6.1 TSÜKKEL JA REKURSIOON

Eelmisel nädalal tutvusime rekursiooniga. Rekursiivne funktsioon kutsub välja iseenda ja see võib toimuda korduvalt. Varem olime korduva tegevuse puhul kasutanud tsükleid. Selles peatükis püüamegi tsüklit ja rekursiooni kõrvutada - nii mõnegi ülesande lahendame mõlemal moel.

Millal lõpetada?

Korduva tegevuse puhul on oluline piiri pidada - millal ikkagi lõpetada? Tsükli puhul on jätkamise või lõpetamise mõttes tähtis tsükli jätkamistingimus. Tsüklit korratakse seni kui jätkamistingimus veel kehtib. While-tsükli puhul on see tingimus selgelt nähtav, näiteks

```
while i < len(järjend):
```

For-tsükli puhul antakse ette teatud kogum elemente, millest iga puhul tuleb tsükli sisu täita. Tingimus siis kontrollib, kas on veel mõni element, mis on vaatamata, näiteks

```
for  i in range(len(järjend))
```

või

```
for element in järjend:
```

Rekursiooni korral kontrollitakse, kas järjekordse väljakutsega on juba jõutud rekursiooni baasini - juhuni, millel on mitterekursiivne lahendus.

Tuletame meelde faktoriaali arvutamise funktsiooni, kus baas on juhul n = 0.

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

Tingimust sobivalt vastupidiseks muutes saab harud ka ära vahetada.

```
def faktoriaal2(n):
    if n != 0:
        return n * faktoriaal2(n - 1)
    else:
        return 1
```

Antud juhtudel võib `else`-osa sisu lihtsalt tingimuslause järele kirjutada, sest `if`-osa `return` nagunii ei luba tingimuslausest edasi.

```
def faktoriaal(n):
    if n == 0:    #Rekursiooni baas
        return 1
    return n * faktoriaal(n-1) #Rekursiooni samm
def faktoriaal2(n):
    if n != 0:
        return n * faktoriaal2(n - 1)
    return 1
```

Akumulaator

Faktoriaali saab muidugi arvutada ka tsükli abil. Seda saab teha mitmel moel, alustame `for`-tsükliga ja nii, et arvutame kasvavate teguritega 1 · 2 · 3 … n

```
def faktoriaal_tsükliga_for_üles(n):
    aku = 1
    for i in range(1, n + 1): #1, 2 … n
        aku *= i
    return aku
```

Edasi jõuab samale tulemusele hoopis `while`-tsükkel kahanevate teguritega n · (n - 1) · … 1

```
def faktoriaal_tsükliga_while_alla(n):
    aku = 1
    while n != 0:
        aku *= n
        n -= 1
    return aku
```

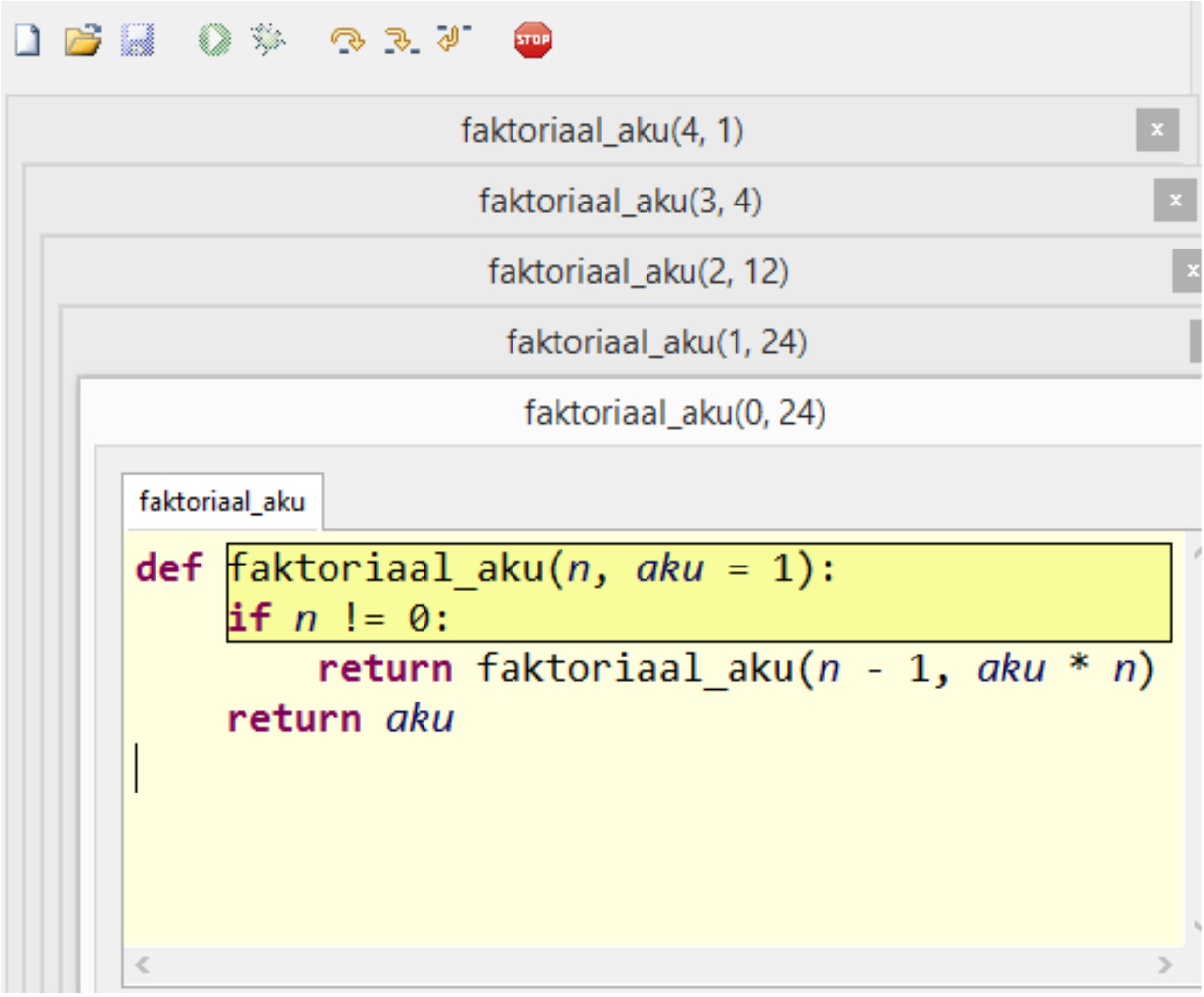
Mõlemal juhul kasutatakse muutujat `aku`, millesse lõpptulemus järjest kogutakse. Vahel nimetatakse sellist muutujat *akumulaatoriks*. Põhimõtteliselt saab akumulaatorit kasutada ka rekursiooni puhul. Nimelt saame korraldada, et vajalik väärtus antakse edasi argumendi ja tagastamise kaudu. Selleks on kasutusel teine argument, mille nimeks panemegi aku.

```
def faktoriaal_aku(n, aku):
    if n != 0:
        return faktoriaal_aku(n - 1, aku * n)
    return aku
```

Kui nüüd see funktsioon välja kutsuda, siis tuleb `aku` esimene väärtus väljakutses ise määrata.

```
print(faktoriaal_aku(4, 1))
```

Kasutades Thonny võimalusi programmi töö läbimängimiseks (*Debug current script*) näeme, et rekursioonile kohaselt kutsutakse funktsiooni järjest välja ja teine argument tõesti "kogub" vajaliku tulemuse endasse.



Kui teraselt pilti jälgida, siis pole seal päris täpselt see funktsioon, mis ülaltoodud. Nimelt on siin hea kasutada Pythoni võimalust anda funktsiooni argumendile vaikeväärtus.

```
def faktoriaal_aku(n, aku = 1):
```

Vaikeväärtust kasutatakse siis, kui sellele argumendile väljakutses väärtust ei anta. Nii vabaneme vajadusest akumulaatori esialgne väärtus ette anda.

```
print(faktoriaal_aku(4))
```

Sabarekursioon

Kui nüüd jõutakse baasini, siis hakatakse seda leitud tulemust, antud juhul 24, järjest tagastama. Oluline on, et midagi rohkemat tagasiteel ei juhtugi - `return` tagastabki igal tasemel selle, mida ta sügavamalt saab

```
return faktoriaal_aku(n - 1, aku * n)
```

Mäletatavasti meie algses rekursiivses faktoriaali leidmise funktsioonis just oluline arvutamine toimuski.

```
return n * faktoriaal(n - 1)
```

Kui kõik tegevused toimuvad enne rekursiivset väljakutset, siis nimetatakse rekursiooni *sabarekursiooniks* (*tail recursion*). Funktsiooni nimetatakse *sabarekursiivseks*, kui pärast tagastusväärtuse saamist tehakse ainult väärtuse tagastamine.

Sabarekursiooni on suhteliselt lihtne teisendada tavaliseks tsükliks (ja ka vastupidi).

Funktsioon

```
def faktoriaal_aku(n, aku = 1):
    if n != 0:
        return faktoriaal_aku(n - 1, aku * n)
    return aku
```

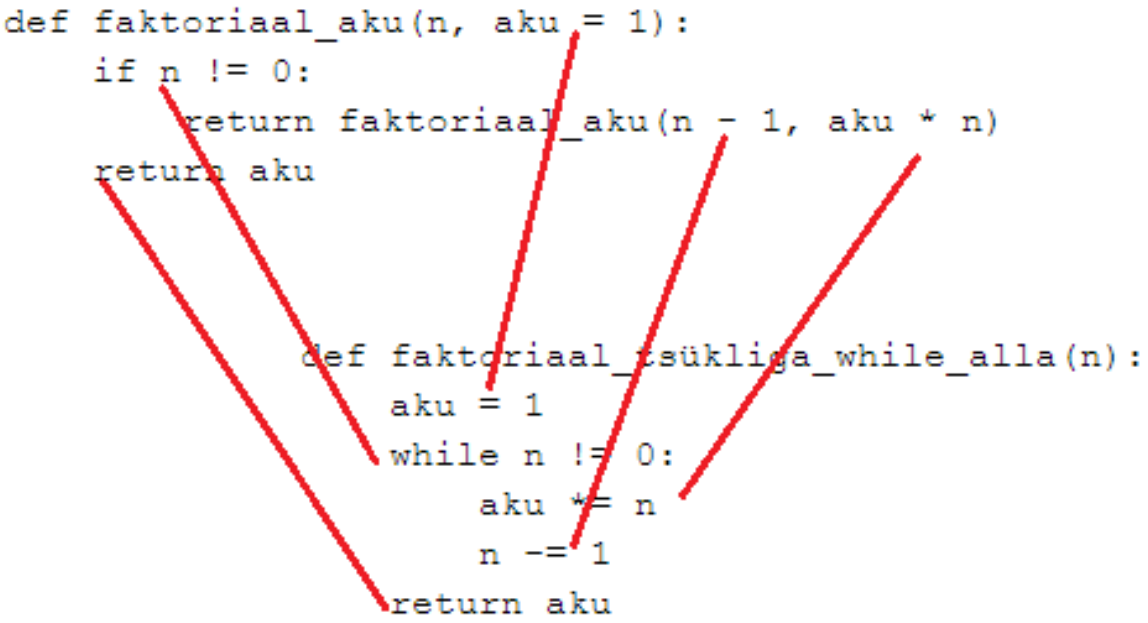
on niisiis sabarekursiivne.

Kuidas aga siis on lood tsükliks teisendamisega?

Võtame võrdluseks `while`-tsüklit kasutava funktsiooni.

```
def faktoriaal_tsükliga_while_alla(n):
    aku = 1
    while n != 0:
        aku *= n
        n -= 1
    return aku
```

Näeme, et tõesti ongi kõik praktiliselt samal kujul olemas nii ühes tsükli kui rekursiooni variandis.



Järjendi summa näide

Vaatleme nüüd järjendi elementide summa arvutamise ülesannet. Loomulikult on Pythonis juba olemas täpselt selleks ettenähtud funktsioon `sum`, aga "unustame" selle hetkeks ära. Summat saame nii tsükliliselt kui rekursiivselt leida väga mitmel moel. Esialgu toome mõned tsükliga variandid, näiteks `for`-tsükliga üle elementide

```
def summa_tsükkel_element(järjend):
    tulemus = 0
    for element in järjend:
        tulemus += element
    return tulemus
```

ning `for`-tsükliga üle indeksite

```
def summa_tsükkel_indeks(järjend):
    tulemus = 0
    for i in range(len(järjend)):
        tulemus += järjend[i]
    return tulemus
```

Nüüd aga kirjutame selle viimase ümber `while`-tsükliga variandiks.

```
def summa_tsükkel_i_while(järjend):
    tulemus = 0
    i = 0
    while i < len(järjend):
        tulemus += järjend[i]
        i += 1
    return tulemus
```

Seda aga saab suhteliselt lihtsalt teisendada sabarekursiivseks.

```
def summa_sabarekursiivne(järjend, i = 0, tulemus = 0):
    if i < len(järjend):
        return summa_sabarekursiivne(järjend, i + 1, järjend[i] +
tulemus)
    return tulemus
```

Muutujat `i` oli meil vaja, et teada, mis kohas me järjendis parajasti oleme. Ilma selleta saame siis hakkama, kui pärast järjendi alguselemendi (`järjend[0]`) liitmise järel edasi juba sellise järjendiga tegutseme, kus alguselement on välja jäetud (`järjend[1:]`)

```
def summa_while_lõige(järjend):
    tulemus = 0
    while len(järjend) > 0:
        tulemus += järjend[0]
        järjend = järjend[1:]
    return tulemus
def summa_sabarekursioon_lõige(järjend, tulemus = 0):
    if len(järjend) > 0:
        return summa_sabarekursioon_lõige(järjend[1:], järjend[0] +
tulemus)
    return tulemus
```

6.2 REKURSIOONI TÜÜPE. HARGNEV REKURSIOON

Sissejuhatus

Eelmises peatükis vaatasime näiteid, kus sama ülesanne oli lahendatud kord tsükliga, kord rekursiooniga. Me ei rääkinud sellest, kumba varianti on mõistlikum kasutada. Siin saab eristada programmi kirjutamise ja programmi täitmise aspekti. Programmi täitmise mõttes kipub tsükliga programm rekursioonist kiirem olema. Programmi kirjutamise mõttes aga võib vastavalt ülesandele tsükkel või rekusioon mugavam olla. Mõnedes eeltoodud näidetes võis rekursioon olla küllaltki kunstlik, samas faktoriaali puhul oli rekursioon üsna loomulik, eriti kui valem juba sellisel rekursiivsust soosival kujul oli. Edasi vaatleme veel näiteid, kus programmi on mugavam rekursiivselt kirjutada.

Alustame kahendotsingust, seejärel vaatame hargnevat rekursiooni ja vastastikkust rekursiooni. Senistes näidetes kutsuti funktsiooni kehas seda funktsiooni ennast välja ülimalt üks kord. Tegemist oli *lineaarse rekursiooniga*, kuna rekursiivsete väljakutsete kontrollvoog oli lineaarne ahel. Võimalikud on aga ka keerukamad rekursiivse väljakutse mustrid. Näiteks *hargneva rekursiooni* e *puurekursiooni* puhul kutsutakse funktsiooni "samal tasemel" rekursiivselt välja rohkem kui üks kord. *Vastastikuse rekursiooni* korral kutsub üks funktsioon küll teist, see teine aga jälle esimest välja.

Kahendotsing

Kellegi või millegi otsimine on elus üsna kesksel kohal. Erinevatel eluetappidel võivad otsingute eesmärgid olla erinevad. Võib-olla sellest materjalilõigust pole otsest abi elu mõtte, armastuse, kurja juure, lohutuse, vea, töökoha otsimisel, aga kaudselt äkki on. Näiteks soovitus püüda vältida otsimist sealt, kus otsitavat kindlasti pole, on ju üsna universaalne.

Olgu meil täisarvude järjend, mille elemendid on sorteeritud mittekahanevalt. Otsimise eesmärke saab erinevaid püstitada, näiteks

- kas etteantud arv on järjendis;
- millises kohas etteantud arv järjendis asub;
- millise elemendi ette etteantud arv järjendisse sobib.

Meie funktsioon otsib, millise indeksiga elemendi ette sobib etteantud arv. Kui on kõigist suurem, siis tagastada indeks, mis võrdne järjendi pikkusega.

Kasutamegi siin ideed, et otsime ainult sealt, kust on mõtet otsida. Nimelt muutujad `algus` ja `lõpp` näitavad, millises järjendi osas otsida tuleb. Esialgu on piirkonnaks

kogu järjend. Edasi aga leitakse keskel asuv elemendi indeks (`kesk = (algus + lõpp) // 2`) ning vastavalt sellele, kumba poolde etteantud arv kuulub, jätkatakse just seal otsimist. Igal järgmisel rekursiivsel väljakutsel muutub piirkond väiksemaks, kuni lõpuks on vaatluse all üks element. See ongi rekursiooni baasiks.

```
def otsi_koht(järjend, x, algus, lõpp):
    if algus == lõpp: #vaatluse alla on jäänud üks element
        if järjend[algus] < x:
            return algus + 1 #
        else:
            return algus
    kesk = (algus + lõpp) // 2
    if järjend[kesk] < x:
        return otsi_koht(järjend, x, kesk + 1, lõpp)
    else:
        return otsi_koht(järjend, x, algus, kesk - 1)

a = [1, 4, 5, 6, 7, 7, 23, 45]
print(otsi_koht(a, 7, 0, len(a) - 1))
```

Hargnev rekursioon e puurekursioon

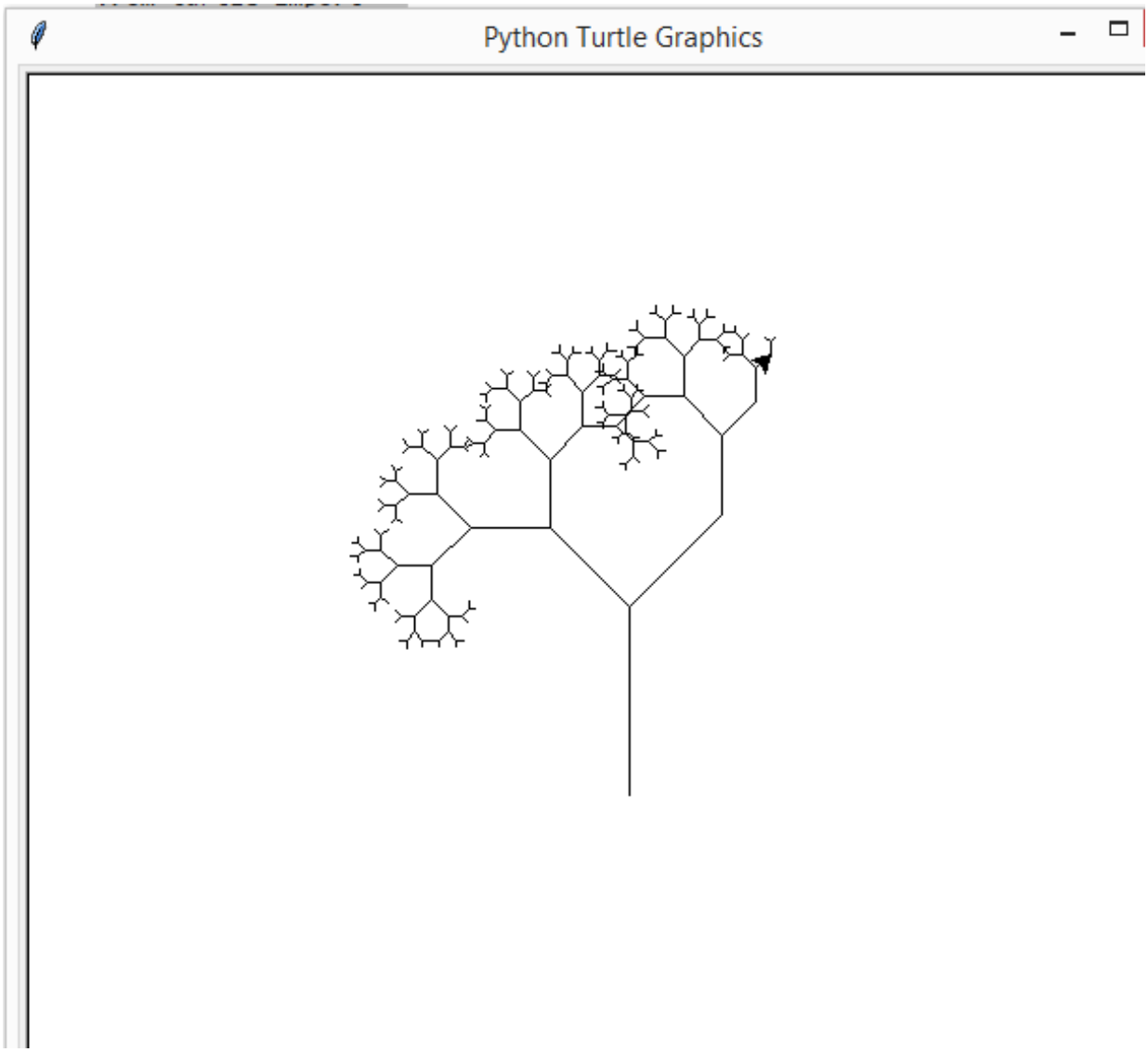
Vahel öeldakse, et rekursiooni on üldse mõtet kasutada, kui ühel tasemel kutsub funktsioon end välja rohkem kui üks kord.

Kui samal tasemel on rohkem kui üks rekursiivne väljakutse, siis programmi töö graafiline esitus meenutab puud. Näitlikustamiseks võimegi vaadelda puud joonistavat funktsiooni.

```
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)
```

Tõepoolest on siin samal tasemel kaks rekursiivset väljakutset `puu(randint(6, 7) / 10 * pikkus)`. Antud juhul on need täpselt samasugused, aga üldiselt ei pruugi olla.

Järgmisel joonisel on puu joonistamine veel pooleli.



Rekursiooni paremaks mõistmiseks võiks järgmise funktsiooni töö käsitsi (või Thonnyga) sammhaaval läbi mängida.

```
def rekFun(x, y):
    print(y)
    if x < 0:
        return y
    else:
        return rekFun(x - 1, y + 2) + rekFun(x - 2, y + 3)

print(rekFun(2, 0))
```

Fibonacci arvud

Sageli tuuakse hargneva rekursiooni näitena Fibonacci arvude arvutamise. Vaatleme seda meiegi.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Näeme, et siingi kutsutakse funktsioon samal tasemel rekursiivselt kaks korda välja.

Fibonacci arvudega on seotud ka küülikute paljunemise ülesanne. Küülikupaaril sünnib iga kuu kaks järglast (emane ja isane). Kahe kuu vanusena hakkavad küülikud ise järglasi saama. Kui palju on küülikuid 12 kuu pärast, kui algul oli üks paar?

Fibonacci arvud on seotud ka kuldlõikega ja taimede ning käbide ehitusega.

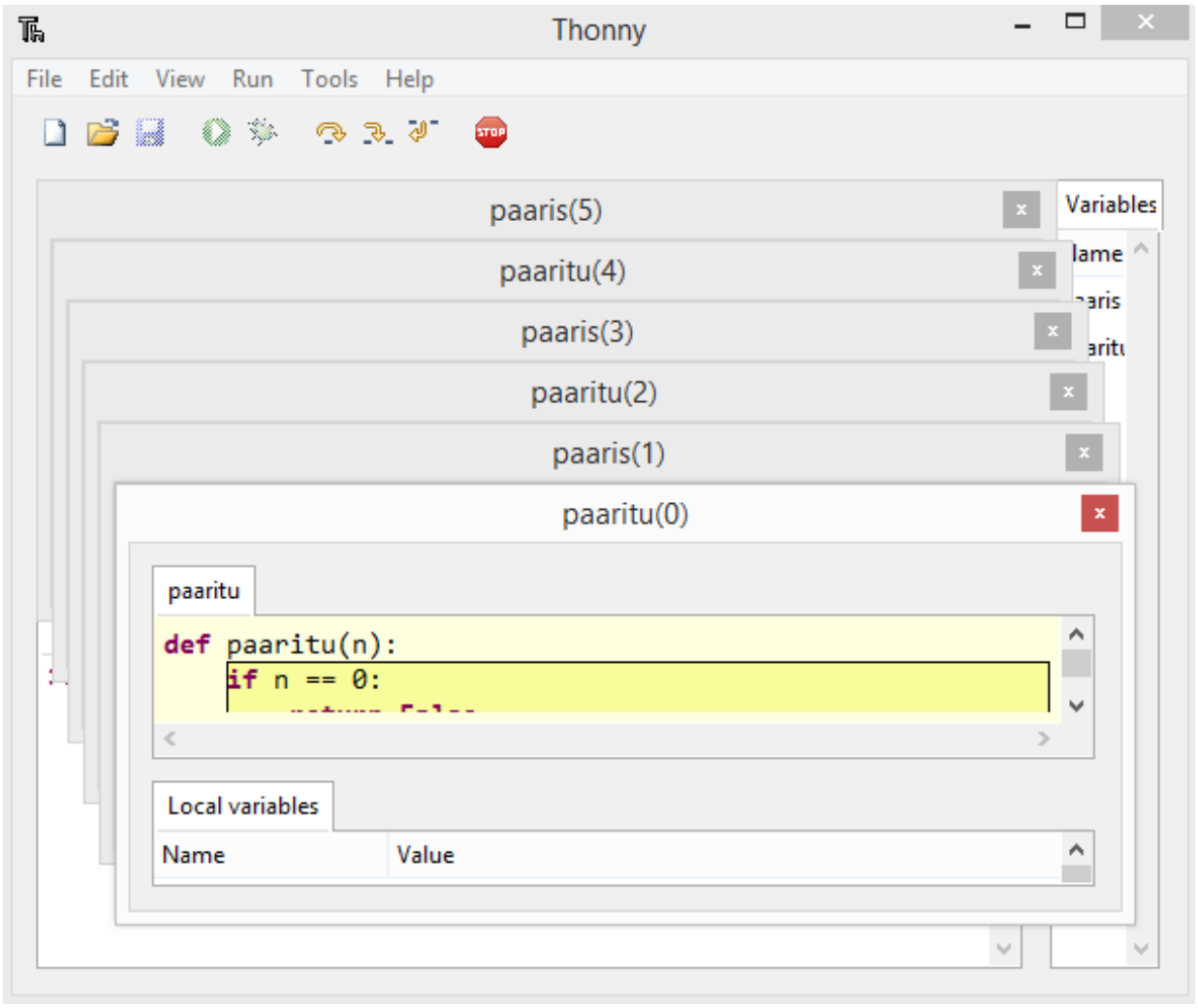
Vastastikune rekursioon

Vastastikuse rekursiooni korral ei kutsugi funktsioon otse end rekursiivselt välja. Välja kutsutakse teine funktsioon, mis aga omakorda kutsub välja esimese. Järgmine näites liigutakse baasi poole funktsioone pidi kordamööda.

```
def paaris(n):
    if n == 0:
        return True
    else:
        return paaritu(n - 1)

def paaritu(n):
    if n == 0:
        return False
    else:
        return paaris(n - 1)
```

Kui seda programmi `paaris(5)` korral Thonnyga läbi mängida, näeme järgmist pilti.


Thonny
File Edit View Run Tools Help
paaris(5)
paaritu(4)
paaris(3)
paaritu(2)
paaris(1)
paaritu(0)
Variables
paaritu
def paaritu(n):
if n == 0:
Local variables
Name
Value

6.3 REKURSIIVSED ANDMESTRUKTUURID

JÄRJENDID JÄRJENDITE SEES

See, et järjendi elemendiks võib olla järjend, ei ole meile uudiseks - oleme ju pikalt tegutsenud kahemõõtmeliste järjenditega. Põhiliselt olid meil analüüsimisel tabelid (maatriksid), mille igas reas oli võrdselt elemente. Tegelikult võivad read olla erineva pikkusega. Mitmed tollased programmid töötaksid ka selliste kahemõõtmeliste järjenditega. Igatahes aga oli meil alati kaks mõõdet - "välimine" järjend koosnes järjenditest. "Sisemistes" järjendites olid aga juba nt arvud. Kui sisemised järjendid oleksid olnud veel omakorda järjendid - oleks tegemist kolmemõõtmelise järjendiga.

Nüüd aga võtame vaatluse alla sellised andmestuktuurid, kus me mõõtmete arvu ei tea. Näiteks olgu meil järjend, mille iga element on täisarv või järjend, milles omakorda on iga element täisarv või järjend, milles omakorda on iga element täisarv või järjend, milles omakorda on iga element täisarv või järjend ...

Näiteks on mõned sellised:

```
[1, [2], [3, 4]]
[1, [2, [[3, 4], [5, 6]]], [7, 8, 9]]
[1, 2, 3, 4]
```

Sääraste andmestruktuuride käsitlemisel on rekursioon suureks abiks. Järgmine funktsioon tagastab kõigi argumendina antud struktuuris oleva arvude summa olenemata sellest, kui sügaval oleva listi sees see arv asub.

```
def summa(struktuur):
    tulemus = 0
    for element in struktuur:
        if isinstance(element, list):
            tulemus += summa(element)
        else:
            tulemus += element
    return tulemus

print(summa([1, [2], [3, 4]]))
print(summa([1, [2, [[3, 4], [5, 6]]], [7, 8, 9]]))
print(summa([1, 2, 3, 4]))
```

Funktsioon `isinstance` abil kontrollitakse, kas vaatluse all olev element on järjend (list). Kui on, siis peame rekursiivselt sügavamale minema. Kui pole, saame liita.

FAILIDE JA KAUSTADE KUVAMINE

Rekursiivse struktuuri moodustavad ka näiteks kaustad ja failid kõvakettal vm. Kaustas võivad olla nii failid, kui kaustad, milles võivad olla failid või kaustad, milles omakorda võivad olla failid või kaustad ...

Simuleerime kaustade ja failidega tegutsemist järgmise ülesandega.

Koostada rekursiivne funktsioon, mis läbib rekursiivselt argumendiks antud mitmemõõtmelise järjendi kõik elemendid. Kui järjendi element on järjend, siis trükitakse ekraanile sõna „Kaust“ ja järjekorranumber (alustades 1-st). Vastasel juhul sõna „Fail:“ ja selle järel failinimi. Kusjuures väljatrükil peab kasutama taanet, et kaustade sisust oleks võimalik visuaalselt aru saada.

Täpsustused. Funktsioon saab argumentidena ette järjendi ja taande, mille vaikeväärtus võib tühi sõne olla.

Näiteks järjendi

```
[["Fail11"], ["Fail21"],
  [["Fail311"], "Fail31", "Fail32", "Fail33"],
  "Fail1", "Fail2"]
```

korral on võimalik väljund

```
Kaust 1:
 Fail: Fail11
Kaust 2:
 Fail: Fail21
Kaust 3:
 Kaust 1:
     Fail: Fail311
 Fail: Fail31
 Fail: Fail32
 Fail: Fail33
Fail: Fail1
Fail: Fail2
```

Taandena võib kasutada tabulaatorit – "\t". Seda, kas element on järjend saame, nagu juba eespool mainitud, teada funktsiooni `isinstance` abil. Alltoodud videos on selleks kasutatud funktsiooni `type`.

Video: https://www.youtube.com/watch?v=Q8Oj5ObzexY

Päris kaustade ja failidega tegutsemiseks on Pythonis olema spetsiaalsed vahendid. Näiteks kaustas olevate failide ja kaustade nimed saab järjendina os.listdir abil. Seda, kas on tegemist

kaustaga (ja mitte failiga), saab kontrollida os.path.isdirabil. Olge aga päris kaustade ja failidega tegutsemisega ettevaatlik, olemas on ka funktsioonid kustutamiseks ...

6.4 HANOI TORNID JA MERSENNE'I ARVUD

Hanoi tornid

Selles silmaringi materjalis vaatleme mängulist, aga väga õpetlikku näidet, mis on seotud rekursiooniga. Tegemist on Hanoi tornide ülesandega. Nimelt on erineva läbimõõduga kettad nii laotud, et kõige suurem on kõige alumine ja väiksemad järjest üksteise peal kuni kõige väiksemani. Võimalik, et ketastes on augud ja neist on varras läbi, aga see pole hetkel meile tähtis.

Eesmärgiks on laduda kettad uuele platsile (vardale) ümber nii, et need oleksid jälle samal moel. Ühel sammul võib liigutada ainult ühte ketast ja mitte kunagi ei tohi suuremat ketast panna väiksema peale. Ümbertõstmisel saab kasutada kolmandat platsi (varrast).

Hanoi tornide ülesannet tutvustas prantsuse matemaatik Edouard Lucas 1883. aastal. Selle ülesandega on seotud erinevad legendid. Näiteks olla Indias ühes templis niimoodi paigutatud 64 kuldset ketast. Või olid sellised kettad hoopis Vietnamis Hanois?

Hanoi tornide ülesannet tuuakse väga sageli rekursiooni näiteks. Nimelt on rekursiivne lahendus intuitiivsem kui tsükliline. Püüamegi nüüd Hanoi tornide ülesande rekursiivse lahenduse läbi mõelda. Nagu ikka rekursiivse lahenduse puhul peab rekursiivne väljakutse olema lihtsama juhu jaoks. Praegusel juhul siis ilmselt väiksema ketaste arvu jaoks.

Võime mõelda nii, et kõige suurema ketta ümber tõstmine õigesse kohta saab toimuda ainult siis, kui kõik ülejäänud on vaheplatsile tõstetud.

Nende kõigi ülejäänute vaheplatsile tõstmine aga ongi ju sama ülesanne ühe võrra väiksema ketaste arvuga. Ainult sihtplatsi rollis on nüüd kolmas n-ö ajutine plats. Nüüd aga tuleb need vaheplatsilt kõik kenasti sihtplatsile suurima peale tõsta, mis on muidugi ka jälle sama funktsiooni väljakutse. Vaheplatsi rollis on aga esimene plats.

Järgnev programm näitabki, kuidas kettaid tõstetakse.

```
def hanoi(arv, lähe, siht, ajutine):
    if arv > 0:
        hanoi(arv - 1, lähe, ajutine, siht)
        print("Ketas liigutati platsilt " + str(lähe) + " platsile "
+ str(siht))
        hanoi(arv - 1, ajutine, siht, lähe)


hanoi(3, "A", "B", "C")
```

Kirjutame nüüd programmi ümber nii, et platside kettad oleksid eraldi järjendites.

```
def hanoi_järjend(arv, lähe, siht, ajutine):
    if arv > 0:
        hanoi_järjend(arv - 1, lähe, ajutine, siht)
        liigutatav = lähe.pop()
        siht.append(liigutatav)
        print("Ketas " + str(liigutatav) + " liigutati platsilt " +
str(lähe) + " platsile " + str(siht))
        hanoi_järjend(arv - 1, ajutine, siht, lähe)

hanoi_järjend(5, [5, 4, 3, 2, 1], [], [])
```

Mersenne'i arvud

Kui loendada, mitu liigutamist need programmid ketaste paigutamisel teevad, siis saame

- 1 ketta puhul 1 liigutamise, lihtsalt tõstetaksegi paika;
- 2 ketta puhul 3 liigutamist, väiksem vaheplatsile, suurem paika, väiksem tema peale;
- 3 ketta puhul saame 3 liigutamisega 2 väiksemat ketast vaheplatsile ja siis suurim paika ja siis jälle 3 liigutamist, et 2 väiksemat tema peale saada.

Kui kettaid on n tükki, siis liigutamisi tehakse $2^n - 1$. See on ka teoreetiliselt kõige väiksem liigutamiste arv.

Enne oli meil juttu Fibonacci arvudest. Selgub, et ka arvudele, mis arvutatakse $2^n - 1$ abil, kus n on naturaalarv, on nimi pandud. Neid arve nimetatakse *Mersenne'i* arvudeks. Mersenne'i arvud on 1, 3, 7, 15, 31, 63, 127 ...

Mersenne'i arvude hulgas paistab olevat üsna mitmeid algarve. Algarv on ühest suurem naturaalarv, mis jagub ainult 1 ja iseendaga. Eeltoodud arvudest on algarvud 3, 7, 31, 127. Prantsuse teadlane Marin Mersenne 17. sajandi algul just selliseid algarve uuriski, mis on kujul $2^n - 1$. Tema auks nimetatigi $2^n - 1$ kujul arvud Mersenne'i arvudeks ja neist algrarvud Mersenne'i algarvudeks.

Viimasel ajal leitud suurimad algarvud on kõik olnud Mersenne'i algarvud. Hetkel teadaolevatest suurim algarv on $2^{74207281} - 1$. Tõestatud on, et algarve on lõpmatult palju. Põnevat infot Mersenne'i algarvude kohta on vikipeedias. Näiteks on seal Mersenne'i algarvude pikkused, leidmise ajad, leidjad.