5.1 TESTIMINE JA SILUMINE

Tarkvara kvaliteet

Nii lühemate kui pikemate programmide puhul on vaja kindel olla, et programm ikka töötab nii nagu nõuded ette näevad ja kasutajad ootavad. Nõuetele ja ootustele vastavuse mõõtu kutsutakse *tarkvara kvaliteediks*. Tarkvara kvaliteedil võib olla ka teisi tähendusi.

Nõudeid ja ootusi võib olla väga erinevat laadi. Nõuded võivad olla näiteks järgmistes aspektides: hõlpsus (*accessibility*), ühilduvus (*compatibility*), konkurrentsus (*concurrency*), tõhusus (*efficiency*), funktsionaalsus (*functionality*), installeeritavus (*installability*), hooldatavus (*maintainability*), jõudlus (*performance*), porditavus (*portability*), töökindlus (*reliability*), turvalisus (*security*), testitavus (*testability*), kasutatavus (*usability*). Siintoodud terminid on vastavuses Standardipõhise tarkvaratehnika sõnastikuga.

Erinevates kohtades võidakse erinevaid aspekte erinevalt sõnastada, rõhutada ja rühmitada. Näiteks *Tarkvara kvaliteedimudelite standardis* ISO/IEC 25010 on esimesel tasemel eristatud kaheksat karakteristikut: funktsionaalne sobivus (*functional suitability*), soorituse tõhusus (*performance efficiency*), töökindlus (*reliability*), ühilduvus (*compatibility*), kasutatavus (*operability*), turvalisus (*security*), hooldatavus (*maintainability*), porditavus (*portability*).

Osa eelnevatest terminitest on intuitiivselt paremini mõistetavad, osa vajaks ilmselt põhjalikumat selgitust. Selles kursuses aga keskendume vaid funktsionaalsusele, sest meile on esmajoones tähtis, et programm teeks seda, mida vaja.

Erinevate aspektide kvaliteedi hindamiseks ja ebakohtade leidmiseks kasutatakse erinevaid metoodikaid. Vahel on vaatluse all kogu süsteem tervikuna. Teatud juhtudel on aga mõistlik vaadata erinevaid osi eraldi. Tarkvara arendamisel on sageli suur osa kvaliteedi kontrollist otsesest programmeerimisest lahku viidud - sellega tegelevad spetsiaalsed inimesed (osakonnad). Samuti on tarkvara otsene programmeerimine ise tavaliselt jaotatud erinevate inimeste vahel.

Kui üks inimene ühe programmilõigu kirjutab, on ta harilikult ise ka esmane kontrollija. Seda olete meie kursuselgi korduvalt ülesandeid lahendades teinud.

Edasi vaatame, kuidas süstemaatiliselt kontrollida programmi funktsionaalsust - kuivõrd programm teeb seda, mis nõutud.

Testimine

Tarkvara kontrollimisel on tähtis osa *testimisel* (kontrollimisel kasutatakse nt ka läbivaatust, tõestamist ja muid meetodeid). Testimisel on erinevaid definitsioone. Siin võtame testimist kitsalt kui programmi käivitamist nõuetele vastavuse kontrollimiseks ja vigade leidmiseks.

Üldjuhul avastatakse testimise käigus suurem osa programmis leiduvatest vigadest. Mida põhjalikum on testimine, seda kvaliteetsem on lõpptulemus. Testimine algab tegelikult juba programmi kirjutamisel. Iga kord kui programmi käivitame ja proovime, kas näiteks äsjakirjutatud funktsioon annab loodetud tulemuse või kas programm üldse käivitub, tegelemegi juba testimisega.

Testimine on oluline tarkvara arenduse erinevatel etappidel, seejuures testitakse tarkvara nii tervikuna kui ka osade kaupa. Ühe konkreetse üksuse (nt funktsiooni) testimist nimetatakse *ühiktestimiseks* (*unit testing*). Ühiktestimiseks võivad programmeerimiskeeltes olla spetsiaalsed vahendid. Näiteks Pythonis võib selleks kasutada lihtsat `assert` lauset:

```
def paaris_liitmine(x):
    if x % 2 == 0:
        return x + 1
    return x

assert paaris_liitmine(1) == 1
assert paaris_liitmine(0) == 1
```

Näitena vaatleme funktsiooni `paaris_liitmine(x)` testimist. Olgu nõuetes kirjas, et funktsioon peab paarisarvulise argumendi korral tagastama sellest argumendist ühe võrra suurema arvu, vastasel juhul argumendi enda.

Märksõnale `assert` järgneb tingimus, mis peab tõene olema. Kui tingimus on tõene, siis programm ei tee mitte midagi; kui tingimus on väär, siis antakse veateade (`AssertionError`).

Nii saab funktsiooni tööd kontrollida mitmete erinevate argumentidega. Kui testitav funktsioon peaks mingil hetkel muutuma, kuid selle jaoks on testid (`assert`-laused) alles, siis saab nende abil automaatselt funktsiooni uuesti testida.

Pythonis on olemas ka moodul unittest, mis võimaldab paindlikumat testimist, kuid nõuab algteadmisi objekt-orienteeritud programmeerimisest:

```
import unittest

def paaris_liitmine(x):
    if x % 2 == 0:
        return x + 1
    return x

class UusTest(unittest.TestCase): # Uus klass
    def test(self):
        self.assertEqual(paaris_liitmine(1), 1) # 1. testjuht
        self.assertEqual(paaris_liitmine(0), 1) # 2. testjuht

if __name__ == '__main__':
    unittest.main()
```

Klassis `UusTest` on funktsioon `test`, milles on omakorda kaks testjuhtu (*test case*).

Programmi käivitades ilmub ekraanile muuhulgas

```
Ran 1 test in 0.000s

OK
```

Proovige funktsiooni `paaris_liitmine` muuta nii, et kontrollis "läbi kukutakse". Näiteks

```
def paaris_liitmine(x):
    if x % 2 == 0:
        return x + 2
    return x
```

Proovige ka funktsiooni, mis olenemata argumendist tagastab arvu `1`. Kas test läbitakse?

```
def paar_liitmine(x):
    return 1
```

Lähemalt testimisest saab lugeda näiteks Python dokumentatsioonist.

Testjuhud

Testjuhud koosnevad sisendandmetest ja nendele vastavatest oodatavatest väljundandmetest. Eelnevas näites olid testjuhud, kus

- sisend oli 0 ja sellele vastav väljund 1 ning
- sisend oli 1 ja sellele vastav väljund 1.

Testimist peaks korraldama süstemaatiliselt. Selles on väga oluline mõistlike *testjuhtude* komplekti moodustamine. Täielikult süstemaatilise lähenemise kõrval on siiski omal kohal ka vähem süstemaatilised võtted. Näiteks *suitsutesti* (*smoke testing*) korral kontrollitakse vaid, kas programmi põhiasjad töötavad. Nimetus on pärit ammuste aegade elektroonika seadmetest, kui vigade korral päriselt suitsu võiski tulema hakata.

Kuna programmeerija ise on oma programmiga liiga seotud, siis tema enda koostatud testjuhud ei pruugi olla piisavalt head. Võimalik, et ka Teil on juhtunud, et enda arvates ju programm töötab hästi, aga automaatkontroll toob veel välja juhte, mille puhul programm õigesti ei tööta.

Nii programmeerijal endal kui ka teistel testijatel (testjuhtude koostajatel) on kasulik teada, mis alusel testimist läbi viia. Eristada saab näiteks *valge kasti*, *läbipaistva kasti* (*white box*) meetodeid ja *musta kasti* (*black box*) meetodeid. Nagu tänapäeval paljude asjadega, on siingi võimalikud *halli kasti* (*grey box*) meetodid.

Musta kasti meetodi puhul ei vaadelda üldse programmi teksti ja testjuhud on valitud ainult nõuete põhjal. Programmi käsitletakse kui "musta kasti", teadmata midagi selle sisemisest tööst. Testija sisestab mingid andmed ja saab vastuseks tulemuse, teadmata, mida andmetega vahepeal tehakse ning kontrollib, kas tulemus ja programmi käitumine on ootuspärane. Valge kasti meetodi korral vaadeldakse ka programmi ja testjuhtude koostamisel arvestatakse muuhulgas konkreetse testitava programmiga.

Programmi võimalike sisendandmete hulk on tavaliselt väga suur ja programmi testimine kõigil võimalikel juhtudel võimatu. Niisiis tuleb teha mõistlik valik.

Silumine

Vigade leidmisel peaks need ka kõrvaldama. Vigade kõrvaldamise protsessi nimetatakse *silumiseks* (*debugging*). Legendi järgi tuleneb ingliskeelne nimetus 1940ndate aastate keskpaigast, kui tollaste suurte arvutite tööd mõni putukas (*bug*) võis tõsiselt häirida.

Tänapäevaste programmeerimisvahendite koosseisus on sageli ka *silur* (*debugger*). Silur võimaldab näiteks seisatada programmi selle töötamise käigus, et uurida, millised muutujad on väärtustatud ja mis väärtus neil parajasti on. Siluriga on programmikoodi võimalik ridahaaval läbida. Ka Thonnys on siluri kasutamise võimalus olemas ("putukaga" nupp või *Run --> Debug current script*).

Lisaks on tänapäevastes programmeerimisvahendites ka mõeldud sellele, et vigu üldse vähem teha. Näiteks püütakse aidata nimede kirjutamisel, sulgude tasakaalustamisel, treppimisel vms.

Lõpetuseks

Üsna populaarseks on näiteks muutunud *testipõhine arendus* (*test driven development*), mille korral esmajärjekorras töötatakse välja testid. Täpsemalt saab lugeda näiteks siit (inglise keeles).

Eesti keeles on tarkvara protsessidest, kvaliteedist ja standarditest põhjalikult kirjutanud Jaak Tepandi Tallinna Tehnikaülikooli Informaatikainstituudist. Seal on juttu ka testimisest. Inglise keeles on testimisest palju kirjutatud, näiteks siin.

5.2 REKURSIOON: SISSEJUHATUS

Funktsioonist kordavalt

Oleme kasutanud erinevaid funktsioone - osa on olnud Pythonis juba valmis, aga oleme neid ka ise funktsioone defineerinud. Kursusel *Programmeerimise alused* oli 6. nädal funktsioonidele pühendatud.

Meenutame, et funktsioon tagastab väärtuse, mis määratakse võtmesõna `return` abil. Kui seda ei tehta, siis tagastatakse spetsiaalne väärtus `None` (eesti keeles "mitte miski"). "Väärtusetud" funktsioonid ei pruugi muidugi väärtusetud olla - nende roll on lihtsalt midagi ära teha, nt `print` väljastab oma argumendi(d) ekraanile, kuid ei tagasta midagi.

Olgu meil järgmine programm

```
def summa(a, b):
    return a + b

def topelt_summa(a, b):
    return 2 * summa(a, b)

print(summa(3, 4))
```

Näeme, et funktsioon `summa` on kutsutud välja funktsiooni `topelt_summa` definitsioonis ja ka põhiprogrammis (funktsiooni `print` argumendina). Veel saab funktsiooni välja kutsuda Thonny käsureaaknas (*Shell*):

```
>>> summa(5, 7)
12
```

Rekursiivne väljakutse

Eespool kutsuti üks funktsioon välja teise funktsiooni definitsioonis. Tegelikult saab funktsiooni välja kutsuda tema enda definitsioonis. Sellisel juhul on tegemist *rekursiivse väljakutsega*. Rekursioon on funktsioonide defineerimise viis, kus defineeritav funktsioon kutsub välja iseennast (kuid mitte sama argumendi väärtusega).

Rekursioon sobib hästi just selliste ülesannete lahendamiseks, kus tervikülesannet saab jaotada mingis mõttes väiksemateks samasugusteks ülesanneteks. Oluline on, et lõpuks oleks need "väiksemad" ülesanded nii väikesed, et nende lahendamine oleks väga lihtne.

Üks tuntumaid rekursiooni näiteid on faktoriaali arvutamine. Positiivse täisarvu *n* faktoriaal (tähistus *n!*) on *n* esimese positiivse täisarvu korrutis. Näiteks $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Eraldi on

kokku lepitud, et $0! = 1$, samuti $1! = 1$. Rekursiivsena saab faktoriaali leidmist kirjeldada nii, et iga järgmise arvu faktoriaali saame esitada eelmise arvu faktoriaali abil. Näiteks $4! = 4 \cdot 3!$ ja omakorda $3! = 3 \cdot 2!$ ning $2! = 2 \cdot 1!$. Ja $1! = 1$ või kui tahame 0 ka mängu võtta, siis võime öelda ka, et $1! = 1 \cdot 0!$ ja $0! = 1$. Üldistatult saame kaks haru:

- $n! = 1$, kui $n = 0$
- $n! = n \cdot (n-1)!$, kui $n > 0$

Programselt saaks me seda definitsiooni kirjeldada niimoodi:

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

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

Korrektsetes rekursiivsetes funktsioonides on alati mitu haru. Protsessi lõppemiseks peab vähemalt üks haru olema ilma rekursiivse väljakutseta. Seda haru nimetatakse *rekursiooni baasiks*. Rekursiivse väljakutsega haru nimetatakse *rekursiooni sammuks*. Rekursiivsete väljakutsetega võib olla ka mitu haru. Samuti võib harva olla kasulik määrata mitu erinevat rekursiooni baasi.

Mõned näited

Sarnaselt tsüklile võimaldab rekursioon kirjeldada korduvtäidetavaid protsesse.

Mänguliselt saab rekursiooni ja ka teisi programmeerimise kontruktsioone harjutada mängus Lightbot. Tasemetel 3.1 ja 3.2 saabki hakkama ainult nii, et protseduur iseennast välja kutsub.

Proovige, mida teeb järgmine programm.

```
def rek_fun(n):
    if n > 0:
        print("Põhi!")
    else:
        print(n)
        rek_fun(n + 2)

rek_fun(-7)
```

Proovige programmi muuta ja käivitage uuesti. Näiteks võib `n > 0` asendada mõne muu tingimusega või muuta ridade `print(n)` ja `rek_fun(n + 2)` järjekorda.

Rekursiivne funktsioon on tegelikult tavaline Pythoni funktsioon. Seega võib tal olla ka mitu argumenti.

Enesetest:

Mida väljastab järgnev koodilõik?

```
def süt(a, b):
  if b == 0:
    return a
  else:
    return süt(b, a % b)
print(süt(20, 12))
```

Vali 4

Vali 8

Vali 20

Vali Muu arv.

Vali Veateade, rekursiooni baasini ei jõuta kunagi.

Rekursiooni kasutatakse laialdaselt programmeerimises, arvutiteaduses ja matemaatikas, samuti keeleteaduses, muusikas, kunstis jne.

Kunstis võib välja tuua näiteks Maurits Cornelis Escheri (1898-1972) tööd.

5.3 REKURSIOONIST DETAILSEMALT

Thonny harrastab rekursiooni

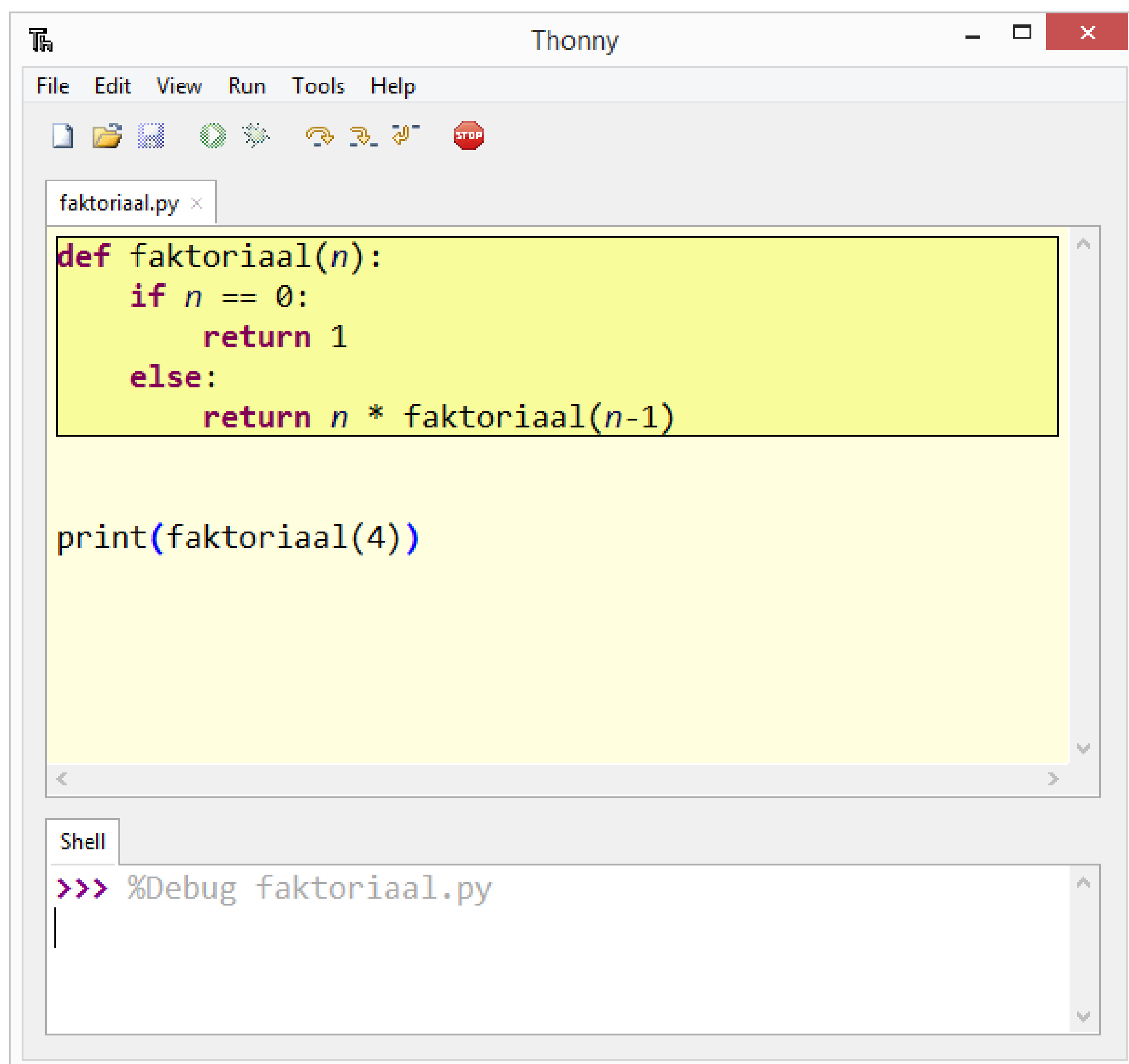
Rekursiivse funktsiooni toimimisest ei pruugi olla lihtne aru saada. Võtame näitlikustamisel appi Thonny silumisvõimalused. Tavaliselt oleme programmi käivitanud ja see on oma töö praktiliselt silmapilkselt ära teinud. Nüüd aga kasutame võimalust `Run` --> `Debug current script`, mille abil saame programmitööd sammukaupa jälgida. Järgmise sammu tegemiseks on mitu võimalust. Esialgu on sobiv kasutada varianti `Step into (F7)`.

Vaatame sedasama faktoriaali arvutamise programmi.

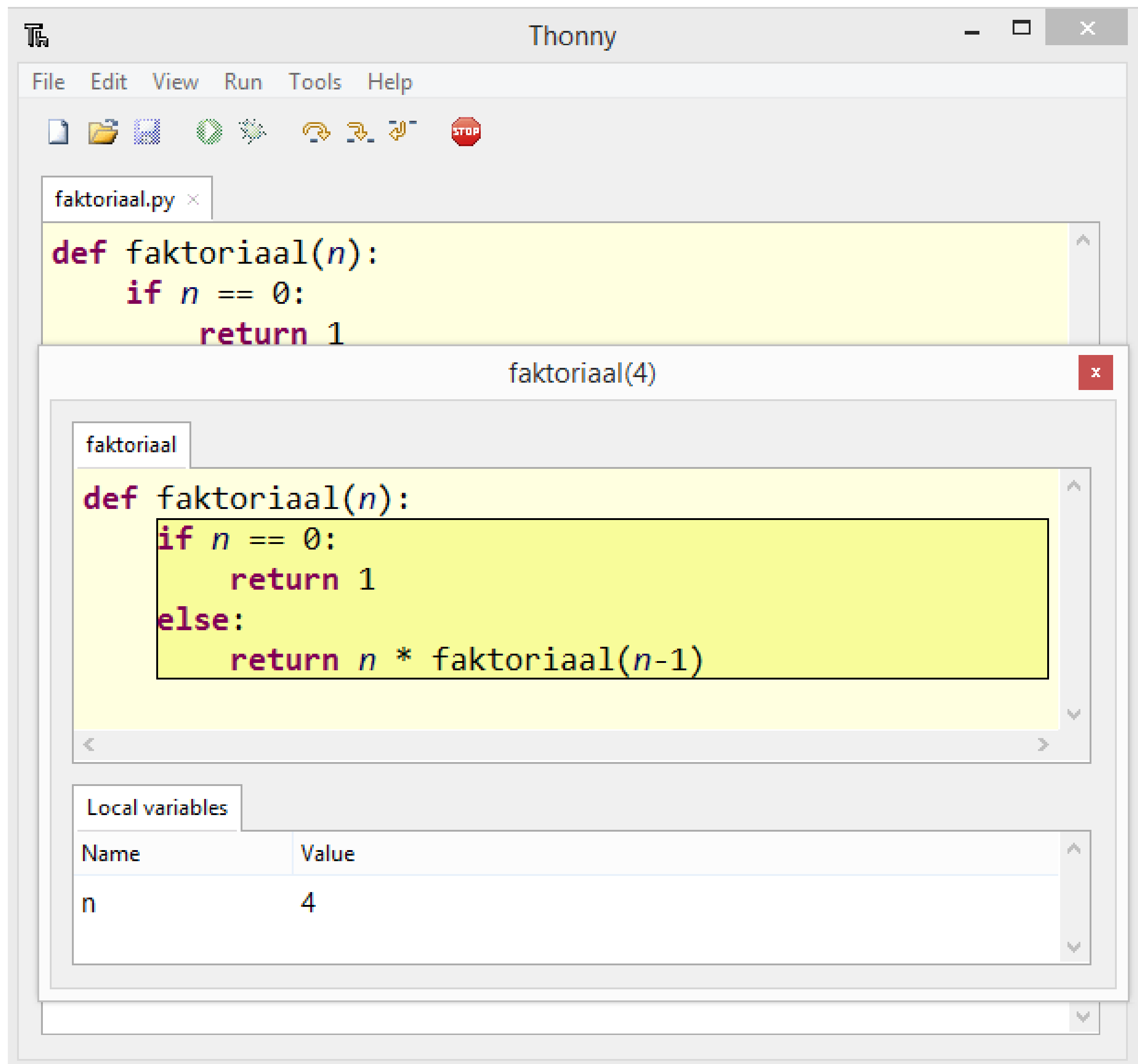
```
def faktoriaal(n):
    if n == 0:
        return 1
    else:
        return n * faktoriaal(n-1)

print(faktoriaal(4))
```

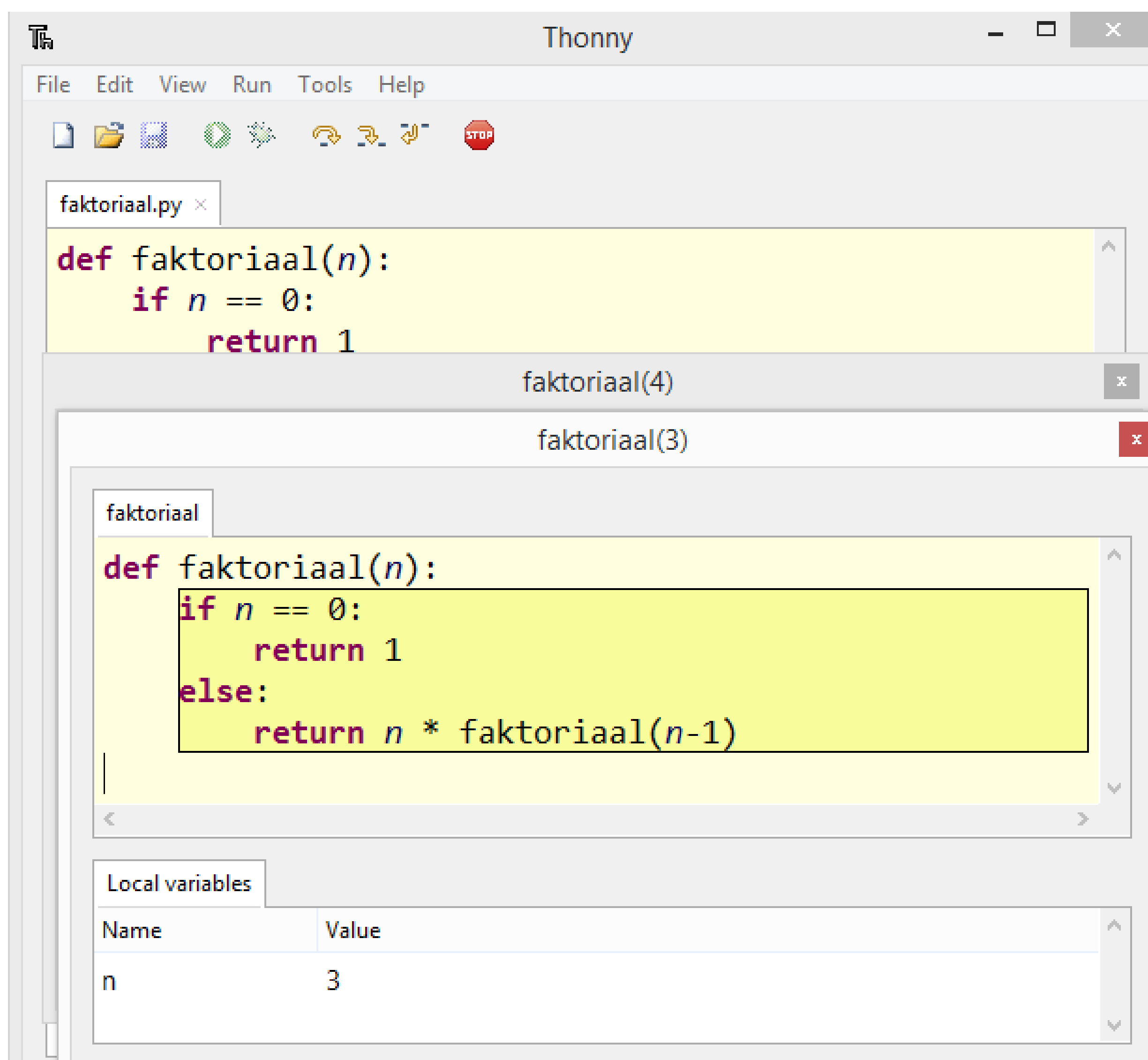
Kui paneksime kohe tööle `Run` --> `Run current script (F5)` saaksime vastuse 24 ja vahepealse töö kohta infot mitte. `Run` --> `Debug current script` abil aga värvub osa koodist kollaseks, mis näitab, kuhu programmi täitmine on jõudnud. Näiteks esimese sammuna värvub funktsiooni kirjeldus - see funktsioon "õpitakse" selgeks.



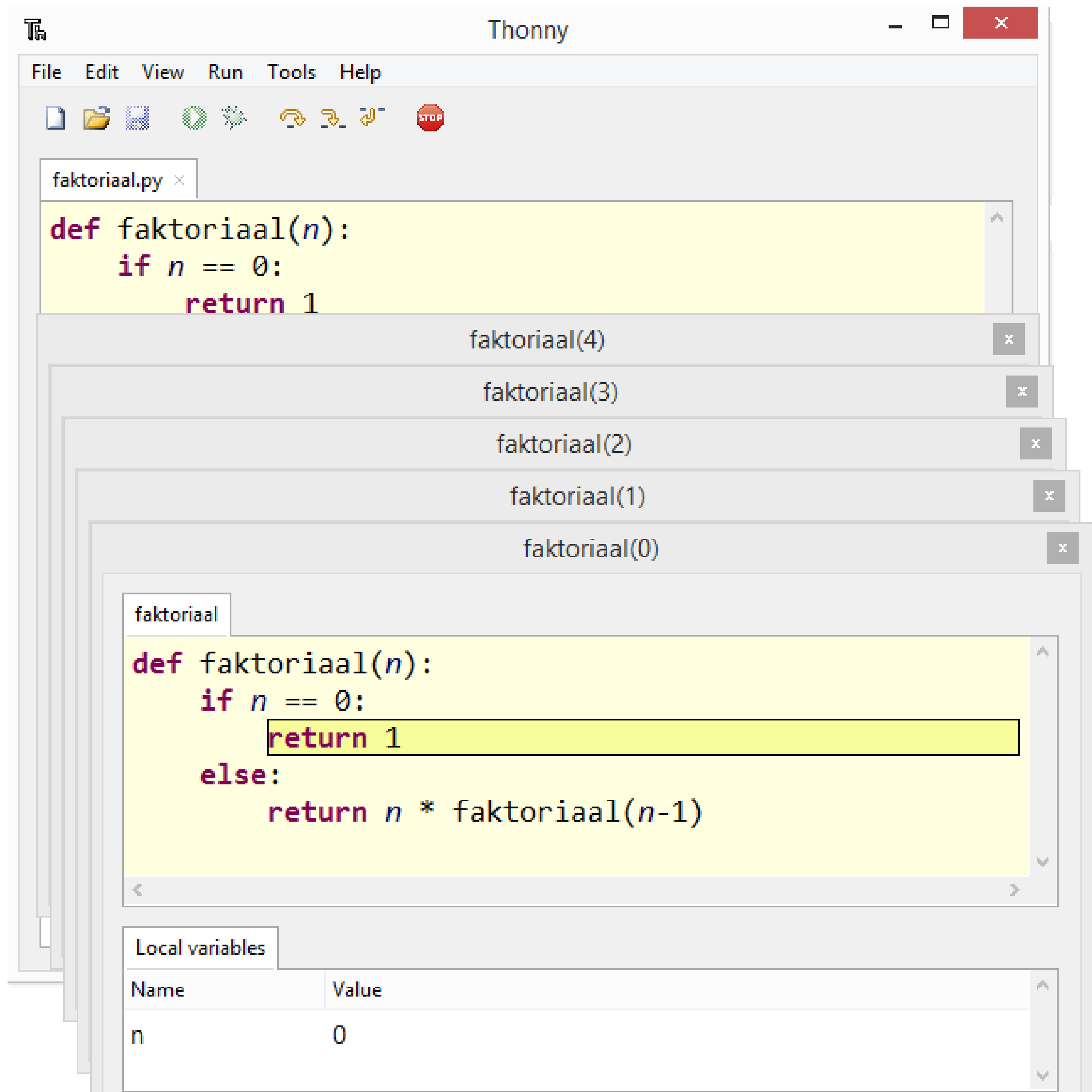
Kui nüüd sammhaaval edasi minna (Step into (F7)), siis jõuame varsti seisuni, kus faktoriaal(4) arvutamiseks ilmub uus aken.



See demonstreeribki, et funktsiooni rakendamine on küllaltki iseseisev. Kui nüüd edasi "sammuda", siis varsti tuleb arvutada `faktoriaal(3)`. Seda veel enne, kui `faktoriaal(4)` lõplikult leitud saab. Ja nii ilmub uus aken.



Edasi sammudes ilmub järjest uusi aknaid, kuni lõpuks jõuame `faktoriaal(0)`, mis ometigi konkreetse tulemuse (1) tagastab.



Nüüd hakkavad aknad järjest sulguma, sest vajalikud andmed arvutamiseks on järjest olemas. Lõpuks tuleb ka oodatud 24 ekraanile.

Soovitav on mängida selliselt läbi ka teised programmid, millest võib-olla muidu hästi aru ei saa.

Rekursiivne väljakutse

Rekursiivsetes funktsioonides võib rekursiivne väljakutse olla erinevates kohtades. Näiteks järgmises programmis on see viimase tegevusena vastavas harus.

```
def print_alla(n):
    if n <= 0:
        print("Stop!")
    else:
        print(n)
        print_alla(n - 1)
```

Enne programmi käivitamist püüdke ennustada, mis ilmub ekraanile järgmistel juhtudel.

```
print_alla(4)
print_alla(0)
print_alla(-4)
```

Näeme, et lause `print(n)` on enne rekursiivset väljakutset ja ekraanile väljastatakse `n` väärtus, mis on just selles konkreetses `print_alla` väljakutses. Kuna iga järgmine väljakutse on ühe võrra väiksema argumendiga (`n - 1`), siis ilmuvad ka arvud ekraanile kahanevas järjekorras.

Muudame nüüd `print(n)` ja `print_alla(n - 1)` järjekorda.

```
def print_kuhu(n):
    if n <= 0:
        print("Stop!")
    else:
        print_kuhu(n - 1)
        print(n)
```

Püüdke enne käivitamist ennustada, mis ekraanile ilmub.

```
print_kuhu(4)
print_kuhu(0)
print_kuhu(-4)
```

Paneme tähele, et nüüd on `print(n)` pärast rekursiivset väljakutset. Seega enne, kui midagi ekraanile väljastatakse, "avanevad" kõik `print_kuhu` väljakutsed. Lõpuks siis `print_kuhu(0)` toimel ilmub ekraanile `Stop!`. Nüüd hakkavad väljakutsed "sulguma", aga vahetult oma töö lõpus väljastatakse ekraanile `n` väärtus, mis selles väljakutses hetkel on. Oluline on, et igas väljakutses on `n` väärtus sõltumatu.

Niisiis ekraanile tekivad arvud kasvavas järjekorras. Võib öelda, et enne rekursiivset väljakutset tehtavad tegevused toimuvad "kahanevalt". Pärast rekursiivset väljakutset tehtavad tegevused toimuvad "kasvavalt".

```
def print_ules(n):
    if n <= 0:
        print("Stop!")
    else:
        print_ules(n-1)
        print(n)
```

Mis aga juhtub siis, kui väljastamise käsk on nii enne kui pärast rekursiivset väljakutset?

```
def print_alla_ules(n):
    if n <= 0:
        print("Stop!")
    else:
        print(n)
        print_alla_ules(n - 1)
        print(n)
```

Rekursioon sõnede ja järjenditega

Senised rekursiooninäited on põhiliselt olnud arvudega seotud. Nüüd vaatleme ka teisi andmetüüpe. Näiteks saab rekursiivselt kontrollida, kas sõne on palindroom - algusest või lõpust loetult sama. Kontrollimisel kasutatakse asjaolu, et sõne on palindroom, kui tema esimene ja viimane sümbol on sama ning nende vahelejääv alamsõne on palindroom. Nii saamegi rekursiivse funktsiooni, kus baasiks on ühesümboliline või tühi sõne, mida saame lugeda palindroomiks. Alamsõne leidmiseks on kasutatud viilutamist: `s[1:-1]` puhul on alamsõne alguse indeks (kaasaarvatud) 1 ja lõpu indeks (väljaarvatud) -1.

```
def on_palindroom(s):
    if len(s) <= 1:
        return True
    else:
        return s[0] == s[-1] and on_palindroom(s[1:-1])
```

Järjendi pikkuse leidmiseks on olemas spetsiaalne funktsioon `len`. Siin harjutame ise sarnaste funktsioonide defineerimist, et teiste samalaadsete ülesannetega ka paremini toime tulla. Vaatleme kahte varianti - üks tsükliga ja teine rekursiooniga.

Tsüklis liidetakse loendurile igat elemendi vaadeldes 1.

```
def pikkus(loend):
    i = 0
    for c in loend:
        i += 1
    return i
```

Rekursiooni korral kasutatame baasina asjaolu, et tühja listi pikkus on 0. Muidu aga on listi pikkus 1, millele on liidetud sellise alamlisti pikkus, kust on välja jäetud esimene element.

```
def rpikkus(loend):
    if loend == []:
        return 0
    else:
        return 1 + rpikkus(loend[1:])
```

Lõpuks tuleme ikkagi arvude juurde tagasi ja vaatleme astendamise funktsiooni. Baasi saame teadmisest, et iga arv astmes 0 on 1. Igal rekursiooni sammul arvutatakse arvu ühe võrra väiksem aste.

```
def aste(n, m):
    if m == 0:
        return 1
    else:
        return n * aste(n, m-1)
```

5.4 KEELETÖÖTLUS

Kuigi tundub, et arvutid tegelevad peamiselt arvude ja matemaatiliste tehetega, pole see kaugeltki nii. Kui mõtleme enda tegevuse peale, siis suure osa ajast otsime me veebist enda jaoks vajalikku materjali, kirjutame tekstitöötlusprogrammi abil mitmesuguseid tekste või suhtleme tuttavatega. Ka arvuti ei võta teksti kui lihtsalt sümbolite jada, nii oskab ta sageli aru saada meie otsingumustrist ja pakkuda vastuseks ka sünonüüme või samu sõnu teises käändes sisaldavaid tekste, parandada õigekirjavigasid jne. Leidub hulgaliselt tõlkimisprogramme (näiteks kõigile kasutatavad Google Translate, TÜ masintõlge). Arvutit saame juhtida hääle abil ja talle teksti dikteerida (kõnetuvastus, näiteks http://bark.phon.ioc.ee/webtrans) ning tekste lugemise asemel ka kuulata (kõnesüntees, näiteks https://www.eki.ee/heli).

Tekstiga tegelemiseks sobib Python väga hästi. Teksti tõsisemaks töötlemiseks vajame küll spetsiaalseid lisamooduleid, mis oskaksid morfoloogilist analüüsi (sõnavormide põhjal sõna algvormi leidmist) ja sünteesi (sõnade algvormide põhjal sõnavormide koostamist), tunneksid keele süntaksit (kuidas võivad sõnad lauses paikneda), semantikat (ehk seda, mida tekst tegelikult tähendab) jms. Sellised vahendeid on palju, levinumateks näideteks on loomuliku keele töötluseks moodul NLTK või eesti keele jaoks mõeldud EstNLTK. Samas saame tekstist rohkem infot saada ka ilma erivahenditeta.

Sõnestamine ja sagedusloend

Tekst koosneb sõnadest ning teksti uurimist võimegi alustada selle sõnestamisest ehk sõnadeks jagamisest. Tegelikult kasutatakse mõistet “sõne” (i. k. string) sageli laiemas tähenduses ning sõltuvalt kontekstist loetakse sõnedeks lisaks sõnadele ka arvud, kirjavahemärgid jne, kuid momendil pole see väga tähtis. Püüame tekstist paremat pilti saada, leides selles olevad sagedasemad sõnavormid. Võtame aluseks ühe teksti (meie näites Arvo Valtoni “Kanaromaani”) ja püüame seda sõnestada. Selleks loeme failist ridade kaupa teksti, tükeldame iga rea tühikute järgi sõnadeks ning lisame sõnavormide järjendisse.

```
# Avame faili
fail = open("valton_kanaromaan.txt", "r", encoding="utf-8")
# Järjend sõnavormide jaoks
sonavormid = []
for rida in fail:
    #print(rida)
    # Eemaldame reavahetused jms rea algusest ja lõpust
    rida = rida.strip()
    # Töötleme rida sel juhul, kui see pole tühi
    if rida != "":
        # Tükeldame rea tühikute kohalt sõnedeks
        sonad = rida.split()
        # Iga tüki lisame sõnavormide järjendisse, kui seda seal
veel pole
        for sona in sonad:
            if sona not in sonavormid:
                sonavormid.append(sona)
# Sulgeme faili
fail.close()
print(sonavormid)
```

Saadud järjendil on mitu puudust:

- järjendis eristatakse suuri ja väikseid tähti (kuigi vahel on see vajalik, näiteks nimede puhul);
- järjendis esinevad sõnad koos kirjavahemärkidega (nii on seal nii sõna “maailma” kui ka “maailma,”);
- me ei tea midagi sõnavormide esinemissageduse kohta.

Seega parem lahenduskäik oleks järgmine: väiketähestame teksti, kustutame ära kõik kirjavahemärgid (teeme selleks praegu suhteliselt lihtsakoelise funktsiooni) ning asendame järjendi sõnastikuga, kus võtmeks on sõnavorm, väärtuseks aga tema esinemissagedus. Nii saame sagedusloendi, millest trükime välja suurima esinemissagedusega sõnavormid.

```
def eemaldaPunktuatsioon(tekst):
    tekst = tekst.replace(".", "")
    tekst = tekst.replace(",", "")
    tekst = tekst.replace("!", "")
    tekst = tekst.replace("?", "")
    return tekst

# Avame faili
fail = open("valton_kanaromaan.txt", "r", encoding="utf-8")

# Sõnastik sõnavormide jaoks
sonavormid = {}

for rida in fail:
    #print(rida)
    # Eemaldame reavahetused jms rea algusest ja lõpust
    rida = rida.strip()
    # Väiketähestame teksti
    rida = rida.lower()
    # Eemaldame punktuatsiooni
    rida = eemaldaPunktuatsioon(rida)
    # Töötleme rida sel juhul, kui see pole tühi
    if rida != "":
        # Tükeldame rea tühikute kohalt sõnedeks
        sonad = rida.split()
        # Kui sellise võtmega elementi sõnastikus veel pole,
        # lisame selle koos väärtusega 1
        # (selleks momendiks on seda vormi esinenud tekstis 1 kord),
        # kui aga on, suurendame selle võtmega elemendi väärtust ühe
võrra
        for sona in sonad:
            if sona not in sonavormid:
                sonavormid[sona] = 1
            else:
                sonavormid[sona] += 1

# Sulgeme faili
fail.close()

print("Sõnavormide arv:", len(sonavormid))

loendur = 1
for vorm in sorted(sonavormid, key=sonavormid.get, reverse=True):
    print(loendur, vorm, sonavormid[vorm])
    loendur += 1
    if loendur > 30:
        break
```

Tähtsamad sõnavormid

Sagedusloendit vaadates näeme, et kuigi seal esinevad sõnad “tsee”, “kana” ja “kukk” (mis on just sellele tekstile iseloomulikud), ei ütle enamik seal olevatest sõnadest (nt “oli”, “ja”, “aga”) meile teksti kohta otseselt midagi. Kuni sagedusloend sisaldab sõnu, mis on keeles üldiselt kõrge sagedusega, ei saa me head pilti sellest, millised sõnad on käesolevale tekstile iseloomulikud. Seega tuleks meil jätta alles vaid need sõnavormid, mis pole kogu eesti keeles sagedased. Kuidas seda teha? Üheks võimaluseks on kasutada sagedaste sõnavormide loendit (eesti_keele_sonavormid.txt, kopeeritud https://keeleressursid.ee/et/83-article/clutee-lehed/256-sagedusloendid): võtta sealt mingi arv sagedasemaid eestikeelseid sõnavorme ning jätta oma teksti sagedusloendisse alles vaid sõnad, mis sagedaste sõnavormide loendis ei esine. Teeme seda järgnevas näites.

```
def eemaldaPunktuatsioon(tekst):
    tekst = tekst.replace(".", "")
    tekst = tekst.replace(",", "")
    tekst = tekst.replace("!", "")
    tekst = tekst.replace("?", "")
    return tekst

eesti_keele_sagedasemad = []
# Avame eesti keele sagedasemate sõnavormide faili
fail = open("eesti_keele_sonavormid.txt", "r", encoding="utf-8")
for rida in fail:
    # Tükeldame rea tühiku järgi, failis on esimesel kohal sõnavormi
sagedus,
    # teisel kohal sõnavorm ise
    rida = rida.split()
    eesti_keele_sagedasemad.append(rida[1])
    # Võtame arvesse vaid sagedasemaid eesti keele sõnavorme
    if len(eesti_keele_sagedasemad) > 200:
        break
fail.close()
#print(eesti_keele_sagedasemad)


# Avame faili
fail = open("valton_kanaromaan.txt", "r", encoding="utf-8")

# Sõnastik sõnavormide jaoks
sonavormid = {}
```

```
for rida in fail:
    # Eemaldame reavahetused jms rea algusest ja lõpust
    rida = rida.strip()
    # Väiketähestame teksti
    rida = rida.lower()
    # Eemaldame punktuatsiooni
    rida = eemaldaPunktuatsioon(rida)
    # Töötleme rida sel juhul, kui see pole tühi
    if rida != "":
        # Tükeldame rea tühikute kohalt sõnedeks
        sonad = rida.split()
        # Kui sellise võtmega elementi sõnastikus veel pole,
        # lisame selle koos väärtusega 1
        # (selleks momendiks on seda vormi esinenud tekstis 1 kord),
        # kui aga on, suurendame selle võtmega elemendi väärtust ühe
võrra
        for sona in sonad:
            # Lisame sagedusloendisse vaid sõnad, mis ei esine
            if sona not in eesti_keele_sagedasemad:
                if sona not in sonavormid:
                    sonavormid[sona] = 1
                else:
                    sonavormid[sona] += 1

# Sulgeme faili
fail.close()

print("Sõnavormide arv:", len(sonavormid))

loendur = 1
for vorm in sorted(sonavormid, key=sonavormid.get, reverse=True):
    print(loendur, vorm, sonavormid[vorm])
    loendur += 1
    if loendur > 30:
        break
```

Kirjavahemärkide kustutamine oli meie eelmises programmis pisut kohmakas: asenduskäsuga pidime iga märgi eraldi eemaldama. Parem lahendus oleks kasutada regulaaravaldist -- sõnesid kirjeldavat üldistavat mustrit --, mis võimaldaks ühe käsuga leida üles kõik kirjavahemärgid ja eemaldada need. Samuti võimaldavad regulaaravaldised mugavalt otsida mingile mallile vastavaid sümbolijärjendeid, näiteks neid, mille teine täht on vokaal ning milles on neli tähte. Regulaaravaldiste kasutamist Pythonis tutvustatakse lähemalt siin.

Edasiarenduseks

Kui koguksime andmeid paljude erinevast žanrist või erinevatel teemadel tekstide kohta, tekiksid meil sagedasemate sõnavormidega sõnastikud iga tekstiklassi kohta (kultuuriartiklid, majandusartiklid ja spordiartiklid või ilukirjandus, ajakirjandus ja seadused). Seejärel oleks

võimalik uusi tekste sõnavara kattuvuse alusel klassifitseerida ühte või teise rubriiki kuuluvateks.

Tegelesime praegu sõnavormide, mitte algvormide ehk lemmadega. Nii saime sagedusloendisse eraldi nii sõnad “kana” ja “kanad” kui sõnad “muna” ja “mune”, kuigi sisulises mõttes on tegemist sama mõistega. Reeglina kasutatakse siiski algvorme ning nende saamiseks vajaksime rohkem keeletöötlusvahendeid, näiteks morfoloogilist analüsaatorit.

Kui sooviksime tuvastada teksti keelt, võiksime toimida samamoodi, andes ette erinevate keelte sõnastikke. Keeletuvastuseks on olemas aga ka lihtsam moodus, mis töötab päris hästi, kui võimalikke keeli pole väga palju. Nimelt on igas keeles tähtede sagedus veidi erinev, näiteks eesti keeles esineb kõige rohkem tähte “a”, inglise keeles aga on kõige sagedasem täht “e” ning “a” on alles kolmandal kohal peale tähte “t” (ingl https://en.wikipedia.org/wiki/Letter_frequency). Nii võiksime ainuüksi mõne tekstirea analüüsi järel öelda, millise keele tekstiga on tõenäoliselt tegemist, ning aluseks poleks vaja sõnade sagedusloendeid, vaid erinevate keelte tähtede sagedusloendeid.