Sisukord

Kursuse "Teeme ise arvutimänge - algus"

4. RAAMAT

FUNKTSIOONID JA OBJEKTID

Tiina Kull

Tartu Ülikool

2012

Ussimäng taas

Tere taas ja palju uusi avastusi sulle algavaks nädalaks! Olen sulle jällegi valmis pannud kolm uut täiendatud versiooni ussimängust. Esimene neist **Vol4**, salvesta endale *uss4.py* alla. Kribi need oma Idle-sse, uuri koodi ja proovi esialgu iseseisvalt jällegi kõigest aru saada. Järgmisel nädalal vaatame erinevaid plokke detailsete selgitustega. Selles lõigus läheb sul aga tarvis veel helifaili. Selle leiad siit: http://math.ut.ee/~kull/kokkuporge.wav ja igaks juhuks ka ussipea ja kokkupõrge: 


```
# -*- coding: utf-8 -*-
#(teavitame, et kasutame selles Pythoni failis
#8-bitilist Unicode kodeeringut - vajalik täpitähtede jaoks)

# Muutused võrreldes eelmise versiooniga:
# Seinaga kokku põrgates muutub pilt ning lisatud ka heli.

from pygame import *

init()    # paneme PyGame'i tööle
mixer.init()    # paneme hääle tekitamise tööle
aken = display.set_mode([640,480])    # loome PyGame'i akna

# loeme failidest sisse kasutatavad pildid ja leiame nende piltide nähtamatud raamid:
pilt_ussipea = image.load('ussipea.png')
pilt_ussipea_raam = pilt_ussipea.get_rect()
pilt_kokkuporge = image.load('kokkuporge.png')
pilt_kokkuporge_raam = pilt_kokkuporge.get_rect()

# loeme sisse helifailid:
heli_kokkuporge = mixer.Sound('kokkuporge.wav')

# määrame taustavärvi
varv_taust = (255,255,255)    # valge

# need muutujad määravad mängu kiiruse
ussisamm = 5    # nii palju pikseleid liigub uss igal sammul
tsykli_viivitus = 20    # nii palju millisekundeid oodata kahe sammu vahel

# edasi algväärtustame mängu olulised muutujad
(ussipea_x,ussipea_y) = (50,50)    # ussi algne asukoht
suund = K_RIGHT    # ussi algne suund

# funktsioon, mis joonistab hetkeseisu ekraanile:
def joonista():
        aken.fill(varv_taust)    # värvime akna taustavärviga üle
        # paneme pildiraami keskpunkti nendele koordinaatidele
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        # paneme pildi raami sisse
        aken.blit(pilt_ussipea,pilt_ussipea_raam)
        # siiani joonistasime nähtamatusse aknasse, nüüd kopeerime saadud tulemuse nähtavaks
        display.flip()

# funktsioon, mis liigutab ussi ühe ussisammu võrra:
def liiguta_ussi():
        global ussipea_x,ussipea_y
 # tahame muuta globaalseid muutujaid, et fun. sees tehtavad muutused väljaspool fun. kajastuks
        if suund == K_UP:
```

```
                ussipea_y = ussipea_y - ussisamm
        elif suund == K_DOWN:
                ussipea_y = ussipea_y + ussisamm
        elif suund == K_LEFT:
                ussipea_x = ussipea_x - ussisamm
        elif suund == K_RIGHT:
                ussipea_x = ussipea_x + ussisamm

# funktsioon, mis kontrollib, kas uss on kokku põrganud seinaga
def kas_on_kokkuporge():
        # paneme ussipea pildiraami õige koha peale
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        # tagastab True, kui ussi pea pildi raam ei mahu enam akna sisse, vastasel juhul False
        return not aken.get_rect().contains(pilt_ussipea_raam)

while True: # lõpmatu tsükkel
        joonista()
        time.delay(tsykli_viivitus)
        liiguta_ussi()
        if kas_on_kokkuporge():
                break # katkestame tsükli
        for e in event.get(): # vaatame üle kõik vahepeal toimunud sündmused
                if e.type == KEYDOWN: # kas kasutaja vajutas nuppu
                        if e.key in (K_UP,K_DOWN,K_LEFT,K_RIGHT): # kui vajutati nooleklahvi
                                suund = e.key   # uss peab roomama sinna suunas
                if e.type == QUIT: # kasutaja sulges akna, lõpetame
                        exit()

# kui programmi täitmine siia jõuab, siis järelikult oli kokkupõrge
heli_kokkuporge.play()
pilt_kokkuporge_raam.center = (ussipea_x,ussipea_y)
aken.blit(pilt_kokkuporge,pilt_kokkuporge_raam)
display.flip()
while not event.get(QUIT):   # kuni kasutaja pole sulgenud akent
        pass   # ei tee midagi
```

vol5

Nüüd hakkab ussimäng juba veidi ilmet võtma. Uuri vol4 ja vol5 erinevusi ning vii sisse muudatused. Lisaks on sul vaja ka uut helifaili, mille leiad siit: http://math.ut.ee/~kull/soodud.wav ning hiire pilti ja söödud hiire pilti: 


```
# -*- coding: utf-8 -*-
#(teavitame, et kasutame selles Pythoni failis
#8-bitilist Unicode kodeeringut - vajalik täpitähtede jaoks)

# Muutused võrreldes eelmise versiooniga:
# Mängu lisatud hiir, uss saab ta ära süüa.

# loeme sisse kasutatavad moodulid:
from pygame import *

init()    # paneme PyGame'i tööle
mixer.init()    # paneme hääle tekitamise tööle
aken = display.set_mode([640,480])    # loome PyGame'i akna

#loeme failidest sisse kasutatavad pildid ja leiame nende piltide nähtamatud raamid:
pilt_ussipea = image.load('ussipea.png')
pilt_ussipea_raam = pilt_ussipea.get_rect()
pilt_kokkuporge = image.load('kokkuporge.png')
pilt_kokkuporge_raam = pilt_kokkuporge.get_rect()
pilt_hiir = image.load('hiir.png')
pilt_hiir_raam = pilt_hiir.get_rect()
pilt_soodud = image.load('soodud.png')
pilt_soodud_raam = pilt_soodud.get_rect()

# loeme sisse helifailid:
heli_soodud = mixer.Sound('soodud.wav')
heli_kokkuporge = mixer.Sound('kokkuporge.wav')

# määrame taustavärvi
varv_taust = (255,255,255)    # valge

# need muutujad määravad mängu kiiruse
ussisamm = 5    # nii palju pikseleid liigub uss igal sammul
tsykli_viivitus = 20    # nii palju millisekundeid oodata kahe sammu vahel

# edasi algväärtustame mängu olulised muutujad
hiir_soodud = False    # kas hiir on ära söödud, algul mitte
(hiir_x,hiir_y) = (300,300)    # hiire asukoha algväärtus
(ussipea_x,ussipea_y) = (50,50)    # ussi algne asukoht
suund = K_RIGHT    # ussi algne suund

# funktsioon, mis joonistab hetkeseisu ekraanile:
def joonista():
        aken.fill(varv_taust)    # värvime akna taustavärviga üle
        if hiir_soodud:
                pilt_soodud_raam.center = (hiir_x,hiir_y)
                aken.blit(pilt_soodud,pilt_soodud_raam)
        else:
                pilt_hiir_raam.center = (hiir_x,hiir_y)
                aken.blit(pilt_hiir,pilt_hiir_raam)
        # paneme pildiraami keskpunkti nendele koordinaatidele
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        aken.blit(pilt_ussipea,pilt_ussipea_raam)    # paneme pildi raami sisse
# siiani joonistasime nähtamatusse aknasse, nüüd kopeerime saadud tulemuse nähtavaks
        display.flip()
```

```
# funktsioon, mis liigutab ussi ühe ussisammu võrra:
def liiguta_ussi():
        global ussipea_x,ussipea_y
 # tahame muuta globaalseid muutujaid, et fun. sees tehtavad muutused väljaspool fun. kajastuks
        if suund == K_UP:
                ussipea_y = ussipea_y - ussisamm
        elif suund == K_DOWN:
                ussipea_y = ussipea_y + ussisamm
        elif suund == K_LEFT:
                ussipea_x = ussipea_x - ussisamm
        elif suund == K_RIGHT:
                ussipea_x = ussipea_x + ussisamm

# funktsioon, mis kontrollib, kas uss on kokku põrganud seinaga
def kas_on_kokkuporge():
        # paneme ussipea pildiraami õige koha peale
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        # tagastab True, kui ussi pea pildi raam ei mahu enam akna sisse, vastasel juhul False
        return not aken.get_rect().contains(pilt_ussipea_raam)

while True: # lõpmatu tsükkel
        joonista()
        time.delay(tsykli_viivitus)
        liiguta_ussi()
        if kas_on_kokkuporge():
                break # katkestame tsükli
        # kui hiir pole söödud ja ussipea on ülekattes hiirega
        if not hiir_soodud and pilt_ussipea_raam.colliderect(pilt_hiir_raam):
                hiir_soodud = True
                heli_soodud.play()
        for e in event.get(): # vaatame üle kõik vahepeal toimunud sündmused
                if e.type == KEYDOWN: # kasutaja vajutas nuppu
                        if e.key in (K_UP,K_DOWN,K_LEFT,K_RIGHT): # kui vajutati nooleklahvi
                                suund = e.key   # uss peab roomama sinna suunas
                if e.type == QUIT: # kasutaja sulges akna, lõpetame
                        exit()

# kui programmi täitmine siia jõuab, siis järelikult oli kokkupõrge
heli_kokkuporge.play()
pilt_kokkuporge_raam.center = (ussipea_x,ussipea_y)
aken.blit(pilt_kokkuporge,pilt_kokkuporge_raam)
display.flip()
while not event.get(QUIT):   # kuni kasutaja pole sulgenud akent
        pass   # ei tee midagi
```

Üllatus, üllatus, mis küll muutunud on? Uuri järele!

```
# -*- coding: utf-8 -*-
#(teavitame, et kasutame selles Pythoni failis
#8-bitilist Unicode kodeeringut - vajalik täpitähtede jaoks)

# Muutused võrreldes eelmise versiooniga:
# Hiire ära söömisel tekib peale viivitust uus hiir, mida jahtida.

# loeme sisse kasutatavad moodulid:
from pygame import *
from random import *

init()    # paneme PyGame'i tööle
mixer.init()    # paneme hääle tekitamise tööle
aken = display.set_mode([640,480])    # loome PyGame'i akna

# loeme failidest sisse kasutatavad pildid ja leiame nende piltide nähtamatud raamid:
pilt_ussipea = image.load('ussipea.png')
pilt_ussipea_raam = pilt_ussipea.get_rect()
pilt_kokkuporge = image.load('kokkuporge.png')
pilt_kokkuporge_raam = pilt_kokkuporge.get_rect()
pilt_hiir = image.load('hiir.png')
pilt_hiir_raam = pilt_hiir.get_rect()
pilt_soodud = image.load('soodud.png')
pilt_soodud_raam = pilt_soodud.get_rect()

# loeme sisse helifailid:
heli_soodud = mixer.Sound('soodud.wav')
heli_kokkuporge = mixer.Sound('kokkuporge.wav')

# määrame taustavärvi
varv_taust = (255,255,255)    # valge

# need muutujad määravad mängu kiiruse
ussisamm = 5    # nii palju pikseleid liigub uss igal sammul
tsykli_viivitus = 20    # nii palju millisekundeid oodata kahe sammu vahel
# nii palju millisekundeid pärast hiire söömist tekib uus hiir
hiire_viivitus = 1000

# edasi algväärtustame mängu olulised muutujad
hiir_soodud = False    # kas hiir on ära söödud, algul mitte
(hiir_x,hiir_y) = (300,300)    # hiire asukoha algväärtus
(ussipea_x,ussipea_y) = (50,50)    # ussi algne asukoht
suund = K_RIGHT    # ussi algne suund

# funktsioon, mis joonistab hetkeseisu ekraanile:
def joonista():
        aken.fill(varv_taust)    # värvime akna taustavärviga üle
        if hiir_soodud:
                pilt_soodud_raam.center = (hiir_x,hiir_y)
                aken.blit(pilt_soodud,pilt_soodud_raam)
        else:
                pilt_hiir_raam.center = (hiir_x,hiir_y)
                aken.blit(pilt_hiir,pilt_hiir_raam)
        # paneme pildiraami keskpunkti nendele koordinaatidele
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        aken.blit(pilt_ussipea,pilt_ussipea_raam)    # paneme pildi raami sisse
# siiani joonistasime nähtamatusse aknasse, nüüd kopeerime saadud tulemuse nähtavaks
        display.flip()
```

```
# funktsioon, mis liigutab ussi ühe ussisammu võrra:
def liiguta_ussi():
        global ussipea_x,ussipea_y
# tahame muuta globaalseid muutujaid, et fun. sees tehtavad muutused väljaspool fun. kajastuks
        if suund == K_UP:
                ussipea_y = ussipea_y - ussisamm
        elif suund == K_DOWN:
                ussipea_y = ussipea_y + ussisamm
        elif suund == K_LEFT:
                ussipea_x = ussipea_x - ussisamm
        elif suund == K_RIGHT:
                ussipea_x = ussipea_x + ussisamm

# funktsioon, mis kontrollib, kas uss on kokku põrganud seinaga
def kas_on_kokkuporge():
        # paneme ussipea pildiraami õige koha peale
        pilt_ussipea_raam.center = (ussipea_x,ussipea_y)
        # tagastab True, kui ussi pea pildi raam ei mahu enam akna sisse, vastasel juhul False
        return not aken.get_rect().contains(pilt_ussipea_raam)

while True: # lõpmatu tsükkel
        joonista()
        time.delay(tsykli_viivitus)
        liiguta_ussi()
        if kas_on_kokkuporge():
                break # katkestame tsükli
        # kui hiir pole söödud ja ussipea on ülekattes hiirega
        if not hiir_soodud and pilt_ussipea_raam.colliderect(pilt_hiir_raam):
                hiir_soodud = True
                heli_soodud.play()
                # käivitame taimeri, uus hiir tekib pärast viivitust
                time.set_timer(USEREVENT,hiire_viivitus)
        for e in event.get(): # vaatame üle kõik vahepeal toimunud sündmused
                if e.type == KEYDOWN: # kasutaja vajutas nuppu
                        if e.key in (K_UP,K_DOWN,K_LEFT,K_RIGHT): # kui vajutati nooleklahvi
                                suund = e.key   # uss peab roomama sinna suunas
                # taimeri aeg otsas, aeg tekitada uus hiir juhuslikku kohta aknas
                if e.type == USEREVENT:
                        hiir_x = randint(aken.get_rect().left,aken.get_rect().right)
                        hiir_y = randint(aken.get_rect().top,aken.get_rect().bottom)
                        hiir_soodud = False   # uus hiir, seda pole veel söödud
                        time.set_timer(USEREVENT,0)   # paneme stopperi kinni
                if e.type == QUIT: # kasutaja sulges akna, lõpetame
                        exit()

# kui programmi täitmine siia jõuab, siis järelikult oli kokkupõrge
heli_kokkuporge.play()
pilt_kokkuporge_raam.center = (ussipea_x,ussipea_y)
aken.blit(pilt_kokkuporge,pilt_kokkuporge_raam)
display.flip()
while not event.get(QUIT):   # kuni kasutaja pole sulgenud akent
        pass   # ei tee midagi
```

Funktsioonid

Suurte programmide, nagu mängud tavaliselt on, kirjutamisel läheb kood suht kiiresti väga pikaks ja keeruliseks nii oma konstruktsioonide ja valemite poolest kui loetavuse enda mõttes. Sa oled omal nahal seda juba ussimängu koodi ümber kirjutades või oma seiklusmängu luues tunda saanud. Kood läheb aina pikemaks ja segasemaks.

Kõige parem oleks kood organiseerida väiksemateks juppideks, kus igast jupist on programmeerijal hea ülevaade, ta teab mida see jupp teeb ja milline tulemus selle jupi töötamisega kaasneb. Pealegi on ühte juppi kergem programmeerida kui kogu programmi korraga kirjutada. Sellist nö jupitamist tegid sa tegelikult juba ussimängu Vol3 pealt Vol4 peale üleminekul. Vol3 kood oli juba suht pikaks veninud, kuid Vol4 tulemus oli palju paremini struktureeritud, kompaktsem ja arusaadavam.

Mis siis muutus Vol4-s?

Võtsime kasutusele funktsiooni mõiste. **Funktsioon** ei olegi mitte midagi muud kui üks jupp koodi suurest programmist, millele on antud oma nimi ja mis oskab tagastada (välja anda) oma töö tulemust. Funktsioone nimetatakse sageli ka **alamprogrammideks**, sest nad käituvad nagu väikesed iseseisvad programmikesed. Funktsioon on käskude kogumik, mis teeb alati midagi. Funktsioone võib vaadata ka kui ehitusmaterjali maja ehitamisel. Iga materjal on oma otstarbega, oma spetsiifiliste omadustega, oma kujuga jne. Paljudest erinevatest ja ka ühesugustest materjali tükkidest saab oskuslikul kombineerimisel ehitada mistahes ehitisi. Sama lugu on funktsioonidega, neist saab oskuslikul kombineerimisel kokku panna mistahes programmi.

Funktsiooni loomine

Funktsioone luuakse ehk **defineeritakse** võtmesõna **def** abil nagu sa ka ussimäng Vol6-s kindlasti tähele panid. **Def** võtmesõna järele kirjutatakse **funktsiooni nimi** (nime valimisel tuleb lähtuda samadest reeglitest nagu muutujate nimede valimise juures) ning kõige lõpuks **sulud**. Sulgude sisse võib, aga ei pea midagi kirjutama, oleneb funktsiooni töö eripärast, kuid sellest veidi aja pärast. Praegu aga võta lahti Idle tekstiredaktor ja katseta funktsiooni defineerimist.


```
def ahv():
    print("Mulle meeldivad ahvid!")
    print("ahvid "*20)
    print("huuh "*50)
    print("Aitab! Kalad on paremad!")

ahv()
```

Defineerisin funktsiooni *ahv*. See funktsioon ei tee mitte midagi muud, kui prindib neli rida teksti. Funktsiooni tööle panemiseks tuleb funktsiooni nimepidi kutsuda, seda me teeme antud näite viimasel real, kusjuures üliolulised on just **sulud**.

Kuid miks on veel kasulik eraldada grupp käske suurest koodist ja anda sellele käskude grupile ühine nimi? Kas tõesti ainult loetavuse parendamiseks?

Funktsiooni välja kutsumine

Vastan eelmise peatüki viimase rea küsimusele: ei, kindlasti ei ole funktsioonid ainult selleks, et pealkirjastada erinevaid koodijuppe umbes nagu pannakse raamatu peatükkidele pealkirju, ei. Funktsioonidel on palju, palju mõistlikum rakendus.

Tänu funktsioonidele võin ma kirjutada ühte koodijuppi ainult ühe korra ning kui mul on seda juppi vaja mitmes erinevas kohas, siis saan vaid viidata juba olemasolevale osale programmis. Seda võimaldab funktsiooni nimi. Võin funktsiooni välja kutsuda mistahes teises programmi osas ja nii mitu korda kui vaja, ilma et oleks pidanud uuesti ja uuesti sama koodijuppi kirjutama.

Vaatame ühte lihtsat näidet:


```
import random

arv1 = random.randint(1,10)
arv2 = random.randint(1,10)

def kiitus():
    print("See vastus on õige!")
    print("Hästi tehtud!")

def laitus():
    print("See ei ole õige vastus.")

print("Kui palju on ", arv1, "+", arv2)
vastus1 = int(input())

if vastus1 == arv1+arv2:
    kiitus()
else: laitus()

print("Kui palju on ", arv1, "-", arv2)
vastus1 = int(input())

if vastus1 == arv1-arv2:
    kiitus()
else: laitus()
```

Siin näites ma defineerin kaks funktsiooni, *kiitus* ja *laitus*. Need funktsioonid ma kutsun välja kahes erinevas if-lauses, kord kahe arvu summa kontrolli juures, kord kahe arvu vahe kontrolli juures. Kuigi mu funktsioonid on väga väikesed ja ei tee suurt midagi, on aja kokkuhoid aga juba märgatav.

Jah, ma oleksin võinud ka need paar lauset kopeerida mõlemasse if-lausesse, kuid see muudab koodi kirjumaks ja loetamatumaks. Pealegi üldjuhul on funktsioonide sisu märksa keerulisem, kui lihtsalt print käsk.

Veel üks oluline punkt, miks kasutada funktsioone. Kui ma pean mitu korda ühte ja seda sama koodijuppi mitmes eri kohas kasutama, oletame et ma olen juba need jõudnud ära kirjutada ja ma ei kasuta funktsioone. Mingi aja pärast aga avastan, et olen selles programmijupis teinud olulise vea, siis nüüd pean vea parandamiseks terve programmi uuesti läbi käima (ütleme nii 10-100 kohas), siis on see üks väga tüütu asi. Funktsioonide korral oleks mul piisanud vaid funktsiooni sees ühe korra muutus teha ja see oleks funktsiooni nime kaudu kui viida kaudu kõikidesse väljakutsutavatesse kohtadesse mõjunud.

Funktsiooni argumendid

Funktsiooni argument on info, mida saame funktsioonile anda ja see kirjutatakse funktsiooni sulgude sisse. Täpselt nii nagu sa nägid ussimängu Vol6 funktsioonides. Sellist info funktsioonile andmist nimetatakse **funktsioonile argumentide andmiseks**.

Mõistete veel rohkem segamini ajamiseks kutsuvad osad programmeerijad funktsiooni sulgudesse lisatud infot ka **parameetriks**. Suurt vahet neil kahel mõistel tegelikult ei ole. Erinevus tuleb inglisekeelsete sõnade kasutamise eripärast. Seega kui ma annan info funktsioonile tema väljakutsumise hetkel, siis kutsutakse seda infot argumendiks ning kui ma kasutan saadud infot funktsiooni sees millegi tegemiseks, siis parameetriks.

Vaatame seda sama arvutamise näidet, aga nüüd koos argumendiga funktsioonis.


```
import random

arv1 = random.randint(1,10)
arv2 = random.randint(1,10)

def kiitus(vastus):
    print(vastus,  " on õige!")
    print("Hästi tehtud!")

def laitus(vastus):
    print(vastus, "ei ole õige vastus.")

print("Kui palju on ", arv1, "+", arv2)
vastus = int(input())

if vastus == arv1+arv2:
    kiitus(vastus)
else: laitus(vastus)

print("Kui palju on ", arv1, "-", arv2)
vastus = int(input())

if vastus == arv1-arv2:
    kiitus(vastus)
else: laitus(vastus)
```

Näiteprogrammis muutus see, et funktsioon kasutab oma töös nüüd ka talle antud infot (argumenti). Argumenti **`vastus`** kasutatakse funktsiooni sees print käsus.

Argument antakse funktsioonile tema väljakutsumise hetkel.

Antud näites on funktsiooni definitsioonis ja funktsiooni väljakutsumisel kasutatud ühte ja seda sama muutuja nime **`vastus`**. Kuid soovitav on seda mitte teha. Muutujanimed (ka argumendi nimed) võivad olla funktsiooni sees täiesti erinevad nendest, mida kasutatakse funktsiooni väljakutsumise ajal.

Argumente võib funktsioonil olla ka mitmeid, igale argumendile tuleb anda oma nimi. Funktsiooni väljakutsumisel on oluline väärtuste järjekord, mis sulgude sisse kirjutatakse.

Funktsiooni töö tulemuse tagastamine

Praegu oleme näitena kasutanud selliseid funktsioone, mis kohe midagi väljastavad print() käsu abil. See ei ole aga tavaline funktsioonide kasutamine. Sageli kasutatakse funktsioone millegi välja arvutamiseks, mingi objekti muutmiseks, millegi lisamiseks ja välja trükkimisega ei ole neil suurt midagi pistmist. Kui aga funktsioon teeb mingi muutuse või arvutuse, kuidas ma muutusest teada saan, kuidas ma arvutuse kätte saan?

Funktsiooni töö tulemuse tagastamine

Selleks kasutatakse funktsiooni sees võtmesõna **return**. Return sõna taha tuleb kirjutada tulemus, mis tahetakse tagastada. Katseta kindlasti järgmist näidet:

```
import math

print("See programm arvutab silindri ruumala.")
print("Mis on silindri kõrgus (cm)?")
kõrgus = float(input())
print("Mis on silindri raadius (cm)?")
raadius = float(input())

def silindri_ruumala(h, r):
    ruumala = round(math.pi*r*r*h, 2)
    return ruumala

print("Silindri ruumala on", silindri_ruumala(kõrgus, raadius),)
```

Antud näites viimase print() käsu sees kutsutakse silindri ruumala funktsioon välja. Pane tähele, et funktsiooni argmentideks anti kasutajalt küsitud **`kõrgus`** ja **`raadius`**. Funktsiooni sees on aga argumentide nimedeks ehk parameetriteks hoopis **`h`** ja **`r`**. Funktsiooni ülesanne on ruumala välja arvutada ja anda tagasi ruumala tulemus **return** käsu abil. Seega ei prindita print käsuga välja funktsiooni nimi või tema sisu vaid **arvutatud silindri ruumala**.

Lokaalsed ja globaalsed muutujad

```
import math

print("See programm arvutab silindri ruumala.")
print("Mis on silindri kõrgus (cm)?")
kõrgus = float(input())
print("Mis on silindri raadius (cm)?")
raadius = float(input())

def silindri_ruumala(h, r):
    ruumala = round(math.pi*r*r*h, 2)
    return ruumala

print("Silindri ruumala on", silindri_ruumala(kõrgus, raadius),)
```

Antud näites on oluline tähele panna erinevaid muutuja nimesid funktsiooni sees ja väljaspool funktsiooni, kuigi tähendavad need muutujad ju mõlemal korral silindri kõrgust ja silindri raadiust. Siin tulevadki mängu mõisted **lokaalne** ja **globaalne muutuja**.

Lokaalseks muutujaks nimetakse muutujat, mis asub funktsiooni sees. Selles näites siis muutujad **h**, **r** ja **ruumala** on lokaalsed. Muutujad **kõrgus** ja **raadius** on aga **globaalsed**. Mis neil vahet on, miks kutsutakse ühte lokaalseks aga teist globaalseks?

Enne kui lähen täpse selgituse juurde, räägin sulle veidi taustast. Muutujate loomine Pythonis on seotud mälu haldamisega. Nimelt ei ole Pythonis vaja mälu peale selle kasutusele võtmist uuesti vabastada, seda teeb Python meie eest ise. Mitmes teises programmeerimiskeeles (C, C++) on vaja muutujate poolt hõivatud mälu ise vabastada. Kui seda ei tehta, võib mälu suht kiiresti täis saada ja arvuti muutub siga-aeglaseks.

Lokaalse ja **globaalse** muutuja vahe seisneb nende muutujate **mõjuala suuruses**. Pythonis on kõik funktsioonide sees olevad muutujad lokaalsed muutujad ja neid muutujaid ei saa väljaspool funktsiooni kasutada. Kui ma näiteks kirjutaksin programmi lõppu veel ühe print rea ja tema sulgudesse kirjutaksin funktsioonimuutuja **`ruumala`**, siis saaksin veateate, mis ütleb, et programm sellist muutujat ei tunne. Pythonis on nii korraldatud, et enne funktsiooni tööle hakkamist ei eksisteeri tegelikult funktsiooni sees kasutatavad muutujad. Alles siis kui funktsioon välja kutsutakse tekitab Python vastavad muutujad (lokaalsed), antud juhul **h, r** ja **ruumala** ning omistab neile seal samas väärtused. **h** saab väärtuse argumendilt kõrgus ja **r** saab väärtuse argumendilt raadius. **Ruumala** leitakse muutujate pi, h ja r korrutamise teel. Kohe kui funktsioon on tulemuse väljakutsujale ehk ruumala küsijale tagastanud, **kustutab** Python kõik funktsioonis kasutusel olnud muutujad - mälu vabastatakse ja nii muutuvadki funktsiooni muutujad programmi jaoks taas olematuks.

Globaalsed muutujad seevastu eksisteerivad kogu programmi töö vältel ja neid saab kasutada igal pool, ka funktsioonide sees.

Kui mujal programmi koodis saab globaalset muutujat iga kell muuta siis funktsiooni sees seda niisama teha ei saa, saab vaid kasutada tema väärtust. Miks nii? Kohe kui ma tahaksin näiteks funktsiooni sees globaalsele muutujale **raadius** anda uue väärtuse, näiteks nii **`raadius=20`**, siis tegelikult selline rida globaalset muutujat ei muuda, sest nagu ma ennist ütlesin, kõik muutujad, mis asuvad funktsiooni sees, luuakse uuesti, isegi kui sellel on sama nimi mis globaalsel muutujal. Sellises olukorras luuakse lihtsalt uus globaalse muutujaga samanimeline kuid lokaalne muutuja.

Globaalse muutuja muutmine funktsiooni sees

Kui mul on aga vaja ikkagi mõne funktsiooni sees muuta globaalset muutujat, siis tuleb globaalse muutuja nime ette kirjutada **global**, sellisel juhul muudab funktsioon tõesti globaalset muutujat ja ei tee valmis ajutise samanimelise lokaalse muutuja.

Pikk jutt räägitud, vaatame kuidas see muutujate asi ka reaalselt programmeerimise juures välja näeb.

Objektid

Juba teist nädalat vaatame erinevaid võimalusi kuidas organiseerida andemid ja koodilõike oma programmis. Sarnaseid muutujaid saime ühte punti siduda listide abil, koodijuppe funktsioonide abil.

Objektide mängu toomine on aga samm veel kaugemale. Objektide abil saab ühise nime alla liigitada nii funktsioone kui muutujaid **koos**. Objektid on programmeerimise maastikul väga levinud idee ja seda kasutatakse väga paljudes erinevates programmeerimiskeeltes. Nii et kui oled ühekorra ideest aru saanud, saad hakkama mistahes teise objekt-orienteeritud programmeerimiskeelega samuti.

Miks objektidest rääkida?

Kui me nüüd objektide teemaga edasi läheme, siis mõistad, miks Pythonit nimetatakse objekt-orienteeritud keeleks - siin on kõik asjad objektide alla kuuluvad, kuigi kohe alguses ei pruugi sellest arugi saada. Siiamaani polegi meie ülesannetes ühtegi kohe arusaadavat objekti ette jäänud, kuid järgmine nädal hakkame põhjalikumalt tegelema graafikaga, siis ilma objekti mõiste tundmiseta nii kergesti hakkama ei saa. Seega olgu antud peatükk sissejuhatuseks järgmisesse nädalasse.

Mis on objekt?

Võtame ühe konkreetse lihtsa näite: raamat. Me võime raamatut vaadata kui objekti. Mida see tähendab?

- Me saame objektiga raamat teha iga kord mingeid tegevusi - näiteks lugeda, kinkida, riiulisse panna, kinni panna, lahti teha jne.
- Me saame kirjeldada selle objekti omadusi - lehekülgede arv, paksud või õhukesed kaaned, mõõtmed, kuju, kirjastus, pealkiri, žanr jne.

Programmeerimises tehakse objektidega täpselt samu asju - **tegevused**, mida objektiga saab teha ja objekti **kirjeldus**. Pythonis kutsutakse andmeid, mida sa objekti kohta tead **atribuutideks** ning tegevusi, mida sa objektiga saad teha **meetoditeks**.

Objekt = atribuudid + meetodid

Näiteks kui me oleksime Pythonis loonud objekti raamat, siis raamatul võiksid olla

- sellised **atribuudid**: `raamat.lk_arv`, `raamat.suurus`, `raamat.kuju` jne.
- ning sellised **meetodid**: `raamat.lugemine()`, `raamat.avamine()`, `raamat.sulgemine()` jne.

Panid tähele erinevust ja seda punkti kasutamise asja? Tuleta meelde, me oleme ju objektidest teadmata neid võtteid näidetes kasutanud küll ja küll. Näiteks ajamõõtmise programm, kus kasutasime `time.sleep()`-i või listidesse andmete juurde panemise juures `linnad.append()`. Seega nii `time` kui `linnad` on Pythoni jaoks objektid ja punktiga taga olev käsk selle objekti meetod.

Atribuutideks saavad olla kõiksugused muutujad ja nende loetelud (arvud, kirjeldused, listid jne) ning **meetoditeks** on tavalised funktsioonid ja need kuuluvad ainult sellele konkreetsele objektile. Atribuudid on informatsioon objekti kohta ja meetodid on tegevus objektiga.

Atribuudid ei ole mitte mingil moel teistsugused kui tavalised muutujad, nendele omistatakse väärtuseid täpselt sama moodi ja neid kasutatakse koodis täpselt sama moodi kui tavalisi muutujaid, millega me juba harjunud oleme. Ainuke erinevus on selles, et atribuuti kasutatakse alati objekti nimega koos ja need on ühendatud **punktiga**.

Objekti loomine

Objekti loomine käib kahes etapis:

- Kõigepealt tuleb defineerida objekti juurde kuuluvad atribuudid ja meetodid (seda punkti võib võrrelda plaani koostamisega maja ehitamisel. Tegelikku maja veel ei ole, kuid joonis on juba valmis). Sellist plaani koostamist Pythonis kutsutakse **klassiks**.
- Teise sammuna tuleb võtta loodud **klass** (ehk plaan) ja konstrueerida selle järgi päris objekt (nagu plaani järgi maja ehitamine). Klassi järgi võib luua mitmeid ühesuguseid objekte nagu ka ühe ja sama plaani järgi võib luua mitmeid ühesuguseid maju. Klassist loodud objekte nimetatakse programmeerimises klassi **isenditeks**.

Uurime ühe klassi ja isendi loomist ning proovi ka oma Idles ühe klassi tegemisega hakkama saada.


```
class Raamat:

    def avamine(self):
        self.olek = "lahti"
        return self.olek

    def sulgemine(self):
        self.olek = "kinni"
        return self.olek
```

Kõrval olevas näites tegin ühe lihtsa klassi raamat, kus on kaks meetodit, avamine ja sulgemine, mis peaksid muutma objekti pildi ekraanil kas lahti olevaks raamatuks või kinni olevaks raamatuks.

Kuhu jäid atribuudid?

Atribuudid ei kuulu klassi juurde vaid iga objekti ehk isendi juurde eraldi. Seetõttu saab iga tüüpplaani järgi ehitatud maja olla ka erinevat värvi või erineva fassaadimaterjaliga. Iga objekt saab omada iseendale omast kirjeldust, kuid käituvad kõik ühe klassi isendid alati ühte moodi.

Objekti isendi loomine

Nagu üleval jutuks oli, siis klassi loomine on vaid plaani tegemine, nüüd tuleb ehitama hakata. Et tekitada isendit, tuleb programmis isend luua ja seda tehakse nii:

```
class Raamat:

    def avamine(self):
        self.olek = "lahti"
        return self.olek

    def sulgemine(self):
        self.olek = "kinni"
        return self.olek

jutukas = Raamat()
jutukas.olek ="kinni"
jutukas.suurus = (15, 10)

print("Tegin just ühe raamatu jutukas.")
print("Jutuka suurus on ", jutukas.suurus[0],"x",jutukas.suurus[1], "cm.")
print("Minu jutukas on praegu", jutukas.olek)
print("Avan raamatu.")
print("Jutukas on nüüd", jutukas.avamine())
print("Panen raamatu kinni.")
print("Jutukas on nüüd", jutukas.sulgemine())
```

Objekti algväärtustamine

Klassi kirjeldusse atribuute otse ei kirjutata, kuid hirmus tüütu oleks ikkagi iga objekti loomisel talle järjest argumente omistada. Selle mure saab lahendada spetsiaalse meetodi **__init__()** abil. Klassi luuakse vastav meetod **__init__()** ning selle sees on võimalik anda igale loodavale isendile nö algväärtused, algparameetrid. **__init__()** on spetsiaalne meetod (nii ees kui taga on kaks alakriipsu), mis käivitub alati, kui uus objekt luuakse, andes seeläbi objektile ka algväärtused ehk esmase kirjelduse. ***init*** tuleb inglisekeelsest sõnast *initializing*, mis tähendabki algväärtuseid andma.

Muudame raamatu näidet nii, et ei peaks eraldi objekti iseloomustama hakkama:


```
class Raamat:
    def __init__(self, olek, suurus):
        self.olek = olek
        self.suurus = suurus

    def avamine(self):
        self.olek = "lahti"
        return self.olek

    def sulgemine(self):
        self.olek = "kinni"
        return self.olek

jutukas = Raamat("kinni", (15, 10))

print("Tegin just ühe raamatu jutukas.")
print("Jutuka suurus on ", jutukas.suurus[0],"x",jutukas.suurus[1], "cm.")
print("Minu jutukas on praegu", jutukas.olek)
print("Avan raamatu.")
print("Jutukas on nüüd", jutukas.avamine())
print("Panen raamatu kinni.")
print("Jutukas on nüüd", jutukas.sulgemine())
```

Muudatuse tulemusel programmi töö ei muutunud absoluutselt, kuid pääseme vaevast igale objektile eraldi algväärtuseid andmast, sest saame need kohe objekti loomisel parameetritena sulgude sisse kirjutada.

__str__()

Pythoni klassidesse on tegelikult peidetud veel üks salajane meetod, mis sama moodi nagu __init__() meetod tehakse alati vaikimisi valmis, mis siis, et me pole seda klassi sees ise defineerinud. Selleks teiseks meetodiks on **__str__()**. Proovi, mida see meetod teeb raamatu näite korral! Selle kindlaks tegemiseks tuleb sul lihtsalt programmi lõppu kirjutada veel üks rida **`print(jutukas)`**. Peaksid saama tulemuseks midagi sellist:

```
Tegin just ühe raamatu jutukas.
Jutuka suurus on  15 x 10 cm.
Minu jutukas on praegu kinni
Avan raamatu.
Jutukas on nüüd lahti
Panen raamatu kinni.
Jutukas on nüüd kinni
<__main__.raamat object at 0x00FFBE50>
>>>
```

Viimane rida selles näites ongi käima läinud __str__() meetodi tulemus. Ta nimelt annab infot küsitud objekti kohta (*jutuka* kohta). Küsisin ma seda infot aga print käsu abil ja vaikimisi, siis kui ma ise pole öelnud, millist infot võiks objekti kohta välja anda, on selleks infoks hoopis rida, mida näed näite viimase reana. Mida see rida tähendab?

- esimene: koht, kuhu antud isend on defineeritud - antud juhul on selleks __main__, mis tähendab põhiprogrammi.
- teine: klassinimi - raamat - jutukas on objekt raamat.
- kolmas: 0x00FFBE50 tähendab mälu aadressi, kus antud objekt paikneb minu arvutis.

Suht kasutu info nö tavalisele programmi kasutajale. Seega kui ma tahan, et minu programm annaks objekti kohta mingit teist infot, näiteks objekti omaduste kirjelduse vms, siis tuleb **__str__()** meetod üle defineerida:

```
class Raamat:
    def __init__(self, olek, suurus):
        self.olek = olek
        self.suurus = suurus

    def avamine(self):
        self.olek = "lahti"
        return self.olek

    def sulgemine(self):
        self.olek = "kinni"
        return self.olek

    def __str__(self):
        teade = """Tere, mina olen objekt raamat
ja ma olen hetkel """+self.olek+".\n" \
"Minu mõõtmed on "+str(self.suurus[0]) \
+"x"+str(self.suurus[1])+"cm."
        return teade

jutukas = Raamat("kinni", (15, 10))

print("Tegin just ühe raamatu jutukas.")
print("Jutuka suurus on ", jutukas.suurus[0],"x",jutukas.suurus[1], "cm.")
print("Minu jutukas on praegu", jutukas.olek)
print("Avan raamatu.")
print("Jutukas on nüüd", jutukas.avamine())
print("Panen raamatu kinni.")
print("Jutukas on nüüd", jutukas.sulgemine())

print(jutukas)
```

self

Igal pool klassi meetodites on kasutatud sõna **self**, mida see teeb, mis see on?

Self on üks kaval sõna:) Asi on nimelt selles, et kuna klassi defineerimine tähendab ainult plaani või joonise tegemist, mille järgi objekte tegema hakatakse. Objekte aga võib olla palju ühesuguseid ainult erineva nimega. Siit tulebki välja vajadus isevärki sõna järele.

Kuidas?

Kui meil on mitu objekti raamatuid, näiteks jutukas, lastekas, romaan vms., siis iga selle objekti jaoks kehtivad täpselt ühed ja samad meetodid, mis on kirjeldatud klassis. Kust programm peab teadma meetodi käima panemisel, millise isendi parameetreid ta peaks muutma? Sellepärast ongi võetud kasutusele lisamuutuja **self**, mis saab väärtuse objektilt, kes ta välja kutsus. Kui kutsuja on `jutukas.avamine()`, siis `self=jutukas`, kui `romaan.sulgemine()`, siis `self=romaan` jne.

Muideks muutuja nime **self** asemel võib tegelikult kasutada mistahes sõna, kuid jällegi, tasub hoida kinni traditsioonidest, sest see muudab teiste koodi lugemise ja sinu koodi lugemise palju arusaadavamaks.

Miks on objektid head?

Lisaks sellele, et ma saan klassifitseerida erinevaid funktsioone ja muutujaid, siis on veel paar head asja, milleta programmeerijad enda elu ettegi ei kujuta.

Esimene neist on polümorfism:) Möh ...?

Kole mõiste, aga lihtne seletus. Ma saan kasutada täpselt sama funktsiooni nime erinevate klasside koosseisus. Oletame, et ma olen valmis teinud programmi erinevate geomeetriliste kujundite ruumalade arvutamiseks. Selles programmis on iga erineva kujundi jaoks defineeritud eraldi klass, mis on ju loogiline, igal kujundil on erinevad omadused, pindalade ja ruumalade arvutamise valemid. Kuid igas klassis saan ma kasutada ühte ja seda sama meetodi nime **`ruumala()`**, sest ta kuulub erinevatesse klassidesse. Segamini ei saa need meetodid kuidagi minna, sest neid meetodeid rakendatakse vaid oma klassi objektidele. Näiteks: **`silinder.ruumala()`**, **`püramiid.ruumala()`**, **`kuup.ruumala()`** jne., kujundi nimi enne punkti ütleb, millisest klassist **`ruumala()`** meetod tuleb välja kutsuda ja seetõttu arvutataksegi igale kujundile just temale sobilik ruumala.

Teine on aga päritavus

Objekt-orienteeritud programmeerimises saab defineerida nö **alamklasse**. Mis see annab? See annab selle, et kõige kõrgemas klassis on mul kirjas ainult need meetodid, mis kuuluvad kõikidele selle klassi objektidele, alamklassidesse aga saan ma juurde kirjutada spetsiifilisemaid meetodeid, mis kõikidel objektidel puuduvad. Näiteks:

Tihti lugu mängudes tuleb koguda igasuguseid esemeid: palle, linde, mune, münte jne. Kõik need esemed võivad kuuluda ühte suurde klassi MänguObjektid, millel on atribuut **`nimi`** (*muna, münt*) ja meetod **`korjaÜles()`**. Kuid sageli on osad esemed mängudes väärtuslikumad kui teised. Näiteks mündid, nendega saab edaspidi kindlasti mingi tehingu teha, seega müntide jaoks tuleks teha alamklass **Münt**, mis pärib kõik meetodid oma ülemuselt, kuid lisaks saame talle juurde kirjutada meetodi **`kuluta()`**, mis aitab rahaga tehinguid teha.

```
class MänguObjekt:

    def __init__(self, name):
        self.name = name

    def korjaÜles(self, mängija):
        # siia tuleb kirjutada kood, mis
        #lisab eseme mängija kollektsiooni

class Münt(MänguObjekt):

    def __init__(self, maksumus):
        MänguObjekt.__init__(self, "münt")
        self.maksumus = maksumus

    def kuluta(self, ostja, müüa):
        # siia tuleb kirjutada kood
        # mis annab mündi müüale
        #ja paneb müüalt saadud eseme
        # ostja kollektsiooni
```

Näide

Mida õppisid?

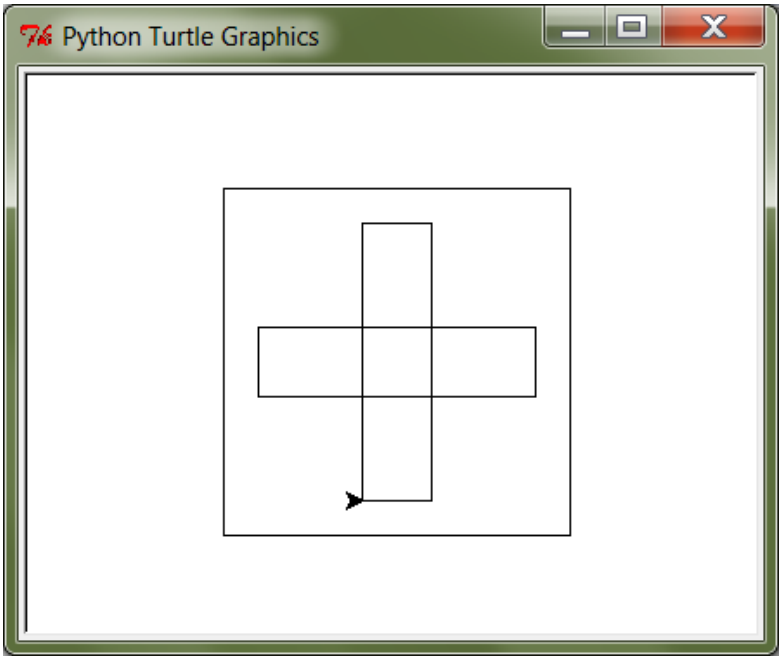
Selle nädalaga sai vast tiir peale tehtud kõigele vajalikule, et järgmine nädal asuda graafika kallale, et juba vingemaid mänge tegama saaks hakata. See nädal õppisid:

- funktsioonide loomist
- funktsioonide kasutamist
- funktsioonide välja kutsumist
- kuidas anda funktsioonile infot (argumente)?
- kuidas saada funktsiooni käest töö tulemus tagasi?
- mis vahe on lokaalsetel ja globaalsetel muutujatel?
- kuidas globaalset muutujat funktsiooni sees muuta?
- mis on objektid?
- kuidas objekte luuakse?
- mis asjad on atribuudid ja meetodid?
- mis asi on klass?
- klassi defineerima
- mis asjad on spetsiaalmeetodid __init__() ja __str__()?
- mis nähtus on polümorfism?
- kuidas luua alamklasse?

TEE ISE!

1. ülesanne: Toll->cm: Kirjuta funktsioon, mis saab argumendina pikkuse tollides ning tagastab pikkuse sentimeetrites. Kutsu funktsiooni programmis välja mitu korda erinevate väärtustega (testimiseks). Lõpuks küsi kasutajalt tema pikkust tollides ja programm peab väljastama ekraanile pikkuse sentimeetrites ning tema nn. "ideaalkaalu" (so. pikkus sentimeetrites - 100, nt. kui pikkus on 185cm, siis ideaalkaal on 85kg).

2. ülesanne: Rist: Kirjutage funktsioon `ristkylik()`, mis saab argumentideks kaks küljepikkust ja joonistab kilpkonnaga neile vastava ristküliku. Joonistage loodud funktsiooni kasutades järgnev kujund (pane tähele, et ristküliku funktsiooni kutsutakse välja 3 korda). Kasuta eelmise nädala turtle käske forward(), left(), right(), up(), down() ja kõige lõpus exitonclick(). **NB!** Segaduse vältimiseks on soovitav funktsiooni töö lõppedes pöörata kilpkonn tagasi algsesse suunda.



3. ülesanne: Objekt: Objektide loomise ja nende hingeelu paremaks tundmiseks mõtle ise välja üks objekt (kavand ehk klass), mille järgi oleks sul võimalik luua mitmeid sellele kavandile vastavaid isendeid ehk objekte. Luua tuleb midagi sarnast nagu võid vaadata näite peatükist.

4. ülesanne: **Lõputöö**: Pane kirja oma selle kursuse lõputöö idee. Kriteeriumid, millele programm peab vastama, leiad avalehelt 4. nädala alt. Idee (väike mängu kirjeldus, mida seal peab tegema, kuidas mäng töötab, millised tegelased vms ekraanil liiguvad ja kuidas?) postita 4. ülesande esitamise foorumisse,teistega jagamiseks. Masssis peitub jõud ja seetõttu võib nii mõnigi idee saada foorumi kaudu häid täiendusi või kasvõi abi ja mõtteid realiseerimiseks.

5. ülesanne: Taas anna mulle tagasisidet, kuid seekord kursusel üldiselt toimetuleku kohta. Uuri küsimärgi alt.