7.1 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 paralleeliseerimiseks, 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.

FUNKTSIOON

Erinevates programmeerimiskeeltes (või ka erinevates programmeerimist käsitlevates materjalides) võivad alamprogrammide nimetused ja liigitamine olla mõnevõrra erinevad. Pythonis tegutseme **funktsioonidega**. Tegelikult olemegi funktsioone juba kasutanud, näiteks *print*, *input*, *randint*. Need funktsioonid on Pythonisse sisseehitatud. Teisisõnu, need on kirjutatud Pythoni arendajate poolt. Kuid kahjuks Pythoni arendajad ei tunne 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 defineerimiseks ehk kirjeldamiseks kasutatakse võtmesõna *def*. Igal funktsioonil on ka 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. Hiljem vaatame ka seda, mida huvitavat nende sulgudega 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 ilmub esimeses reas 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

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
```

Kokkuvõttev funktsiooni tutvustav video
https://youtu.be/98IGeQuV_aI

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

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 ühe põneva puu joonistamise näite pärast poole toome.

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 äraregemine, vaid hoopis mingisuguse tulemuse arvutamine.

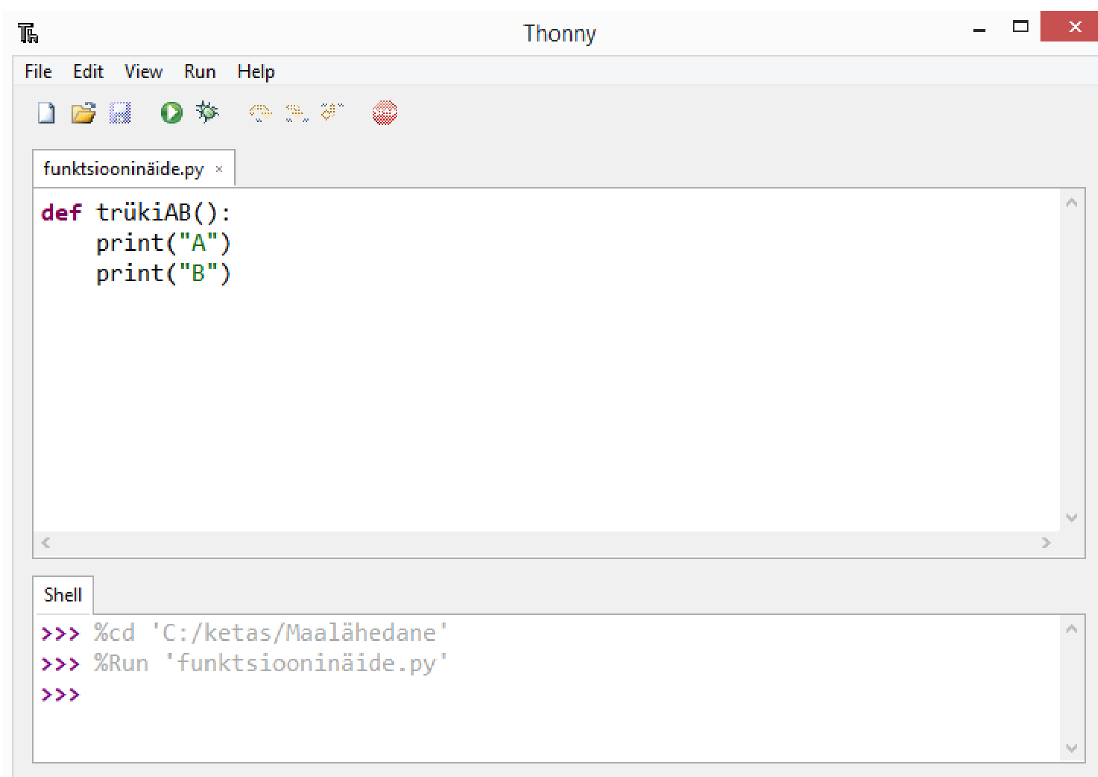
KÄSUREAAKEN

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 **>>>**.

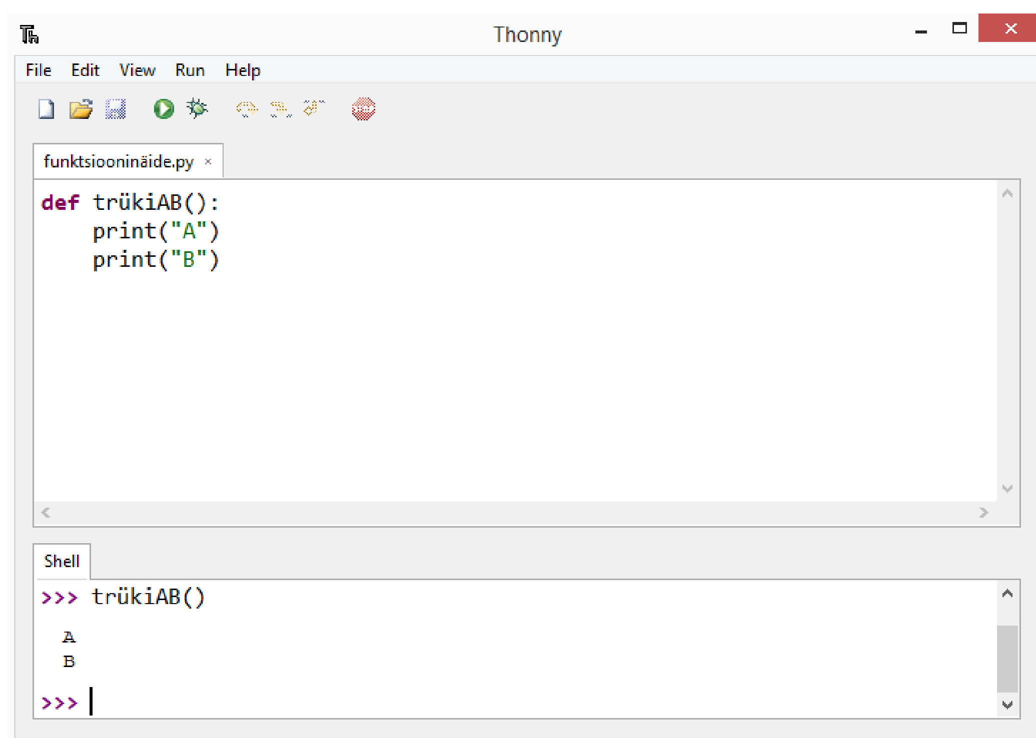
Olgu meil programm, milles on ainult funktsiooni kirjeldus, aga selle funktsiooni rakendamist pole.

```
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.

7.2 Veel funktsioonidest. 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

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 tagastatata - `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))
```

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):
    summa = x + y
    return summa
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(summa)` abil väljastada muutuja *summa* väärtust, siis seda saame teha vaid funktsiooni kirjelduses, väljaspool funktsiooni kirjeldust muutujat nimega *summa* ei eksisteeri.

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 *summa(x,y)*

Vali Veateade, sest *summa(a,b)* asemel peab *summa(4,5)*

Vali Veateade mingil muul põhjusel

MITU FUNKTSIOONI KOOS

Lõpetuseks toome mõned näited, 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)
```

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`.

7.3 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.
    forward(100)             # Tsükli keha läbitakse neli korda.
    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, oli 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 300 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. Näitkes 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*.

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()
```

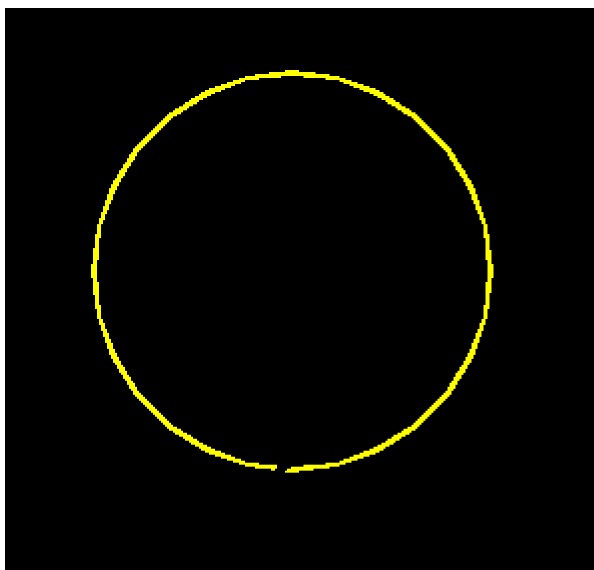
Järgnevalt esitame näite, mis joonistab ette antud pikkusega ja värviga tähe. Esitatud programmis on defineeritud kahe argumendiga funktsioon täht(pikkus, värv). Näiteks kollase tähe joonistamiseks pikkusega 100 pikslit kirjutame täht(100, "yellow").

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 200 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.



7.4 Silmaring. Ametid

Infotehnoloogia alal on palju erinevaid töökohti. Osades ametites on programmeerimine kesksel kohal, teistes jälle on seos kaudsem. Eesti Infotehnoloogia ja Telekommunikatsiooni Liit on loonud portaali, kus on ka erinevate IT ametite ülevaated (http://startit.ee/karjaar/).

IT sektoris töötavate inimeste kohta on selles silmaringi materjalis kaks videot.

Esimene on spetsiaalselt kursuse *Programmeerimisest maalähedaselt* jaoks tehtud video *Firmad ja ametid: intervjuud inimestega* (http://www.uttv.ee/naita?id=21173), kus mitmed programmeerimisega tugevalt seotud inimesed räägivad oma ametist, programmeerimisest üldiselt ja selle õppimisest ning näitavad, millega nad igapäevaselt tegelevad.

Teine on kursuse *Programmeerimise alused* jaoks tehtud video *Inimesed ITs* (http://www.uttv.ee/naita?id=23564), kus mitmed programmeerimisega seotud inimesed räägivad oma ametist, ITst ja programmeerimisest üldiselt ja selle õppimisest ning arutatakse teemal *IT on poiste mängumaa*.

7.5 Kontrollülesanne VII

Laeva teekond

Taust

Sõlm on mõõtühik, mida merenduses kasutatakse laevade ja näiteks tuule kiiruse mõõtmisel. Kui laev liigub kiirusega üks sõlm, siis läbib ta tunnis ühe meremiili (1852 meetrit). Nimetus tuleb sellest, et aegu tagasi mõõdetigi kiirust teatud moel sõlmede loendamisega. Kui on teada kiirus sõlmedes, siis saab arvutada laeva teekonna ööpäevas näiteks selle valemi abil:

teekond (kilomeetrites) = kiirus (sõlmedes) * 1852 / 1000 * 24

Ülesandes nõuame lihtsuse mõttes teekonna ümardamist täisarvuni.

Ülesanne

Defineerige funktsioon nimega *laeva_teekond*, mis võtab argumendiks laeva kiiruse sõlmedes ning arvutab eeltoodud valemi järgi laeva teekonna ööpäevas (kilomeetrites) ja **tagastab** selle. Teekonna pikkus peab olema ümardatud täisarvuni. **Ümardada tuleb juba funktsiooni sees** ehk funktsiooni *laeva_teekond* poolt tagastatud väärtus peab juba olema ümardatud. Ümardamiseks saab kasutada funktsiooni *round*. Näiteks *round(arv)* ümardab muutuja *arv* väärtuse täisarvuni.

Rakendage loodud funktsiooni programmis, kus kasutaja käest küsitakse laeva kiirus sõlmedes ja seejärel väljastatakse ekraanile laeva teekond ööpäevas kilomeetrites.

- Kasutaja sisestatud sõne täisarvuks teisendamiseks saab kasutada funktsiooni *int*.
- Oluline on, et teekonna arvutamise funktsioon ise ei küsiks kasutajalt kiirust ja see funktsioon ise ka ei väljastaks tulemust ekraanile. Need tegevused tehakse programmis väljaspool funktsiooni.

NB! Funktsiooni nimi peab olema täpselt see, mis on ülesandes ette antud, vastasel juhul loeb automaatkontroll lahenduse valeks.

Näide programmi tööst:

```
>>> %Run yl7.py
  Mitu sõlme on laeva kiirus? 30
  Laev läbib ööpäevas kilomeetrites: 1333
>>> 
```

Kontrollülesannete lahendused esitatakse *Moodle*'is.

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.

Ülesande lahendamisel võib abiks olla järgmine video
https://youtu.be/UUsnovBnSRM

7.6 Maalähedane lugu VII

Rasmus jõi kohvi harva. Liisu ei olnud ka eriline kohvisõber, aga linnas oli ta ikka vahel latet või lihtsalt espressot joonud. Tegelikult oli neil kodus isegi masin, millega selliseid peeni jooke teha sai, aga seda kasutas pigem Liisu ema, kes ilma kohvita ennast "üldse inimesena ei tundnud" - madal vererõhk vist?

Maal tundus Liisule kuidagi kohatu Adalt mingeid erilisi kohviliike küsima hakata. Hommikul joodi kohvi küll, siis musta või piimaga. Kohvipulber oli ikka pakist võetud. Ada rääkis sellega seoses ühe loo 80-ndate aastate lõpust - ajast, kui need kõvad kohvipulbripakid alles hakkasid Eestisse jõudma. Polnud seda ilmaimet varem nähtud ja kuidas seda pakki siis lahti peaks saama. Selline tugev pakk - eks ikka kirvega. Ada mäletas, et nii olla naabritalus juhtunud. Rasmus selles väga kindel polnud, sest kuidagi kahtlaselt tundusid varasematest aegadest kõik piinlikud lood olema juhtunud pigem naabritel, aga kõik uhked lood pigem neil. Aga no mine võta nüüd enam kinni.

Kohvist oli Adal aga lugusid veel. Noored said teada, et raskematel aegadel oli kohvipulbrit segatud sigurijuure või isegi võilillejuure või tammetõru pulbriga. Liisu teadis rääkida, et sigur olla tervisele kasulik ja prantslased ning sakslased panevad seda ikka kohvile juurde. Inuliin pidi sees olema. See jutt Adale meeldis, tegelikult oli aianurgas sigur täitsa olemas ja võililli oli muidugi ka. Ada ütles veel, et varem müüdi (kui üldse müüdi) kohvi ikka ubadena - mingil ajal isegi roheliste ubadena. Kuidas siis ubadest või sigurijuurest jook sai?

Nüüd läks Ada tuppa ja võttis kapi otsast ühe kummalise kasti, millel oli vänt peal!

Kuskil sahtlis oli ka kohviube ja nüüd läkski lahti kohvi jahvatamine. Ikka ülevalt oad sisse ja kohviveski sahtlist jahvatatud pulber välja!

> VAHELEPÕIGE Ada, Liisbet ja Rasmus ei mõelnud muidugi sellele, et kohviveski abil saab selgitada programmeerimise alamprogrammi (funktsiooni) tööd. Argumentideks on kohvioad ja tulemuseks on kohvipulber. Seejuures ei pea jahvatamisel teadma, mis seal

kohviveski sees täpselt toimub. Muidugi, kui tahta ise masinat ehitada, siis on vaja just täpselt teada, mis ja kuidas.

Endajahvatatud kohvist sai siis varsti ka jook valmis ja seda nautisid kõik - ikkagi isetehtud. No küll mitte isekasvatatud - siin olid Brasiilia tublid inimesed abiks olnud. Järgmiseks päevaks lubas aga Ada siguritki röstida, siis oleks nagu samba sammudesse oiget ja vasembat pandud.