)   # Tegur on 0.6 või 0.7
        left(50)
        back(pikkus)

delay(0)
speed(10)
left(90)
puu(100)
```

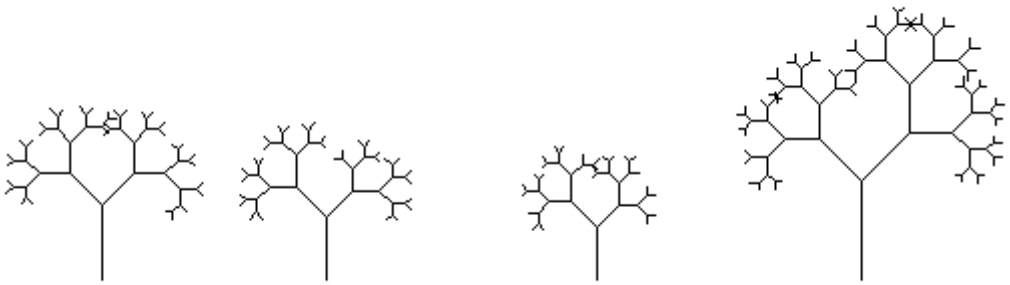
Ühel juhul tuli selline puu.



Teeme nüüd programmi, mis paneb mitu juhusliku suurusega puud kõrvuti.

```
from turtle import *
from random import *

def puu(pikkus):
    if pikkus < 5:
        forward(pikkus)
        back(pikkus)
    else:
        forward(pikkus)
        left(45)
        puu(randint(6,7) / 10 * pikkus)
        right(90)
        puu(randint(6,7) / 10 * pikkus)
        left(45)
        back(pikkus)

def mets(puudearv):
    i = 0
    left(90)
    while i < puudearv:

        pendown()
        puu(randint(20,59))        # Juhusliku pikkusega puu
        penup()
        right(90)
        forward(randint(100,149))  # Puude vahe on ka juhuslik
        left(90)
        i = i + 1

delay(0)
speed(10)
mets(4)

exitonclick()
```

Antud juhul on siis nelja puuga mets.

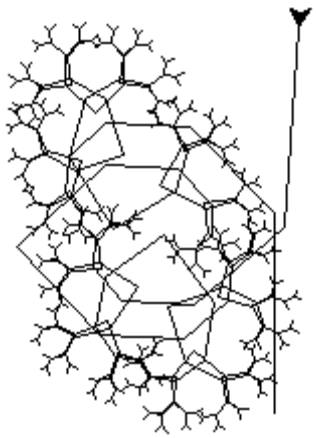


Korrast kaoseni

Kuigi me mitmeid asju muutsime ja lasime isegi juhuslikult arvutada, meenutasid saadud kujundid ikkagi puid. Teeme nüüd näiliselt suhteliselt väikese muudatuse, nimelt muudame ühe pööramise 45 kraadi asemel 44 kraadiks. Tõenäoliselt me silmaga nendel nurkadel vahet ei teeks.

```
from turtle import *
from random import *

def puu(pikkus):
    if pikkus < 5:
        forward(pikkus)
        back(pikkus)
    else:
        forward(pikkus)
        left(44)
        puu(randint(6,7) / 10 * pikkus)
        right(90)
        puu(randint(6,7) / 10 * pikkus)
        left(45)
        back(pikkus)

delay(0)
speed(10)
left(90)
puu(100)

exitonclick()
```

Mis aga juhtub kujundiga?



Näeme, et puuga enam tegemist pole. Väike muudatus viis meid hoopis teise kujundi juurde. Fraktalite puhul nii kipubki olema, et vahel on väike muudatus väga olulise tähendusega.

6.6 Lugu: Alkeemia V

LUGU: ALKEEMIA V

Selle loo on kirjutanud Tartu Ülikooli informaatika eriala esmakursuslane Ander Peedumäe.

Hiline hommikupäike viis endaga kaasa õhtupooliku väsimuse. Kuigi koobas oli ööhakul pimedana tundunud, oli päeval valgust küllalt. Suur osa koopa seintest oli vulkaaniline klaas, mis paiskas mitmevärvilisi valgusvihke kivist majadele.

Tea avas silmad ning ronis ahjupealselt asemelt maha. Linnavanem oli nii temale kui ka Rale kindlustanud sooja voodi ja peavarju nii kauaks, kui tarvis. See ei olnud isegi linna päästmise ainus hüve, sest vanem oli maininud ka hiiglaslikku pidusööki. Tundus, et alkeemikutesse ei suhtuta selles mägilinnas tavalise eelarvamusega.

Ra, kes oli varjuolendeid materdades räsida saanud, ei ärganud meelsasti. Siiski oli tarvis end keskpäevaks püsti ajada, sest linnavanem oli lubanud noortele alkeemikutele näidata midagi, millest mitmed linnaelanikudki ei olnud teadlikud. Nimelt olevat küünlaid süütava hernetondi valmis meisterdanud maag, kes ammusel ajal koopalinnas enda uurimistöid tegi, ning tema elupaik olevat veel alles.

Kiiresti sai selgeks, et kui koopas lepiti kokku kohtumine, siis tõenäoliselt toimus see raeplatsil, kus veider hernehirmutis linna valvas. Pesunaised kogunesid ja rääkisid ilmast ning lapsed mängisid kivikulli. Ra kõrva riivasid jutud peatselt saabuvast lumetormist ning aina rohkem hakkas tunduma, et lahkumisega on kiire.

„Me peame täna minema hakkama või jääme siia terveks talveks," seletas Ra olukorda Teale, kes kortsutas selle peale kulmu.

„Kuue nädala pärast on Ülikoolis lõpupäev. Kui me hiljem jõuame, siis peame järgmisel aastal lõpetama." Ra oli kindel, et ta ei suuda veel ühte aastat päeval õppida ning öösiti tööd rügada, et suures linnas korterit üürida. Plaan oli selge. Sel õhtul pidid nad kaduma.

Üle väljaku astus valge habemega mees, kelle jässakat keha ehtis sinine kuub. Linnavanem naeratas ning juhtis noored linna serva, kus ta peatus ning hakkas seina käsipidi kompima. Tundus, et temagi üllatuseks läbistas kindlas kohas käsi seina. Vanem astus seinale aina lähemale, kuni ta sellesse kadus. Ra ja Tea järgnesid. Seina taga avanes kivitrepp, mis viis päris kõrgele mäe tipu lähedale. Mägiklaasist lae all konutas pisike hütt, mille välimus oli kindlaks märgiks, et seal ei ole keegi pikemat aega elanud.

Vanem sirutas käe välja ning lausus: „Siin me nüüd olemegi. Selles hütis elas maag, kui mina olin veel väike poiss. Võite julgelt sisse astuda: ehk leiate sealt midagi huvitavat. Minul oleks targem minna õhtust pidusööki korraldama."

Noored tänasid ning lasid vanal mehel lahkuda. Majja pääsemine ei olnud kuigi keerukas, kuna puudusid uksed ja aknad. Hütis liikumine oli aga küllaltki raske, sest siin-seal vedeles tolmu- ja kivikuhjades vanu esemeid ja puuhalge. Oli tõenäoline, et maag, kes siin elas, oli kõik tähtsamad asjad enne lahkumist kaasa võtnud või ära peitnud.

Tea sikutas prahihunnikust välja seljakoti, pühkis sellelt tolmu ning näitas Rale. Koti peale oli tikitud peenikeses kirjas „Klaaskuul“ ning selle all „Paberitükk“. Tüdruk avas kompsu ning tõepoolest, kotis oligi pisike läbipaistev kuul ning tükk paberit.

Tea tõstis kulmu: „Ei saa ju olla, et see kott oli tehtud ainsa klaaskuuli ja paberitüki kandmiseks.“

„Kalla see tühjaks,“ pakkus Ra kahtlustavalt.

Tea poetas klaaskuuli paberi järel põrandalaudadele ning uuris kotiriiet uuesti. Sõnad olid kadunud. „Maagiline kandekott tuleks meile kasuks,“ nentis Ra, „Õigupoolest igasugune kott oleks teretulnud, aga me peaksime ennem uurima, kuidas see töötab.“

Tea mõtles pikalt, vaatas kotti ning pistis siis enda käe kotisuust sisse. Riidele tekkis kiri „Käsi“. Veider, selles oli midagi tuttavat, kuid tundus, et kotile ei olnud kirjutatud maagilist käsurida. Tea kehitas õlgu ning Ra pidi sellega leppima, sest ka temal ei olnud paremat ideed.

Peale maja läbi otsimist ostsid Ra ja Tea endale korralikud mägikitsenahast saapad, kindad ja mütsid enda viimaste müntide eest ning jäid ootama õiget hetke lahkumiseks.

Peagi algas sööming linnaväljakul. Sajad lauad olid toidu all lookas ning pidulised ei pannud tähelegi, kui Ra sujuvate liigutustega saiakesi ja vorste seljakotti lükkas. Tea peatas noormehe näppamistalgud ning viipas teda varjulisemasse kohta. Tüdruk oli uurinud enda märkmeid ning ütles enesekindlalt: „Ma tean, kuidas see kott töötab! Sa annad talle sisendi, mille ta nimetab ning koti küljele tikib. Tal ei ole käsurida näha, sest tegemist on funktsiooniga.“ Tüdruk uuris koti voodrivahesid ning leidis väikse maagilise kirja, „See on funktsiooni nimi, mis kutsub esile käsurea kusagilt mujalt iga kord, kui sa midagi kotti pistad.“

Ra taipas, et selline maagia võimaldab panna asjadesse pikki käsuridasid vaid paari sõna abil, nii kaua kuni sellele on mõnes sinu kirjutatud raamatus viidatud.

Kaugemal hakkas rääkima tuttava mehe hääl ning noored kuulsid ka enda nimesid. Tundus, et nad saavad kohe selles linnas kangelasteks, mis tähendas, et kätte oli jõudnud parim hetk pakkida kotid Ra savihärjale selga ning lahkuda märkamatult.

6.7 Kuuenda nädala kontrollülesanded 6.1, 6.2, 6.3

Kuuendal nädalal tuleb esitada nelja kohustusliku ülesande lahendused. Neljanda ülesande puhul on võimalik valida lahendamiseks vähemalt üks järgmistest ülesannetest, kas 6.4a, 6.4b või 6.4c (võib ka kaks või kolm lahendada). Lahendused tuleb esitada Moodle'is, kus need kontrollitakse automaatselt. Moodle'is on ka nädalalõputest 10 küsimusega, millest tuleb vähemalt 9 õigesti vastata.

Kui teile tundub, et automaatkontroll töötab ebakorrektselt, siis palun kirjutage aadressil prog@ut.ee.

Kontrollülesanne 6.1. Bänner

Erinevates poodides ja veebikauplustes võib kohata reklaame, mis kutsuvad inimesi ostlema. Reklaamimiseks kasutatakse näiteks bännereid, mis soovitud reklaamlauset korduvalt kuvavad.

Esmalt koostada funktsioon `banner`, mis

1. võtab argumendiks reklaamlause, mida soovitakse kuvada;
2. tagastab reklaamlause, kus kõik tähed on suured tähed.

Näide `banner` funktsiooni rakendamisest:

```
>>> banner("Osta ja Sa ei kahetse!")
'OSTA JA SA EI KAHETSE!'
>>>
```

Teiseks koostada programm, mis

1. küsib kasutajalt, mitu korda soovitakse reklaamlauset kuvada;
2. küsib kasutajalt, millist reklaamlauset soovib kasutada;
3. rakendab tsükli abil kasutaja sisestatud arv kordi funktsiooni banner, kus igal tsükli sammul tuleb funktsioon välja kutsuda argumendiga, milleks on kasutaja sisestatud reklaamlause;
4. väljastab loodud tsükli abil reklaamlause kasutaja sisestatud arv kordi.

Funktsiooni kirjelduses ei ole tsüklit, vaid funktsiooni kasutatakse tsükli kehas.

NB! Funktsiooni nimi peab olema täpselt see, mis on ülesandes ette antud.

Näide programmi tööst:

```
>>> %Run yl6.1.py

  Mitu korda soovite reklaamlauset kuvada? 3
  Sisestage reklaamlause: Ostan, müün ja vahetan!
  OSTAN, MÜÜN JA VAHETAN!
  OSTAN, MÜÜN JA VAHETAN!
  OSTAN, MÜÜN JA VAHETAN!

>>> %Run yl6.1.py

  Mitu korda soovite reklaamlauset kuvada? 5
  Sisestage reklaamlause: Tulge kõik ostlema! Kõik -70%.
  TULGE KÕIK OSTLEMA! KÕIK -70%.
  TULGE KÕIK OSTLEMA! KÕIK -70%.
  TULGE KÕIK OSTLEMA! KÕIK -70%.
  TULGE KÕIK OSTLEMA! KÕIK -70%.
  TULGE KÕIK OSTLEMA! KÕIK -70%.

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi `murelahendajalt`. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 6.2. Teleri suurus (tollides)

Optimaalseks kauguseks teleri ja vaataja silmade vahel arvatakse olevat 2,5 korda ekraani diagonaali pikkus. Seega kui on teada kaugus diivanist teleri asukohani, siis teleri sobiva diagonaali saab arvutada järgmise valemi järgi:

(teleri diagonaal tollides) = (kaugus meetrites) * 100 * 0,39 / 2,5

Esamalt defineerida funktsioon nimega `teleri_diagonaal`, mis

1. võtab argumendiks ühe arvu, mis tähistab kaugust diivanist teleri asukohani meetrites;
2. arvutab selle põhjal teleri diagonaali tollides;
3. tagastab teleri diagonaali tollides.

Tagastatud teleri diagonaal peab olema ümardatud täisarvuni. Ümardamist peab sooritama funktsioon ise ja selleks tuleb kasutada funktsiooni `round`.

Teiseks rakendada loodud funktsiooni programmis, kus

1. kaugus diivanist telerini (meetrites) küsitakse kasutaja käest;
2. väljastatakse teleri diagonaal (tollides) ekraanile.

Oluline on, et teleri diagonaali arvutamise funktsioon ise ei küsiks kasutajalt midagi ja see funktsioon ise ka ei väljastaks tulemust ekraanile. Need tegevused peab tegema programmis väljaspool funktsiooni, funktsiooni töö on vaid arvutada.

NB! Funktsiooni nimi peab olema täpselt see, mis on ülesandes ette antud.

Näited programmi tööst:

```
>>> %Run yl6.2.py

  Sisesta kaugus: 4
  62

>>>

>>> %Run yl6.2.py

  Sisesta kaugus: 2.75
  43

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 6.3. Peo eelarve

Juubelile on kutsutud hulk inimesi, kellest osa on teatanud, et nad tulevad ja ülejäänute kohta ei ole midagi teada. Peo eelarve koosneb kahest osast: söök ja ruumi rent. Söögi peale arvestatakse iga osaleja kohta 10 eurot. Ruumi rent maksab sõltumata osalejate arvust 55 eurot. Planeerimiseks on vaja programmi, mis arvutab

- maksimaalse eelarve (arvestades, et kõik kutsutud inimesed tulevad kohale) ja
- minimaalse eelarve (arvestades, et kohale tulevad ainult need, kes on juba seda teatanud).

Esmalt koostada funktsioon `eelarve`, mis

1. võtab argumendiks külaliste arvu tähistava täisarvu;
2. arvutab selle põhjal eelarve kogusumma;
3. tagastab eelarve kogusumma. Näiteks argumendiga 5 tagastab funktsioon arvu 105.

Teiseks koostada programm, mis

1. küsib kasutajalt kutsutud inimeste arvu;
2. küsib kasutajalt inimeste arvu, kes on juba tulemisest teatanud;

3. arvutab ja väljastab ekraanile maksimaalse eelarve, kasutades koostatud funktsiooni `eelarve`;
4. arvutab ja väljastab ekraanile minimaalse eelarve, kasutades koostatud funktsiooni `eelarve`.

Näited programmi tööst:

```
>>> %Run yl6.3.py

  Mitu inimest on kutsutud? 25
  Mitu inimest tuleb? 15
  Maksimaalne eelarve: 305 eurot
  Minimaalne eelarve: 205 eurot

>>>
```

```
>>> %Run yl6.3.py

  Mitu inimest on kutsutud? 80
  Mitu inimest tuleb? 48
  Maksimaalne eelarve: 855 eurot
  Minimaalne eelarve: 535 eurot

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi `murelahendajalt`. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

6.8 Kuuenda nädala kontrollülesanded 6.4abc

Kuuendal nädalal tuleb esitada nelja kohustusliku ülesande lahendused. Neljanda ülesande puhul on võimalik valida lahendamiseks vähemalt üks järgmistest ülesannetest, kas 6.4a, 6.4b või 6.4c (võib ka kaks või kolm lahendada). Lahendused tuleb esitada Moodle'is, kus need kontrollitakse automaatselt. Moodle'is on ka nädalalõputest 10 küsimusega, millest tuleb vähemalt 9 õigesti vastata.

Kui teile tundub, et automaatkontroll töötab ebakorrektselt, siis palun kirjutage aadressil proge@ut.ee.

Järgmisest kolmest ülesandest (6.4a, 6.4b, 6.4c) tuleb lahendada vähemalt üks.

Kontrollülesanne 6.4a Tervitused mõtisklustega

Väiksematel üritustel on külaliste ühekaupa tervitamine lihtne. Suurematel üritustel võib ühekaupa tervitamine olla juba kurnavam tegevus.

Esmalt koostada funktsioon `tervitus`, mis

1. võtab argumendiks tervituse järjekorranumbri arvuna, mis näitab mitmes tervitus on käsil;
2. kuvab väljakutsel ekraanile täpselt sellisel kujul tervituse ja vastuse koos tervituse järjekorranumbriga (n tähistab tervituse järjekorranumbrit):

Võõrustaja: "Tere!"
Täna n. kord tervitada, mõtiskleb võõrustaja.
Külaline: "Tere, suur tänu kutse eest!"

Teiseks koostada programm, mis

1. küsib kasutajalt külaliste arvu;
2. rakendab tsükli abil vastav arv kordi funktsiooni `tervitus`, kus igal tsükli sammul tuleb funktsioon välja kutsuda ühe võrra suurema argumendiga kui eelmisel korral.

Funktsiooni kirjelduses tsüklit pole. Küll aga funktsiooni ennast rakendatakse tsükli kehas.

NB! Funktsiooni nimi peab olema täpselt see, mis on ülesandes ette antud.

Näited programmi tööst:

```
>>> %Run yl6.4a.py

  Sisestage külaliste arv: 3
  Võõrustaja: "Tere!"
  Täna 1. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 2. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 3. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"

>>>
```

220 külalise puhul on algus ja lõpp sellised.

```
>>> %Run yl6.4a.py

  Sisestage külaliste arv: 220
  Võõrustaja: "Tere!"
  Täna 1. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 2. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 3. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
```

...

```
  Täna 217. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 218. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 219. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"
  Võõrustaja: "Tere!"
  Täna 220. kord tervitada, mõtiskleb võõrustaja.
  Külaline: "Tere, suur tänu kutse eest!"

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 6.4b Mündid

Euromüntide seerias on kuus erineva nimiväärtusega senti: 1 sent, 2 senti, 5 senti, 10 senti, 20 senti, 50 senti. Sendid väärtustega 1, 2 ja 5 on pronksikarva (vasega kaetud teras), sendid väärtustega 10, 20 ja 50 on kullakarva (vasesulam, mis sisaldab alumiiniumi, tsinki ja tina, nn *Nordic Gold*).

Peres on kombeks, et pronksikarva mündid panna hoiupõrsasse.

Müntide andmed on failis kirjas nii, et iga mündi väärtus on eraldi real. Näiteks nii:

```
1
20
20
5
50
2
2
1
```

Esmalt koostada funktsioon `pronksikarva_summa`, mis

1. võtab argumendiks täisarvude järjendi ja
2. tagastab selles järjendis olevate arvude 1, 2 ja 5 summa.

Teiseks koostada programm, mis

1. küsib kasutajalt selle faili nime, milles asuvad sentide väärtused;
2. moodustab täisarvujärjendi vastavast failist loetud väärtustest;
3. rakendab funktsiooni `pronksikarva_summa`, andes argumendiks koostatud täisarvujärjendi;
4. väljastab ekraanile tulemuseks saadud kõikide pronksikarva sentide summa.

Näide programmi tööst:

Näiteks ülaltoodud andmete korral failis nimega *mündid.txt* peab ekraanile ilmuma

```
>>> %Run yl6.4b.py

  Sisesta failinimi: mündid.txt
  Hoiupõrsasse läheb 11 senti.

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi `murelahendajalt`. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 6.4c Kuupäev

Kuupäevade esitamisel tekib enim probleeme, kui kuupäev kirjutatakse kujul „05.06.2005“ – sellisel puhul pole võimalik aru saada, kas on mõeldud 5. juunit või hoopis 6. maid. Eestis ja enamikes teistes riikides kirjutatakse kuupäev reeglina formaadis DD.MM.YYYY, kuid Ameerika Ühendriikides on levinum järjekord MM.DD.YYYY. (Huvi korral vt https://en.wikipedia.org/wiki/Date_format_by_country.) Segaduse vältimiseks tuleks kuu nimi välja kirjutada.

Esmalt kirjutada kõigepealt funktsioon `kuu_nimi`, mis

1. võtab argumendiks kuu järjekorranumbri;
2. tagastab vastava kuu nime (väikeste tähtedega).

Teiseks luua funktsioon `kuupäev_sõnena`, mis

1. võtab argumendiks ühe sõnena esitatud kuupäeva formaadis “DD.MM.YYYY” (nt '24.02.1918');
2. tagastab selle sama kuupäeva kujul *<päev>. <kuu_nimi> <aasta>. a* (nt '24. veebruar 1918. a'), kusjuures `kuupäev_sõnena` peab ühe toimingu delegeerima funktsioonile `kuu_nimi`. Abiks võib ka olla funktsioon `split`.

Kolmandaks kirjutada programm, mis

1. küsib kasutajalt kuupäeva kujul “DD.MM.YYYY”;
2. väljastab ekraanile vastava kuupäeva sõnena kujul*<päev>. <kuu_nimi> <aasta>. a*.

Näited programmi tööst:

```
>>> %Run yl6.4c.py

  Sisesta kuupäev kujul DD.MM.YYYY: 24.02.1918
  24. veebruar 1918. a

>>>
```

```
>>> %Run yl6.4c.py

  Sisesta kuupäev kujul DD.MM.YYYY: 20.08.1991
  20. august 1991. a

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.