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
```

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 · 3!* ja omakorda *3! = 3 · 2!* ning *2! = 2 · 1!*. Ja *1! = 1* või kui tahame *0* ka mängu võtta, siis võime öelda ka, et *1! = 1 · 0!* ja *0! = 1*. Üldistatult saame kaks haru:

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

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

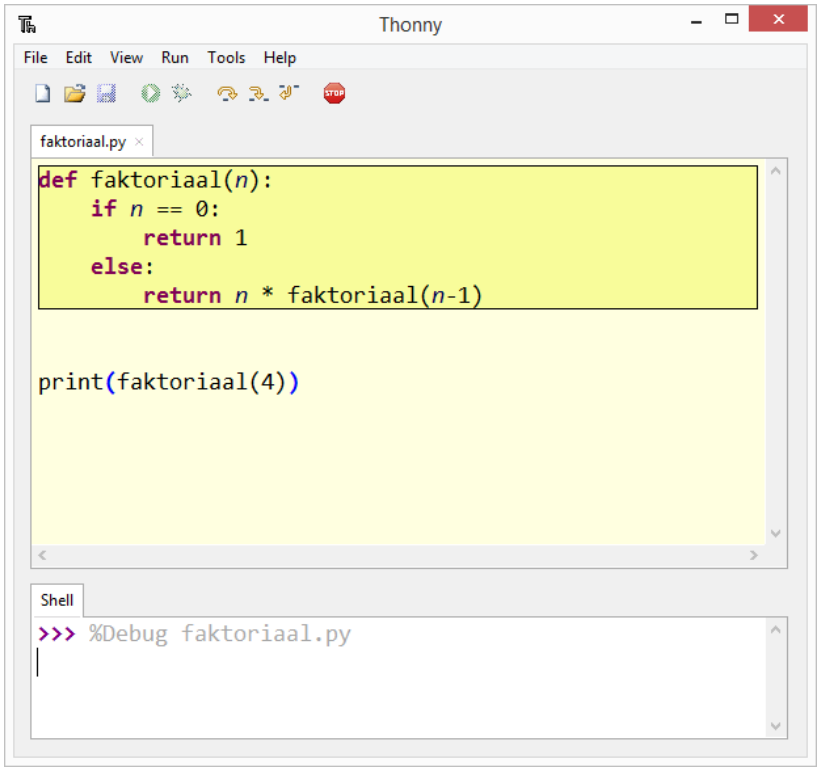
Thonny harrastab rekursiooni

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).

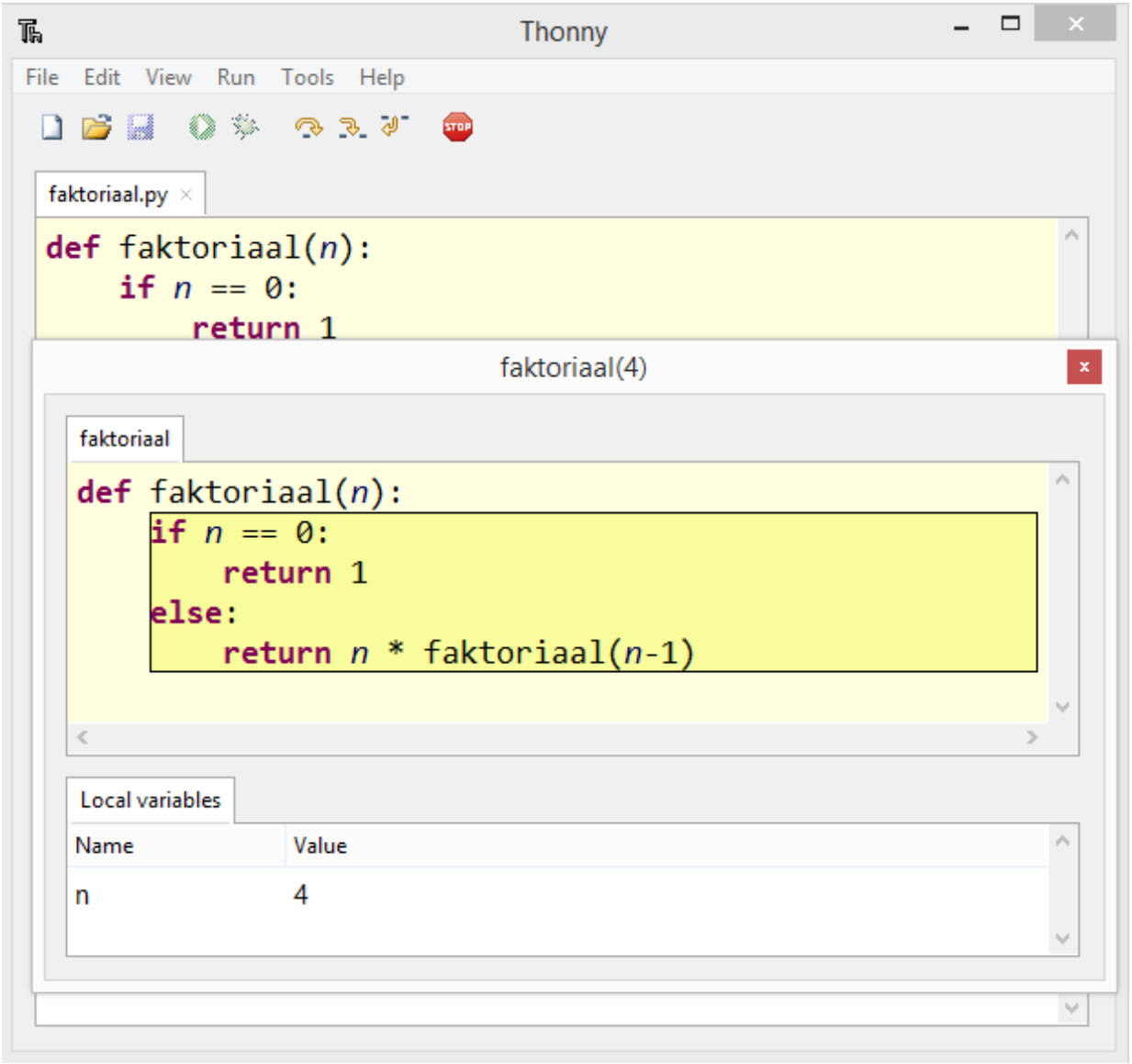
Vaatame sedasama faktoriaali arvutamise programmi.

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

print(faktoriaal(4))
```

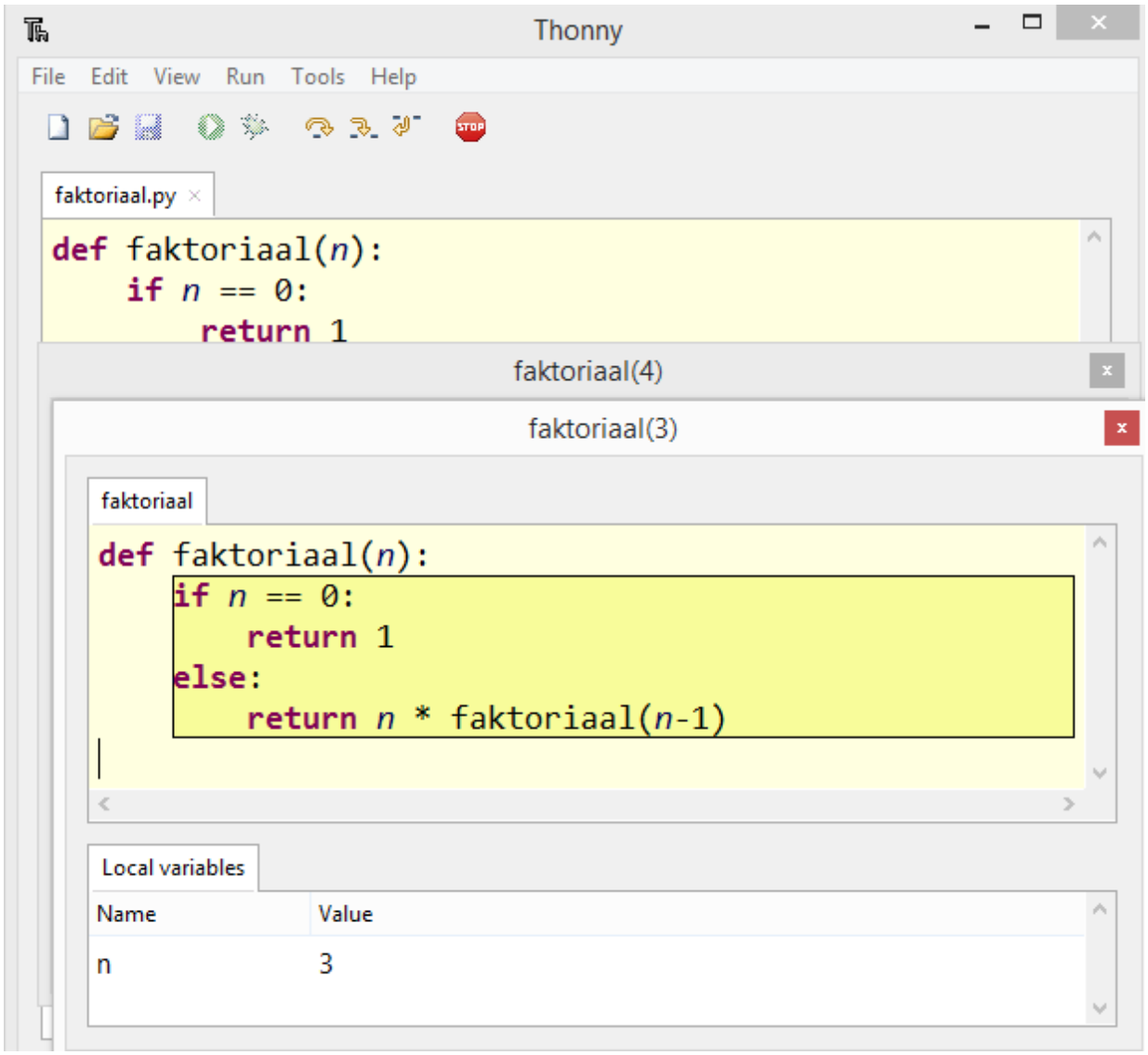
Kui paneksime kohe tööle Run --> Run current script (F5) saaksime vastuse 24 ja vahepealse töö kohta infot mitte. Run --> Debug current script abil aga värvub osa koodist kollaseks, mis näitab, kuhu programmi täitmine on jõudnud. Näiteks esimese sammuna värvub funktsiooni kirjeldus - see funktsioon "õpitakse"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



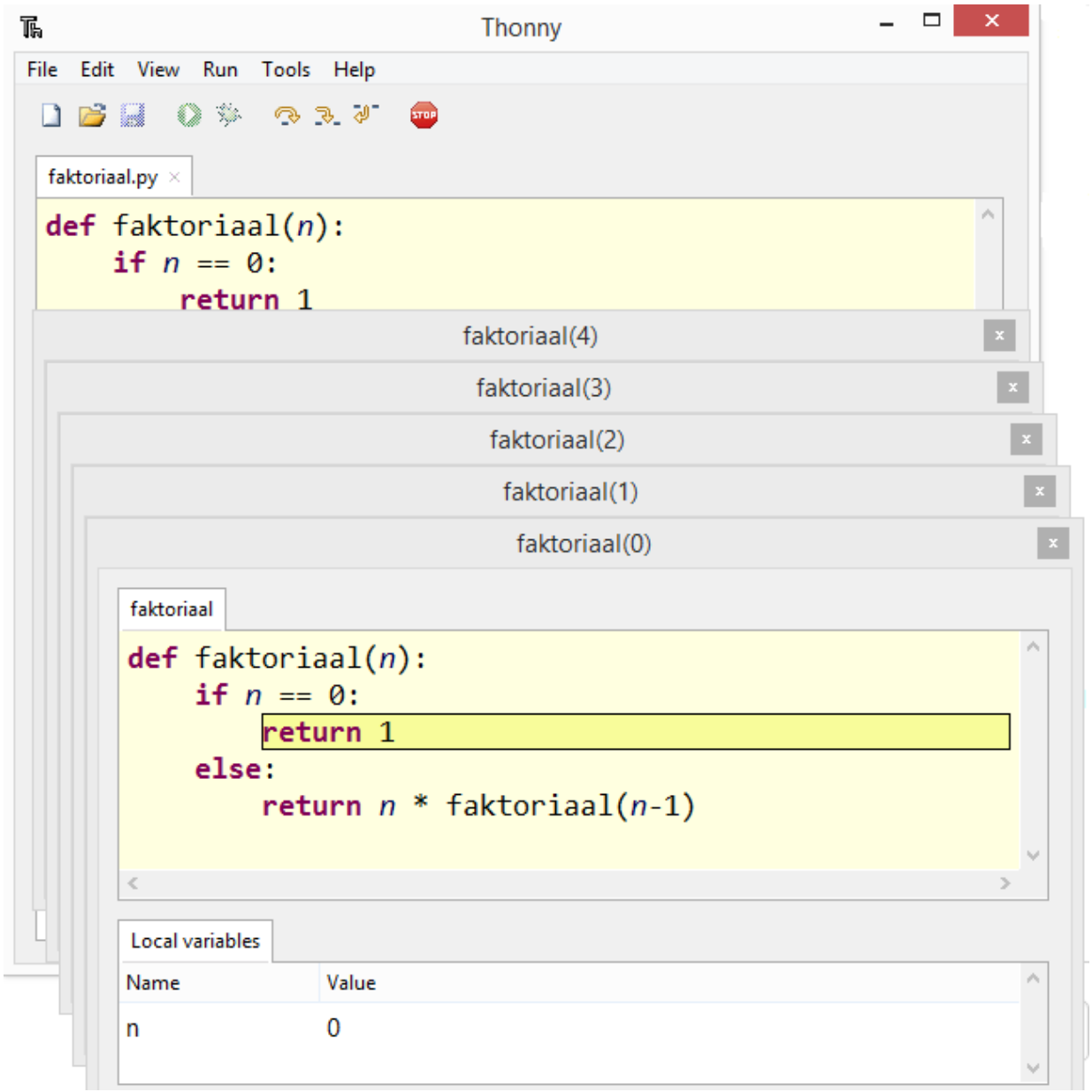
Kui nüüd sammhaaval edasi minna (`Step into (F7)`), siis jõuame varsti seisuni, kus `faktoriaal(4)` arvutamiseks ilmub uus aken.



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 ANDMETÖÖTLUS MOODULIGA PANDAS

Materjal on veel toorevõitu. Palun teatage parandusettepanekutest vastavas foorumis!!!

Sissejuhatus

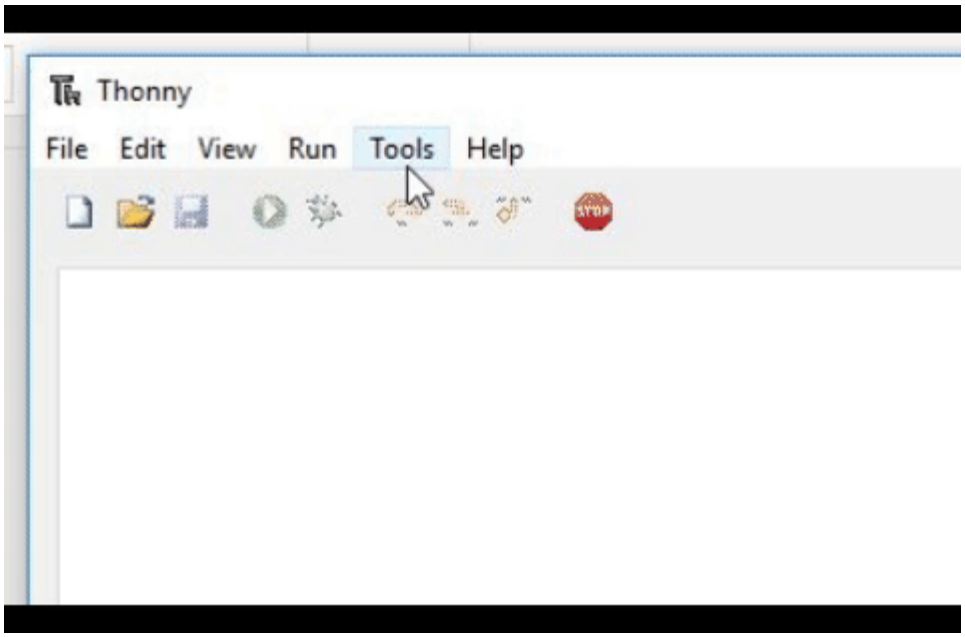
Elame ajastul, kus kogutakse hulgaliselt erinevaid andmeid. Ühelt poolt sisaldab see näiteks ohte privaatsusele, teiselt poolt aga annab võimalusi õigemaid otsuseid teha. Otsustamiseks või ka lihtsalt millestki parema ülevaate saamiseks tuleb andmeid mõistlikult töödelda. Andmetöötluseks on väga erinevaid vahendeid, millest käesoleval kursusel kasutame Pythoni moodulit Pandas. Selle nädala materjalides tutvustatakse Pandast ja vaadeldakse andmeid teatrikülastuste kohta. Ülesanne on aga raamatukogude kohta. Andmed on saadud statistikaameti veebilehelt, kus on tohutult palju erinevaid andmeid. Valmis andmeid on võimalik saada ka mitmetest muudest kohtadest. Samuti on võimalik ise andmeid koguda, näiteks pulsikella abil. Natuke etteruttavalt võib öelda, et 7. ja 8. nädalal tehtav projekt võib põhineda andmetöötlusel. Seega on need materjalid ka projekti ettevalmistuseks.

Kuna Pandase kasutamise kohta eriti palju eestikeelseid materjale pole ja pole ka väljakujunenud eestikeelset terminoloogiat, on materjalides kasutatud mõningaid otsemugandusi, nt *seeria* (Series), *andmefreim* (DataFrame). Andmetöötluse materjalid on seotud Inga Konovalova bakalaureusetööga. Tema hoiab silma peal ka vastaval foorumil ning kohendab soovituste põhjal materjale.

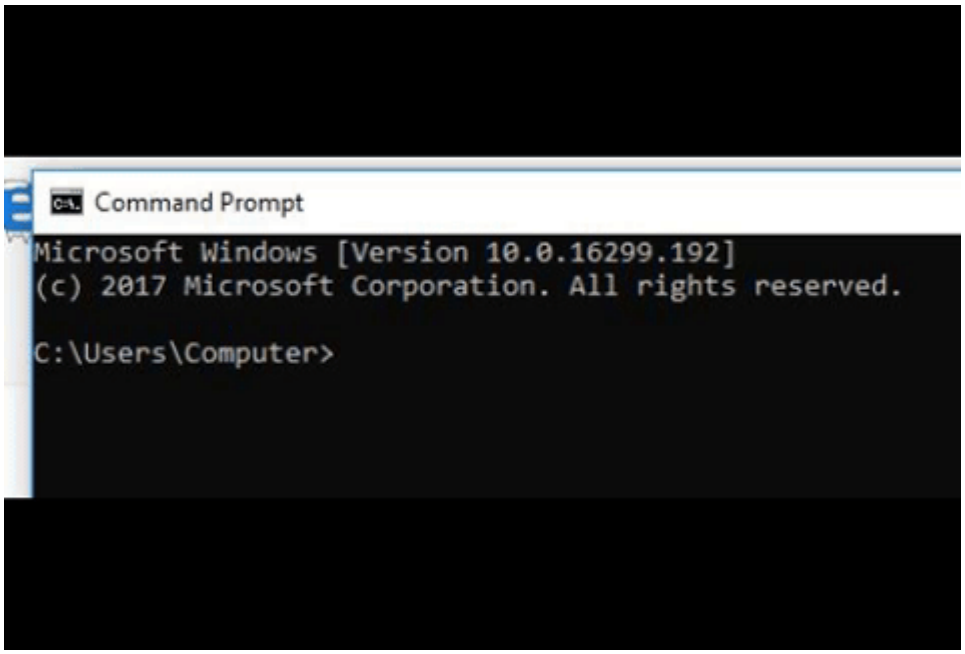
Installeerimine

Kursusel *Programmeerimise alused II* kasutavad paljud osalejad Pythonis programmide kirjutamiseks keskkonda Thonny. Vaatamegi, kuidas Thonnys installeerida Pythoni andmetöötluseks mõeldud moodul Pandas.

Pandase paigaldamiseks tuleb Thonny ülamenüüst valida Tools --> Manage packages ja otsingusse kirjutada *Pandas* ning vajutada nuppu Install.



Kui te ei kasuta Thonnyt, siis võib Pandase installeerimine olla mõnevõrra keerulisem. Windowsi käsurealt saab Pandase installeerida käsuga `py -m pip install pandas`

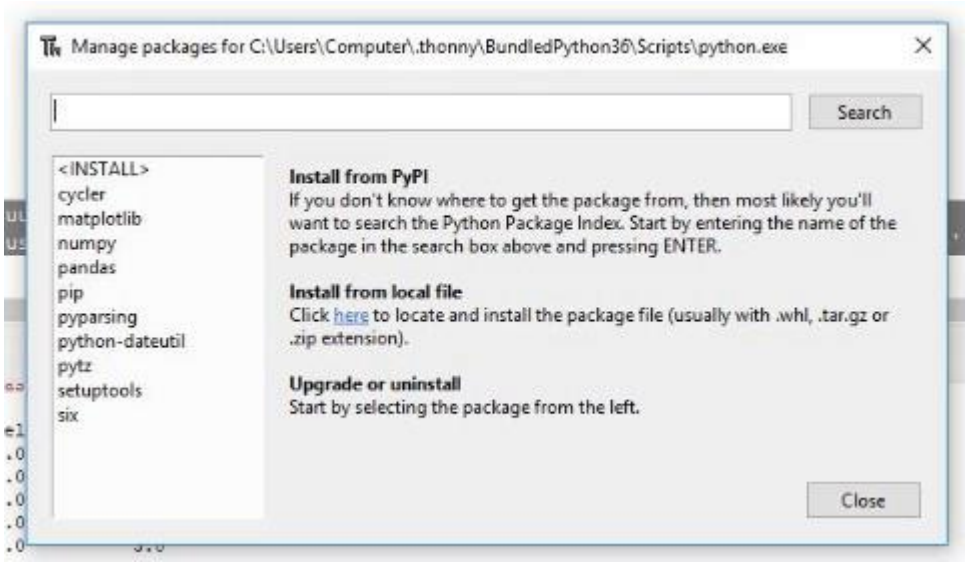


Jupyter notebookis saab Pandase installeerida reaga: `!pip install pandas`. Lisainfot installeerimise kohta saab pandase veebilehelt.

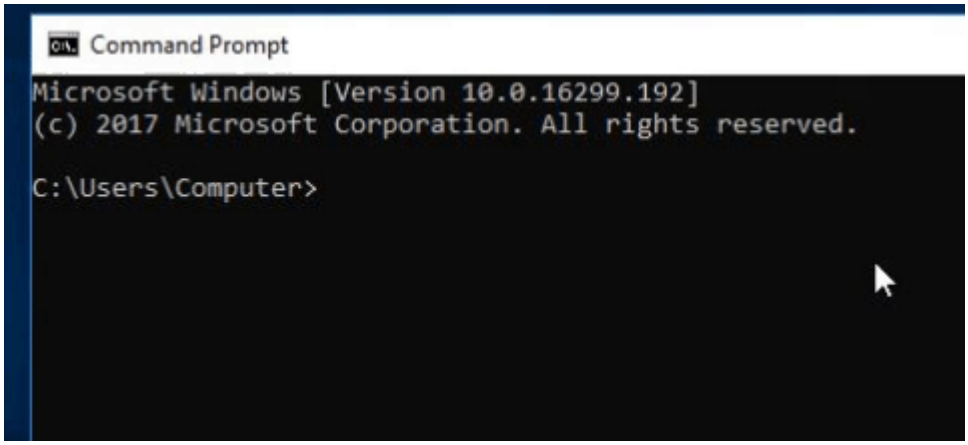
Koos Pandasega installitakse ka moodul NumPy. Tegelikult saaks pelgalt NumPy abiga andmeid töödelda, kuid Pandases on rohkem funktsioone ning andmete töötlemine on mugavam. Mooduliga NumPy saab soovi korral tutvuda siin.

Graafikute tegemiseks läheb vaja moodulit Matplotlib, mille installeerimine on sama nagu Pandase oma:

Thonnys paigaldamiseks tuleb ülamenüüst valida `Tools` --> `Manage packages` ja otsingusse kirjutada *matplotlib* ning vajutada nuppu `Install`.



Windowsi käsurealt saab Matplotlib installeerida käsuga `py -m pip install matplotlib`.



Jupyter notebookis tuleks kasutada koodirida: `!pip install matplotlib`.

Andmestruktuurid

Pandase kohta leidub väga vähe eestikeelseid materjale ja terminoloogia on lünklik. Kasutame siin materjalides ingliskeelseid termineid ja nende otsetõlkeid: *seeria*, *andmefreim* ja *paneel*.

Pandases on andmete hoidmiseks 3 põhilist andmestruktuuri:

- `Series` - seeria
- `DataFrame` - andmefreim
- `Panel` - paneel

Enimkasutatav andmestruktuur on andmefreim ning ka nendes materjalides keskendutakse sellele. Esmalt aga tutvustatakse seeriaga.

Seeria (*Series*)

Seeria (Series) on ühemõõtmeline, sarnastest andmetest koosnev, muutumatu suurusega andmestruktuur. Sellest võib mõelda ka kui kindla suurusega sõnastikust, milles igale võtmele vastab kindel väärtus.

Erinevus sõnastiku ja seeria vahel

Struktuuri poolest meenutab seeria kõige rohkem sõnastikku. Andmeid hoitakse siltidega seostatuna ja vajadusel võib kindla elemendi sildi järgi kätte saada. See meenutab sõnastiku võtme ja väärtuse süsteemi.

Sõnastiku puhul

```
isikud_sonastik = {'Võistleja nr 1': 'Mari', 'Võistleja nr
2': 'Joonas', 'Võistleja nr 3': 'Kati'}
print(isikud_sonastik)
```

saame

```
{'Võistleja nr 1': 'Mari', 'Võistleja nr 2': 'Joonas', 'Võistleja nr
3': 'Kati'}
```

Seeria puhul

```
import pandas as pd
isikud_seeria = pd.Series({'Võistleja nr 1': 'Mari', 'Võistleja nr
2': 'Joonas', 'Võistleja nr 3': 'Kati'})
print(isikud_seeria)
```

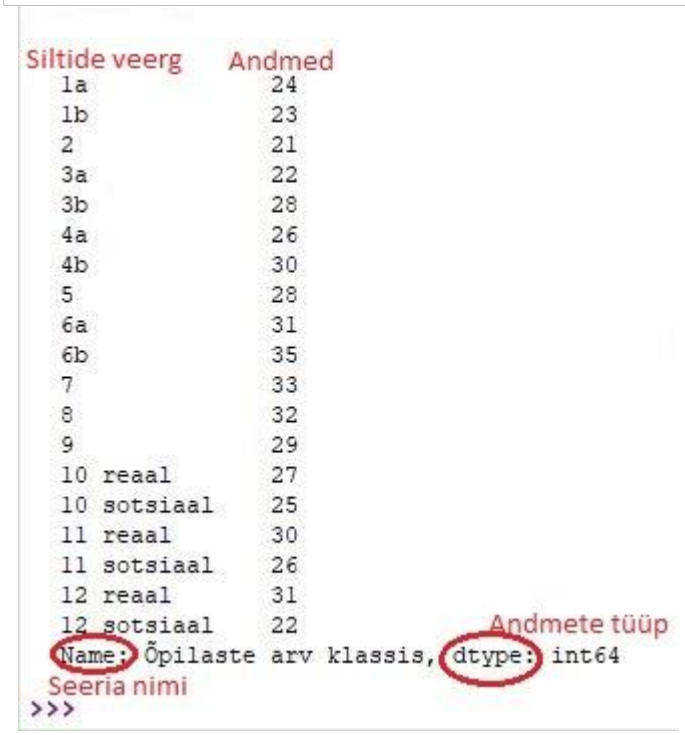
aga

```
Võistleja nr 1    Mari
Võistleja nr 2    Joonas
Võistleja nr 3    Kati
dtype: object
```

Sõnastikku tasub kasutada juhul, kui on soov andmeid võtmete ja väärtuste paaridena hoida ning pole vajadust edaspidise põhjalikuma andmete töötluse järele. Andmetöötluse jaoks sobib paremini seeria, sest sellel on rohkem funktsioone (aritmeetilised tehted, sorteerimine, filtreerimine jne). Tegevused toimuvad ka efektiivsemalt ning kood on lihtsam.

Näitena vaadeldakse seeriat, milles hoitakse õpilaste arvusid klasside kaupa.

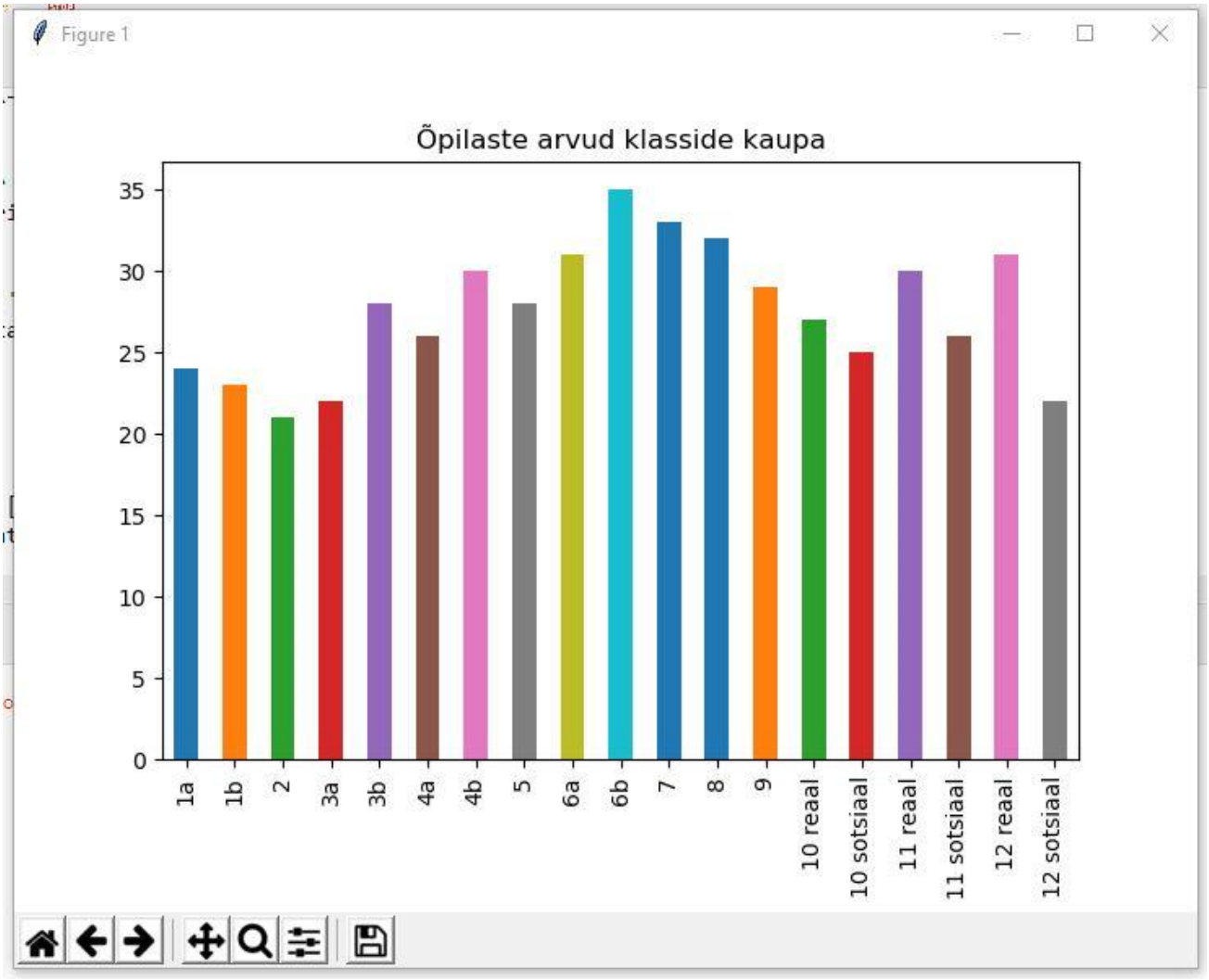
```
import pandas as pd
import matplotlib.pyplot as plot

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')
print(klassid)
```



Seeria puhul loetakse veeruks vaid andmete veergu ja seal asuvaid andmeid töödeldakse. Siltide veerg on vajalik konkreetsetele andmetele viitamiseks.

Illustreerime ülaltoodud andmeid graafikul.

```
klassid.plot.bar(title = 'Õpilaste arvud klasside kaupa')
plot.show()
```



Täpsemalt saab graafikute loomise ja kujundamise kohta infot materjalide lõpuosas esitatud näitest ja näiteks ingliskeelsest materjalist.

Sarnasus järjendi ja seeria vahel

Seeria on teatud mõttes sarnane tavalise järjendiga. Järgmises näites on püütud järjendile eriti sarnane seeria tekitada.

```
import pandas as pd
järjend = ['a', 'b', 'c', 'd']
print(järjend)
print('Element indeksiga 0 järjendis on:', järjend[0])
```

Ekraanile ilmub siis

```
['a', 'b', 'c', 'd']
Element indeksiga 0 järjendis on: a
```

Anname ka seerias samad indeksid

```
seeria = pd.Series(['a', 'b', 'c', 'd'], index = [0, 1, 2, 3])
print(seeria)
print('Element sildiga 0 seerias on:', seeria[0])
```

Nüüd saame ekraanile

```
0   a
1   b
2   c
3   d
dtype: object
Element sildiga 0 seerias on: a
```

Enesetest:

Mida väljastab selline koodilõik?

```
seeria = pd.Series(['a', 'b', 'c', 'd', 'e', 'f'],
index = [1, 3, 5, 7, 9, 11])
print(seeria[5])
```

- Vali c
- Vali f
- Vali Midagi muud
- Vali Veateate, sest sellist elementi pole

Nii seeria kui järjendi puhul on konkreetsele elemendile lisatud viide, mille järgi saab vajadusel elemendi kätte. Järjendis on elementidel järjekorranumbrid ehk indeksid `0, 1, 2, ..., n`. Seeria puhul on lisaks indeksitele tegemist siltidega (ingl *label*), mille kasutaja saab ise määrata. Sildid ei pruugi olla alati järjestikused ja isegi mitte arvud. Klasside õpilaste arvu näites polnud näiteks arvud.

Uue seeria loomine

Uue Series-tüüpi andmestruktuuri saab luua järgmiselt.

```
pandas.Series(data, index, dtype, copy)
```

Kõiki parameetreid ei pea defineerimisel kasutama, mingi parameetri ärajätmisel kasutatakse selle vaikimisi määratud väärtust. Kui näiteks ära jätta `data`, kuid lisada siltide järjend, siis genereeritakse seeria, mille igale sildile vastab spetsiaalne väärtus `NaN` (*Not a Number*). Lisaks tabelis olevatele parameetritele võib kasulikuks osutuda ka `name`, mille lisamisel saab seeriale määrata nime.

Järgmine tabel kirjeldab, mis tüüpi väärtused igale parameetrile saab anda ja mida need tähendavad.

Parameeter	Selgitus
`data`	Andmed võivad olla järjendi, sõnastiku või konstandi (üks element) kujul. Näiteks `pandas.Series({'Mari': '12', 'Karmo': '15', 'Malle': '20'}, ...)` või `pandas.Series(['punane', 'kollane', 'sinine'], ...)` või `pandas.Series('a', ...)`
`index`	Siltide järjend. Vaikimisi määratakse siltideks arvud 0, 1, 2,..., n-1, kus n on elementide arv parameetriga `data` lisatud andmestruktuuris.
`dtype`	Andmete tüüp, mida seerias hoitakse. Vaikimisi tuvastatakse tüüp antud andmetest. Näiteks kui hoitakse seerias ujukomaarve, siis vaikimisi määratakse tüübiks `float`.
`copy`	Määrab, kas andmetest, mida kasutatakse, tehakse koopia. Vaikimisi väärtus `false`.

Lisaks tabelis olevatele parameetritele, võib kasulikuks osutuda ka `name`, mille lisamisel saab seeriale määrata nime.

Vaatame ülaltoodud klasside ja õpilaste näidet uuesti ning analüüsime selle seeria loomist:

```
klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22], #andmed ehk data
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], #sildid
name = 'Õpilaste arv klassis') #seeriale antud nimi
```

Siltide arv (antud juhul klasside arv) peab olema võrdne parameetriga `data` antud andmestruktuuri elementide arvuga ehk õpilaste arvude arvuga. Antud juhul on meil tegemist 19 erineva klassiga ning järelikult ka õpilaste arvusid peab olema andmete järjendis 19. Parameetri `name` abil on seeriale lisatud nimeks “Õpilaste arv klassis”. Muudele parameetritele määrati väärtused vaikimisi.

Loodud seeria näeb ekraanile väljastatuna välja niisugune.

```
1a              24
1b              23
2               21
3a              22
3b              28
4a              26
4b              30
5               28
6a              31
6b              35
7               33
8               32
9               29
10 reaal        27
10 sotsiaal     25
11 reaal        30
11 sotsiaal     26
12 reaal        31
12 sotsiaal     22
Name: Õpilaste arv klassis, dtype: int64
```

Andmete tüübiks on vaikimisi määratud `int64`, sest õpilaste arvude näol on tegemist täisarvudega.

Enesetest:

Milline on väljundi viies rida?

```
seeria = pd.Series([1, 2, 3, 4, 5], index = [0, 2, 4,
6])
print(seeria)
```

Vali `5 NaN`

Vali `NaN 5`

Vali Viiendat rida polegi, sest programm on vigane.

Erandjuhuna saab seeria luua nii, et loomisel on lisatud vaid üks väärtus ja näiteks 4 silti. Sel juhul korratakse seda ühte väärtust 4 korda ning luuakse 4 sama väärtusega elementi.

```
import pandas as pd
seeria = pd.Series(7, index = [0, 2, 4, 6])
print(seeria)
```

Ekraanile saame

```
0   7
2   7
4   7
6   7
dtype: object
```

Sildid võivad olla arvud, sõned või mingit muud mittemuteeritavat tüüpi.

```
seeria = pd.Series([7, 'Tartu Ülikool', 3.14, -
1789710578, 'Informaatika!'], index = ['A', '2', 'C', '4', 'E'])
print(seeria)
```

Väljund

```
A    7
2    Tartu Ülikool
C    3.14
4    -1789710578
E    Informaatika!
dtype: object
```

Ülaltoodud näidetes loodi seeriad kasutades järjendit või konstanti (ühte elementi). Lisatavad andmed võivad olla ka sõnastikus.

```
import pandas as pd
dict_isikud = {'Võistleja nr 1' : 'Mari','Võistleja nr
2': 'Joonas','Võistleja nr 3': 'Kati'}
seeria = pd.Series(dict_isikud)
print(seeria)
```

Väljund

```
Võistleja nr 1    Mari
Võistleja nr 2    Joonas
Võistleja nr 3    Kati
dtype: object
```

Selleks, et tutvuda osaga võimalustest, mida seeria kasutamine andmete töötlemiseks annab, võtame ette juba tuttava klasside õpilaste arvude näite.

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')


print(klassid)

1a             24
1b             23
2              21
3a             22
3b             28
4a             26
4b             30
5              28
6a             31
6b             35
7              33
8              32
9              29
10 reaal       27
10 sotsiaal    25
11 reaal       30
11 sotsiaal    26
12 reaal       31
12 sotsiaal    22
Name: Õpilaste arv klassis, dtype: int64
```

Oletame, et meil on vaja eraldada ainult algklasside õpilaste arvud. Seda saab teha kahel moel, kasutades andmete asukohta seerias või silte. Asukoha järgi saab andmeid eraldada seetõttu, et lisaks siltidele, on igal elemendil olemas ka indeks, mis tähistab numbriliselt elemendi asukohta seerias. Nagu ka järjendis, on indeksiteks arvud `0, 1, 2, ..., n-1`, kus n tähistab elementide arvu seerias.

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')
algklassid =
klassid[['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b']]
print(algklassid)
```

Ja teistmoodi

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')
algklassid = klassid[0:10]
print(algklassid)
```

Tulemus tuleb mõlemal juhul täpselt sama.

```
1a    24
1b    23
2     21
3a    22
3b    28
4a    26
4b    30
5     28
6a    31
6b    35
Name: Õpilaste arv klassis, dtype: int64
```

Järgmisena vaatame, mitu klassi kokku on meil seerias, selleks kasutame `.size` tunnust.

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')
print(klassid.size)
```

Väljastatakse `19`.

Tihti on vaja andmeid mingis kindlas järjekorras sorteerida, sorteerime klassid õpilaste arvude järgi kahanevas järjekorras funktsiooni `.sort_values()` abil. Samuti saab sorteerida ka siltide järgi funktsiooni `.sort_index()` abil.

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index =
['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7', '8',
 '9', '10 reaal', '10 sotsiaal', '11 reaal', '11 sotsiaal', '12
reaal', '12 sotsiaal'], name = 'Õpilaste arv klassis')
klassid_kahanevalt = klassid.sort_values(ascending=False)
print(klassid_kahanevalt)
6b             35
7              33
8              32
12 reaal       31
6a             31
11 reaal       30
4b             30
9              29
3b             28
5              28
10 reaal       27
11 sotsiaal    26
4a             26
10 sotsiaal    25
1a             24
1b             23
3a             22
12 sotsiaal    22
2              21
Name: Õpilaste arv klassis, dtype: int64
```

Mõnikord on vaja lisada seeriasse andmeid juurde. Vaatame, kuidas saab kahe seeria andmed ühendada. Lisame juurde õpetajate arvu, õpilaste arvu kokku ja klasside arvu kokku. Selleks loome uue `Series`-tüüpi andmestruktuuri, mille andmeteks on järjend õpetajate arvuga, õpilaste arvuga kokku ja klasside arvuga kokku ning mille siltideks on 'Õpetajate arv', 'Õpilaste arv kokku' ja 'Klasside arv kokku'. Õpilaste arvu kokku saame kasutades `.sum()` funktsiooni klasside seeria peal ning klasside arvu kokku leiame atribuudiga `.size`, mida kasutasime ka varem.

```
import pandas as pd

klassid =
pd.Series([24, 23, 21, 22, 28, 26, 30, 28, 31, 35, 33, 32, 29, 27, 2
5, 30, 26, 31, 22],
index=['1a', '1b', '2', '3a', '3b', '4a', '4b', '5', '6a', '6b', '7'
, '8', '9', '10 reaal', '10 sotsiaal', '11 reaal', '11
sotsiaal', '12 reaal', '12 sotsiaal'], name='Õpilaste arv klassis')
uus_seeria = pd.Series([25, klassid.sum(), klassid.size],
index=['Õpetajate arv', 'Õpilaste arv kokku','Klasside arv kokku'])
klassid = klassid.append(uus_seeria)
print(klassid)
```

Tulemuseks on uuendatud klasside seeria. Oluline on meelde jätta, et funktsioonide rakendamisel seeria peal, tuleb uus ja modifitseeritud muutuja alati uuesti omistada, kas siis uuele muutujale või kirjutada vana seeria üle.

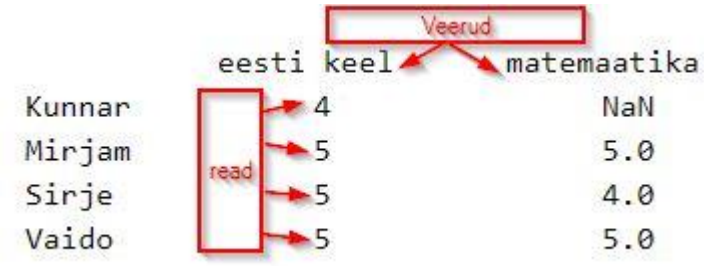
```
1a                    24
1b                    23
2                     21
3a                    22
3b                    28
4a                    26
4b                    30
5                     28
6a                    31
6b                    35
7                     33
8                     32
9                     29
10 reaal              27
10 sotsiaal           25
11 reaal              30
11 sotsiaal           26
12 reaal              31
12 sotsiaal           22
Õpetajate arv         25
Õpilaste arv kokku    523
Klasside arv kokku     19
dtype: int64
```

Andmefreim (`DataFrame`)

Juba seeria abil saab andmeid teatud määral töödelda ja illustreerida. Veelgi põnevamaid võimalusi pakub andmefreim (`DataFrame`), mis kujutab endast põhimõtteliselt tabelit. Seeria tähistab andmefreimis (`DataFrame`) ühte veergu. Andmefreimis võivad olla erinevat tüüpi andmed ja selle suurust saab muuta. Ridadele ja veergudele saab rakendada aritmeetilisi funktsioone - näiteks saab veergusid liita, lahutada, korrutada ja jagada ning arvutada rea või veeru kaupa selle maksimumi, keskmist jpm.

Esimese näitena vaatame andmefreimi, milles on õpilaste nimed ning eesti keele ja matemaatika hinded.

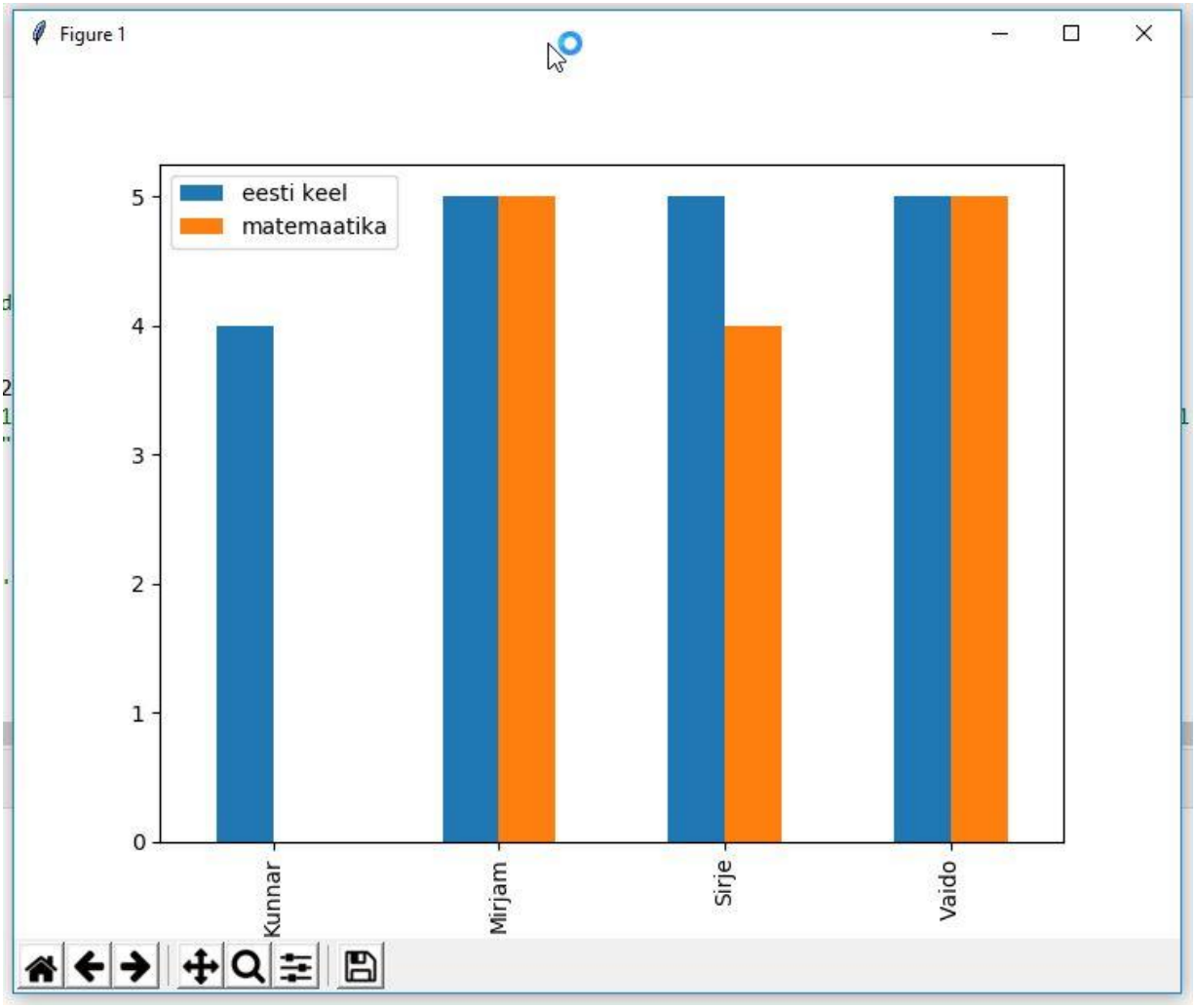
```
import pandas as pd
import matplotlib.pyplot as plot
opilased = { 'matemaatika': pd.Series([5, 4, 5], index =
['Mirjam', 'Sirje', 'Vaido']),
'eesti keel': pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)
print(opilased_tabel)
```



Andmefreimis ei loeta siltide veergu ja veergude pealkirjasid ridade ja veergude hulka. Erinevalt seeriast, vastab siin igale sildile üks rida. See koosneb põhimõtteliselt pealkirjaga seeriatest (veergudest), millel on ühised sildid.

Illustreerime ülaltoodud andmeid graafikul:

```
opilased_tabel.plot.bar()
plot.show()
```



Andmefreimi saab luua järgmiselt:

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

Ka andmefreimi puhul ei pea kõiki parameetreid defineerimisel kasutama, parameetri ärajätmisel kasutatakse selle vaikimisi määratud väärtust.

Parameeter	**Selgitus**
`data`	Andmed võivad olla n-dimensioonilise andmemassiivi, sõnastiku, konstandi, järjendi või teise andmefreimi kujul. Näiteks: `pandas.DataFrame([['Ruudi', 12], ['Malle', 10]], ...)`
`index`	Ridade siltide järjend, pikkus peab olema võrdne ridade arvuga. Vaikimisi määratakse arvud 0, 1, 2, ..., n-1, kus n on ridade arv.
`columns`	Veergude siltide järjend, pikkus peab olema võrdne veergude arvuga. Vaikimisi määratakse arvud 0, 1, 2, ..., m-1, kus m on veergude arv.

`dtype`	Andmetüüp, mida DataFrames olevatele andmetele määratakse. Vaikimisi järeldatakse andmetest.
`copy`	Määrab, kas andmetest, mida kasutatakse, tehakse koopia. Vaikimisi väärtus `False`.

Vaatame ülaltoodud klasside ja õpilaste näidet uuesti ning analüüsime selle seeria loomist:

```
import pandas as pd
opilased = { 'matemaatika': pd.Series([5, 4, 5], index =
['Mirjam', 'Sirje', 'Vaido']),
'eesti keel': pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)
print(opilased_tabel)
```

Sisendiks antakse sõnastik, mille võtmed on veeru pealkirjad ja vastavateks väärtusteks kaks seeriat (kaks veergu). Seeriate siltideks on õpilaste nimed ning andmeteks õpilaste poolt saadud hinded. Pange tähele, et mõlemas seerias on õpilaste arv võrdne hinnete arvuga, kuid lisatavad seeriad pole sama pikad. Muudele parameetritele määrati väärtused vaikimisi.

Rohkem näiteid andmefreimide loomisest:

Tühja andmefreimi loomine:

```
import pandas as pd
tühi = pd.DataFrame()
print(tühi)
```

Väljund:

```
Empty DataFrame
Columns: []
Index: []
```

Andmefreimi loomine järjendi põhjal:

Tegemist on järjendite järjendiga, mis antakse ette andmefreimi andmetena. `Columns` parameetriks antud järjend määrab ära veergude pealkirjad. Andmed jaotatakse veergudesse vastavalt nende paigutusele järjendites, iga järjend on eraldi rida ja sisemiste järjendite elemendid asuvad erinevates veergudes. Määratud on ka andmefreimis arvuna antud vanuste andmetüüp, selleks on `float`.

```
import pandas as pd
isikud = [['Juss',10],['Illar',12],['Mari-Liis',13]]
andmestruktuur = pd.DataFrame(isikud, columns = ['Nimi','Vanus'],
dtype=float)
print(andmestruktuur)
```

```
     Nimi         Vanus
0    Juss         10.0
1    Illar        12.0
2    Mari-Liis    13.0
dype: object
```

Andmefreimi loomine sõnastiku põhjal:

Sarnaselt ülaltoodud näitele on siingi sõnastiku võtmed veergude pealkirjadeks. Konkreetsetele võtmetele vastavad järjendid on aga veergudes väärtusteks. Andmefreimi loomisel on etteantud ka siltide järjend.

```
import pandas as pd
voistlejad = {'Nimi' :
['Malle', 'Kalle', 'Jaanus', 'Pille'], 'Vanus' : [24, 34, 51, 47]}
voistlejad_tabel = pd.DataFrame(voistlejad, index = ['Nr. 1', 'Nr.
2', 'Nr. 3', 'Nr. 4'])
print(voistlejad_tabel)
```

Väljund:

```
         Nimi    Vanus
Nr. 1    Malle     24
Nr. 2    Kalle     34
Nr. 3    Jaanus    51
Nr. 4    Pille     47
```

Andmefreimi (`DataFrame`) loomine sõnastikust, milles on seeriad:

Ülaltoodud näide õpilastest ja hinnetest sobib hästi näiteks, kus andmefreim luuakse sõnastikust, milles on seeriad.

```
import pandas as pd
opilased = { 'matemaatika' : pd.Series([5, 4, 5],
index=['Mirjam', 'Sirje', 'Vaido']),
'eesti keel' : pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)
print(opilased_tabel)
```

Väljund:

```
        eesti keel     matemaatika
Kunnar           4           NaN
Mirjam           5           5.0
Sirje            5           4.0
Vaido            5           5.0
```

Matemaatika veerus oli andmeid vaid 3 inimese kohta, seega lisati Kunnari matemaatika hindeks `NaN` ehk siis "pole arv" väärtus (ingl *not a number*). Sellest tulenevalt muudeti selle veeru väärtused ujukomaarvudeks (ingl *float*), sest `NaN` väärtust loetakse ujukomaarvuks.

Edasi tutvume mõnede andmefreimi võimalustega ja selleks on jällegi hea kasutada õpilaste ja hinnete näidet.

```
import pandas as pd
opilased = { 'matemaatika': pd.Series([5, 4, 5], index =
['Mirjam', 'Sirje', 'Vaido']),
'eesti keel': pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)
print(opilased_tabel)
```

Oletame, et pandi välja veel mitme aine hinded ning vaja oleks lisada ka need tabelisse.

```
import pandas as pd
opilased = { 'matemaatika': pd.Series([5, 4, 5], index =
['Mirjam', 'Sirje', 'Vaido']),
'eesti keel': pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)
print(opilased_tabel)
```

Algse tabeli kuju:

```
            eesti keel  matemaatika
Kunnar           4           NaN
Mirjam           5           5.0
Sirje            5           4.0
Vaido            5           5.0
```

Selleks loome sama põhimõtte järgi uue andmefreimi ning ühendame `.join()` funktsiooni abil kaks andmefreimi omavahel.

```
import pandas as pd
lisa = { 'inglise keel' : pd.Series([5, 4, 5, 5, 3], index =
['Mirjam', 'Sirje', 'Vaido','Karmen','Kunnar']),
             'kehaline kasvatus' : pd.Series([5, 5, 5, 4, 5],
index = ['Mirjam', 'Sirje', 'Vaido', 'Karmen', 'Kunnar'])}

lisa_tabel = pd.DataFrame(lisa)
```

Uute andmetega tabeli kuju:

```
             inglise keel  kehaline kasvatus
Mirjam              5                  5
Sirje               4                  5
Vaido               5                  5
Karmen              5                  4
Kunnar              3                  5
```

Näeme, et lisatavas tabelis on rohkem õpilasi, seega peame ühendama algse õpilaste tabeli uue tabeliga, et kõik nimed oleksid olemas ka uues tabelis.

```
import pandas as pd
opilased = { 'matemaatika': pd.Series([5, 4, 5], index =
['Mirjam', 'Sirje', 'Vaido']),
'eesti keel': pd.Series([5, 5, 5, 4], index =
['Mirjam', 'Sirje', 'Vaido', 'Kunnar'])}
opilased_tabel = pd.DataFrame(opilased)

lisa = { 'inglise keel' : pd.Series([5, 4, 5, 5, 3], index =
['Mirjam', 'Sirje', 'Vaido','Karmen','Kunnar']),
             'kehaline kasvatus' : pd.Series([5, 5, 5, 4, 5],
index = ['Mirjam', 'Sirje', 'Vaido', 'Karmen', 'Kunnar'])}
lisa_tabel = pd.DataFrame(lisa)

uus_tabel = lisa_tabel.join(opilased_tabel)
print(uus_tabel)
```

Uue tabeli kuju:

```
             inglise keel  kehaline kasvatus  eesti keel  matemaatika
Mirjam              5                  5           5.0          5.0
Sirje               4                  5           5.0          4.0
Vaido               5                  5           5.0          5.0
Karmen              5                  4           NaN          NaN
Kunnar              3                  5           4.0          NaN
```

Lähenemas on õppeaasta lõpp ja lisanduvad ka viimased üksikud hinded, mis tuleb tabelisse lisada. Lisame hinded ühekaupa õigele kohale. Esimese parameetriga määrame aine (veeru), mille hinnet soovime muuta ning teise parameetriga õpilase nime (rea):

```
uus_tabel['matemaatika']['Karmen'] = 3
uus_tabel['eesti keel']['Karmen'] = 5
uus_tabel['matemaatika']['Kunnar'] = 3
```

Kuna keskmisest hindest oleneb ekskursioonile minek, soovime välja arvutada keskmised hinded ja lisada need uude veergu "Keskmine hinne". Lisame uue veeru ja arvutame õpilaste keskmised hinded. Õige rea valimiseks kasutame tunnust `.loc` ning keskmise arvutamiseks reas funktsiooni `.mean()`:

```
uus_tabel['Keskmine hinne'] =
pd.Series([uus_tabel.loc['Sirje'].mean(),
uus_tabel.loc['Mirjam'].mean(),
                                        uus_tabel.loc['Vaido'].mean(
), uus_tabel.loc['Karmen'].mean(), uus_tabel.loc['Kunnar'].mean()],
index = ['Sirje', 'Mirjam', 'Vaido', 'Karmen', 'Kunnar'])
```

Pane tähele, et uude veergu lisatavad keskmised peavad olema siltide järjendis olevate siltidega samas järjekorras, siis saab õige hinne lisatud õige õpilase juurde.

Tabel, milles on puuduvad hinded ja keskmised:

```
        inglise keel  kehaline kasvatus  eesti keel  matemaatika  \
Mirjam             5                   5         5.0          5.0
Sirje              4                   5         5.0          4.0
Vaido              5                   5         5.0          5.0
Karmen             5                   4         5.0          3.0
Kunnar             3                   5         4.0          3.0
```

Järgmises peatükis leiate veelgi näiteid andmetöötluse jaoks kasutatavatest andmefreimi funktsioonidest.

5.5 TEATRIKÜLASTUSTE NÄIDE

Materjal on veel toorevõitu. Palun teatage parandusettepanekutest vastavas foorumis!!!

Sissejuhatus

Eelmises peatükis tutvustati Pandase põhilisi andmestruktuure - seeriat (`Series`) ja andmefreimi (`DataFrame`). Nüüd tegutseme andmefreimi abil põhjalikumalt tegelike andmetega. Kui siin on vaatluse all teatrikülastused, siis ülesanne tuleb raamatukogude kohta.

Senini oleme vaadanud olukordi, kus me defineerime mõnes teises andmestruktuuris asuvate algandmete järgi andmefreimi (`DataFrame`). Selleks, et säästa end andmete ümberkirjutamisest, saab andmeid ka otse failist andmefreimi (`DataFrame`) laadida.

Siin näete lühikest videot sellest, kuidas andmeid statistikaametist alla laadida ning neid töödelda.

Andmete saamine failist

Kasutades andmetöötluseks Pythoni moodulit Pandas, saab andmed lihtsalt sisse lugeda näiteks csv-failist. Võtame näiteks faili KU086.csv, mille sisu näeb tavalise tekstiredaktoriga (nt notepad) avades välja selline:

```
 ;2010;2011;2012;2013;2014;2015;2016
Teatrite arv;29;34;41;41;37;49;46
Lavastused;417;464;487;490;511;550;540
..uuslavastused;173;190;203;186;196;216;196
Etendused;4593;5012;5678;5803;6010;6434;6573
Vaatajad, tuhat;899.9;1008.3;1143.0;1090.7;1047.1;1146.6;1186.0
Teatrisk�igud 1000 elaniku
kohta;671.5;752.5;864.1;827.5;796.6;872.2;901.4
```

Ärgem laskem end hetkel segada moondunud ä-tähest. Hiljem tegeleme ka sellega.

```
import pandas as pd
andmed = pd.read_csv('KU086s.csv', delimiter=';')
print(andmed)
```

Muutuja `delimiter` väärtus näitab, milline eraldaja on andmeid sisaldavas failis määratud, antud juhul on tegemist semikooloniga. Seda, missugust eraldajat kasutatakse, saab näiteks statistikaameti andmebaasis valida andmete salvestamisel, kuid vajadusel võib andmefaili teisele kujule ka ümber teisendada.

Funktsiooni `read_csv` kasutamisel on veel mitmeid muid parameetreid, mida võib vaja minna. Nendega saab lähemalt tutvuda siin.

Saame väljundiks:

```
                                                  2012    2013    2014  \
0                                       Raamatukogud   559.0   556.0   549.0
1                                     Lugejad, tuhat   383.2   377.7   368.1
2                Lugejaid keskmiselt raamatukogu kohta   686.0   679.0   670.0
3  Laenutusi keskmiselt lugeja kohta, arvestusuksust    30.5    29.6    29.1
4                                           Tootajad  1554.0  1544.0  1528.0

     2015    2016
0   540.0   536.0
1   363.4   351.3
2   673.0   655.0
3    28.6    28.5
4  1471.0  1442.0
```

Tutvumine andmetega

Püüame täpsemalt tutvuda meie poolt sisse loetud tabeliga. Kuna sisseloetud tabel on meil nüüd `DataFrame`-kujul, siis saame kasutada pakutavaid Pandase võimalusi. Püüame näiteks teada saada, mitu veergu ja rida on tabelis.

Tunnus `DataFrame.shape` annab andmefreimi mõõtmed:

```
print(andmed.shape)

(6, 8)
```

```
# Nii saab teada, mitu veergu meil tabelis on.
# Veergude arv asub tunnuse shape väljundi teisel positsioonil
print("Tabelis on ", andmed.shape[1], " veergu.")

Tabelis on  8  veergu.

----------------------------------

# Veergude pealkirjad
print(andmed.columns)

Index([' ', '2010', '2011', '2012', '2013', '2014', '2015', '2016'],
dtype='object')

----------------------------------

# Nii saab teada, mitu rida meil tabelis on:
# Ridade arv asub tunnuse shape väljundi esimesel positsioonil
print("Tabelis on ", andmed.shape[0], " rida.")

Tabelis on 6 rida.

----------------------------------
```

```
# Prindime 2 esimest rida
print(andmed.head(2))

                  2010   2011   2012   2013   2014   2015   2016
0  Teatrite arv   29.0   34.0   41.0   41.0   37.0   49.0   46.0
1    Lavastused  417.0  464.0  487.0  490.0  511.0  550.0  540.0

---------------------------------

# Prindime 2 viimast rida
print(andmed.tail(2))

                                    2010    2011    2012    2013    2014  \
4                   Vaatajad, tuhat  899.9  1008.3  1143.0  1090.7  1047.1
5  Teatrisk?igud 1000 elaniku kohta  671.5   752.5   864.1   827.5   796.6

     2015    2016
4  1146.6  1186.0
5   872.2   901.4
```

Andmete töötlemise funktsioonid

Näeme, et hetkel on eraldi veerud siltidega ja veerg, mille väärtused võiksid tegelikult siltidena kasutuses olla.

	Soovitud siltide veerg	2010	2011	2012	2013
0	Teatrite arv	29.0	34.0	41.0	41.0
1	Lavastused	417.0	464.0	487.0	490.0
2	..uuslavastused	173.0	190.0	203.0	186.0
3	Etendused	4593.0	5012.0	5678.0	5803.0
4	Vaatajad, tuhat	899.9	1008.3	1143.0	1090.7
5	Teatrisk?igud 1000 elaniku kohta	671.5	752.5	864.1	827.5

Muudame soovitud veeru siltide veeruks. Veergude pealkirjade järjendist näeme, et see veerg on pealkirjaga " ".

```
andmed = andmed.set_index(' ')

# Prindime 5 esimest tabeli rida, et kontrollida.
print(andmed.head())

                   2010    2011    2012    2013    2014    2015    2016

Teatrite arv       29.0    34.0    41.0    41.0    37.0    49.0    46.0
Lavastused        417.0   464.0   487.0   490.0   511.0   550.0   540.0
..uuslavastused   173.0   190.0   203.0   186.0   196.0   216.0   196.0
Etendused        4593.0  5012.0  5678.0  5803.0  6010.0  6434.0  6573.0
Vaatajad, tuhat   899.9  1008.3  1143.0  1090.7  1047.1  1146.6  1186.0
```

Sama tulemuse saaksime ka andmete sisselugemisel parameetrit `index_col` kasutades.

```
andmed = pd.read_csv('KU086s.csv', delimiter=';', index_col=' ')
```

Vahel on vaja lisada andmeid tabelisse juurde. Seda saab teha veergude või ridade lisamise abil. Lisame veeru, kus on arvud 1-6.

```
andmed['Uus veerg'] = [1, 2, 3, 4, 5, 6]
print(andmed)
```

```
                                2010    2011    2012    2013    2014  \

Teatrite arv                    29.0    34.0    41.0    41.0    37.0
Lavastused                     417.0   464.0   487.0   490.0   511.0
..uuslavastused                173.0   190.0   203.0   186.0   196.0
Etendused                     4593.0  5012.0  5678.0  5803.0  6010.0
Vaatajad, tuhat                899.9  1008.3  1143.0  1090.7  1047.1
Teatrisk?igud 1000 elaniku kohta   671.5   752.5   864.1   827.5   796.6

                                2015    2016  Uus veerg

Teatrite arv                    49.0    46.0          1
Lavastused                     550.0   540.0          2
..uuslavastused                216.0   196.0          3
Etendused                     6434.0  6573.0          4
Vaatajad, tuhat               1146.6  1186.0          5
Teatrisk?igud 1000 elaniku kohta   872.2   901.4          6
```

Kui on vaja lisada väärtused vastavatesse kindla sildiga ridadesse, saab seda teha uue seeria (Series) loomise ning selle tabelisse lisamise abil.

```
andmed['Uus veerg'] = pd.Series([1,2,3,4,5,6], index =
['Lavastused', 'Teatrite arv', '..uuslavastused', 'Etendused', 'Vaatajad,
tuhat', 'Teatrisk?igud 1000 elaniku kohta'])
print(andmed)
```

```
                                2010    2011    2012    2013    2014  \

Teatrite arv                    29.0    34.0    41.0    41.0    37.0
Lavastused                     417.0   464.0   487.0   490.0   511.0
..uuslavastused                173.0   190.0   203.0   186.0   196.0
Etendused                     4593.0  5012.0  5678.0  5803.0  6010.0
Vaatajad, tuhat                899.9  1008.3  1143.0  1090.7  1047.1
Teatrisk?igud 1000 elaniku kohta   671.5   752.5   864.1   827.5   796.6

                                2015    2016  Uus veerg

Teatrite arv                    49.0    46.0          2
Lavastused                     550.0   540.0          1
..uuslavastused                216.0   196.0          3
Etendused                     6434.0  6573.0          4
Vaatajad, tuhat               1146.6  1186.0          5
Teatrisk?igud 1000 elaniku kohta   872.2   901.4          6
```

Nüüd aga kustutame loodud veeru. Kasutame selleks 3 erinevat funktsiooni: drop(), pop() ja del.

pop()

```
# pop() funktsioon tagastab eemaldatud veeru, eemaldab veeru
tabelist, ei vaja uuesti omistamist

eemaldatud = andmed.pop('Uus tulp')
print(eemaldatud)
```

```
Teatrite arv                          2
Lavastused                            1
..uuslavastused                       3
Etendused                             4
Vaatajad, tuhat                       5
Teatrisk?igud 1000 elaniku kohta      6
Name: Uus tulp, dtype: int64
```

drop()

```
# drop() funktsioon tagastab uuendatud tabeli, milles on märgitud veerg
eemaldatud, vajab uuesti omistamist, sest protseduuri tehakse tabeli koopia
peal, veeru eemaldamiseks tuleb kasutada parameetrit axis=1

andmed = andmed.drop(['Uus tulp'], axis=1)
print(andmed)
```

```
                                  2010    2011    2012    2013    2014  \

Teatrite arv                       29.0    34.0    41.0    41.0    37.0
Lavastused                        417.0   464.0   487.0   490.0   511.0
..uuslavastused                   173.0   190.0   203.0   186.0   196.0
Etendused                        4593.0  5012.0  5678.0  5803.0  6010.0
Vaatajad, tuhat                   899.9  1008.3  1143.0  1090.7  1047.1
Teatrisk?igud 1000 elaniku kohta   671.5   752.5   864.1   827.5   796.6

                                  2015    2016

Teatrite arv                       49.0    46.0
Lavastused                        550.0   540.0
..uuslavastused                   216.0   196.0
Etendused                        6434.0  6573.0
Vaatajad, tuhat                  1146.6  1186.0
Teatrisk?igud 1000 elaniku kohta   872.2   901.4
```

Proovime tabelist ka rea eemaldada. Teeme seda algul indeksipõhiselt ja seejärel eemaldame rea, mis vastab mingile määratud kriteeriumile.

drop()

```
andmed = andmed.drop(['Teatrite arv'], axis=0)
print(andmed)

2010    2011    2012    2013    2014  \

Lavastused                          417.0   464.0   487.0   490.0   511.0
..uuslavastused                     173.0   190.0   203.0   186.0   196.0
Etendused                          4593.0  5012.0  5678.0  5803.0  6010.0
Vaatajad, tuhat                     899.9  1008.3  1143.0  1090.7  1047.1
Teatrisk?igud 1000 elaniku kohta    671.5   752.5   864.1   827.5   796.6

                                    2015    2016

Lavastused                          550.0   540.0
..uuslavastused                     216.0   196.0
Etendused                          6434.0  6573.0
Vaatajad, tuhat                    1146.6  1186.0
Teatrisk?igud 1000 elaniku kohta    872.2   901.4
```

ix()

Eemaldame read, kus on 2012. aasta väärtused väiksemad või võrdsed arvust 1000.

```
andmed = andmed.ix[df['2012'] >= 1000]
print(andmed)

                  2010    2011    2012    2013    2014    2015    2016

Etendused         4593.0  5012.0  5678.0  5803.0  6010.0  6434.0  6573.0
Vaatajad, tuhat    899.9  1008.3  1143.0  1090.7  1047.1  1146.6  1186.0
```

Muudame indeksi Teatrisk?igud 1000 elaniku kohta nimetust, et imeliku ä-tähe asendaja asemel päris ä-täht saada.

```
andmed = andmed.rename(index={'Teatrisk?igud 1000 elaniku kohta' : 'Teatriskäigud
1000 elaniku kohta'})
print(andmed.tail(1))
                                 2010   2011   2012   2013   2014   2015   2016

Teatriskäigud 1000 elaniku kohta  671.5  752.5  864.1  827.5  796.6  872.2  901.4
```

Andmete kajastamine graafikul

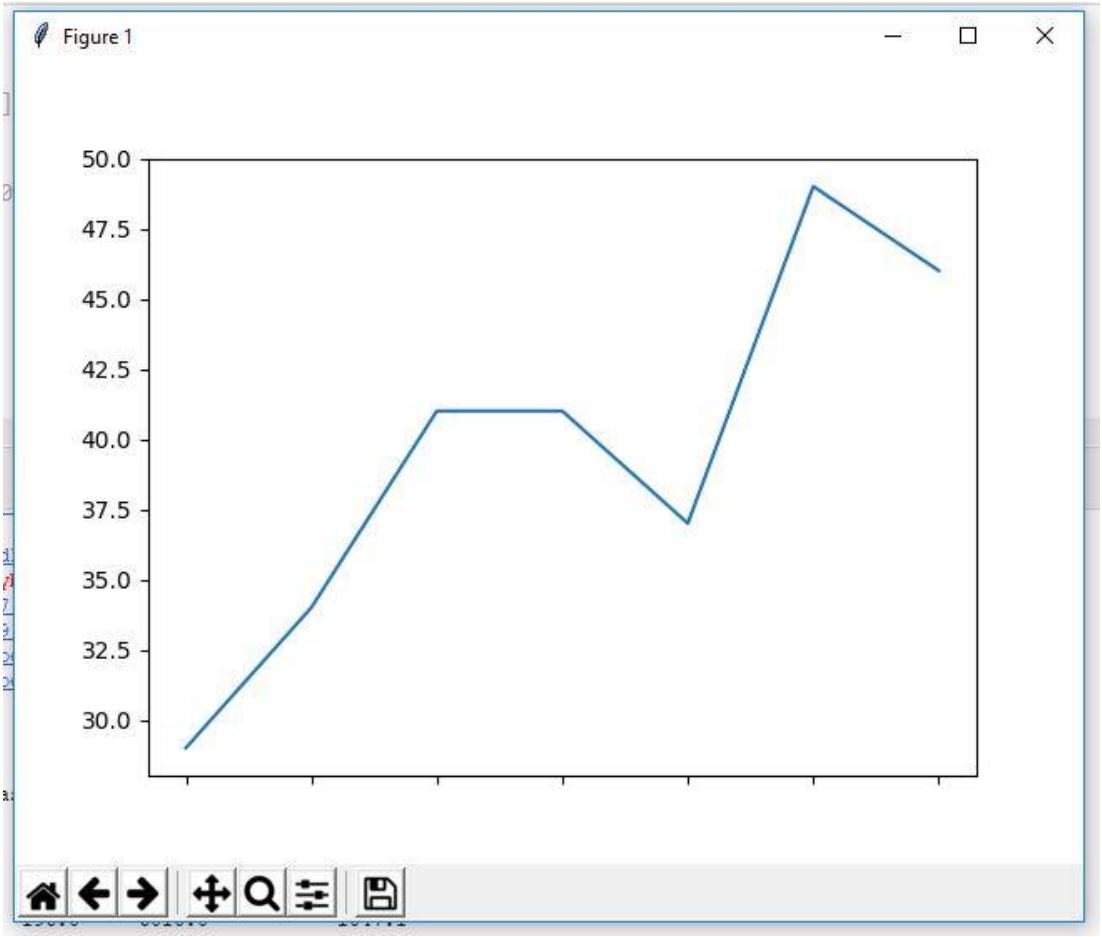
Andmete põhjal saab graafiku teha mooduli matplotlib abil. Moodul tuleb installide enne importimist. Sellega tegelesime eelmises peatükis.

```
# Impordime mooduli ja tekitame oma andmete kohta graafiku.
import matplotlib.pyplot as plot

# Transponeerime andmed, muudame veergude pealkirjad indeksiteks ja
indeksid veergude pealkirjadeks.
# Näitame graafikul, kuidas muutus aastate jooksul teatrite arv
transponeeritud_andmed = andmed.T
transponeeritud_andmed['Teatrite arv'].plot()
plot.show()
```

Transponeeritud andmed:

```
                0           1                2          3                4  \
      Teatrite arv  Lavastused  ..uuslavastused  Etendused  Vaatajad, tuhat
2010            29         417              173       4593            899.9
2011            34         464              190       5012           1008.3
2012            41         487              203       5678             1143
2013            41         490              186       5803           1090.7
2014            37         511              196       6010           1047.1
2015            49         550              216       6434           1146.6
2016            46         540              196       6573             1186

                                    5
      Teatrisk�igud 1000 elaniku kohta
2010                             671.5
2011                             752.5
2012                             864.1
2013                             827.5
2014                             796.6
2015                             872.2
2016                             901.4
```

Graafik:



Andmete kirjutamine faili

Kui andmed on töödeldud, võib tekkida vajadus need uude faili kirjutada. Seda saab teha funktsiooni `.to_csv` abil. Kirjutame muudetud andmed faili nimega uued_andmed.csv, kasutame eraldajana semikoolonit.

```
andmed.to_csv('uued_andmed.csv', sep=';', encoding='utf-8')
```

5.6 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.