3.1 ENNIK

Järjend

Erinevates programmeerimiskeeltes on mõnevõrra erinevad andmestruktuurid üht tüüpi muutujate/väärtuste koos käsitlemiseks. Paljudes keeltes on kõige elementaarsem andmestruktuur *massiiv* (ingl *array*), mis on sisuliselt fikseeritud pikkusega nimekiri sama tüüpi väärtustest. Pythonis massiivi selle mõiste klassikalises tähenduses pole, vaid sama rolli täidab dünaamilisem ja sõbralikum *järjend* (ingl *list*). Järjend on muutuva pikkusega ja selles saab hoida ka eri tüüpi väärtuseid. Järjendiga oleme juba palju tegelenud ja tema kohta teame nii mõndagi:

- Järjendi tähistamiseks või tekitamiseks kasutatakse nurksulge `[` ja `]`.
- Järjendi elementide järjestus on oluline: `[1, 2]` ei ole võrdne järjendiga `[2, 1]`.
- Järjendi elementide poole saab pöörduda indeksite abil: kui `lst` on järjend, siis `lst[2]` on selle järjendi kolmas element.
- Järjendi pikkuse saame teada funktsiooniga `len`.
- Mingi elemendi järjendisse kuulumist saame kontrollida võtmesõna `in` abil: `3 in lst` väärtuseks on kas `True` või `False`.
- Funktsioonide `min` ja `max` abil saame leida järjendist vastavalt minimaalse või maksimaalse elemendi (eeldusel, et elemendid on mingil moel järjestatavad, nt arvud).
- Järjendit saab viilutada: `lst[1:5]` annab alamjärjendi, milles on `lst` elemendid indeksitega 1, 2, 3 ja 4.

Ennik

Lisaks järjendile on Pythonis olemas sarnane andmestruktuur, mille nimi on *ennik* (ingl *tuple*). Naljakas eestikeelne nimi on tuletatud sellest, et kolme elemendiga kogumit nimetatakse kolmikuks, nelja elemendiga nelikuks, viiega viisikuks jne. Kui sellises struktuuris on suvaline arv *n* elementi, siis võiks saadut nimetada nimega *n-ik* ehk eestipärasemalt *ennik*.

Enniku tähistamiseks kasutatakse nurksulgude asemel ümarsulgusid `(` ja `)`. Näiteks kolmeelemendilise enniku ehk kolmiku saame tekitada nii:

```
kolmik = (1, 4, 6)
```

Üheelemendilise enniku (üksiku) tekitamine on pisut ebaintuitiivsem:

```
yksik = (1,)
```

Koma ainukese elemendi järel ütleb Pythonile, et tegemist on ennikuga, mitte aritmeetilise avaldisega, kus ümarsulud omavad teist tähendust.

Tuleb välja, et kõik eelnevalt loetletud järjendi omadused (peale esimese) kehtivad ka enniku puhul:

- Ennikus on elementide järjestus oluline: `(1, 2)` ei ole võrdne ennikuga `(2, 1)`.
- Enniku elementide poole saab pöörduda indeksite abil: kui `enn` on ennik, siis `enn[2]` on selle enniku kolmas element.
- Enniku pikkuse saame teada funktsiooni `len` abil.
- Mingi elemendi ennikusse kuulumist saame kontrollida võtmesõna `in` abil: `3 in enn` väärtuseks on kas `True` või `False`.
- Funktsioonide `min` ja `max` abil saame leida ennikust vastavalt minimaalse või maksimaalse elemendi (eeldusel, et elemendid on mingil moel järjestatavad, nt arvud).
- Ennikut saab viilutada: `enn[1:5]` annab alamenniku, milles on `enn` elemendid indeksitega 1, 2, 3 ja 4.

Siiski pole järjend ja ennik funktsionaalsuse mõttes võrdsed. Enniku kõige olulisem erinevus peitub selles, et see ei ole muudetav (täpsemalt *muteeritav*, ingl *mutable*). Ennikusse ei ole pärast selle tekitamist võimalik elemente lisada, neid eemaldada ega muuta, seega pole ennikul näiteks järjendite tuntud käske `append`, `remove`, `pop`, `insert`, `reverse` ega `extend`. Sellest tuleneb, et ennik on ka alati fikseeritud pikkusega.

Ennikud kasutatakse sageli just siis, kui struktuuri pole vaja programmi käivitamise ajal muuta. Näiteks kui programmis on vaja tähistada objekte, millel on mitu omadust, kuid mida pole vaja programmi töötamise ajal muuta, siis on mõistlik nende esitamiseks kasutada ennikuid. Samuti kasutatakse ennikuid tihti siis, kui funktsioonist on vaja tagastada mitu väärtust.

Näiteks see funktsioon leiab kahest järjendist elemendid, mille korrutis on suurim ja tagastab need elemendid paarina:

```
def suurim_korrutis(lst1, lst2):
    suurimad = (0, 0)
    for el1 in lst1:
        for el2 in lst2:
            if el1 * el2 > suurimad[0] * suurimad[1]:
                suurimad = (el1, el2)
    return suurimad
```

Kui aga ühe funktsiooni asemel, mis tagastab enniku tulemustest, on võimalik teha mitu funktsiooni, millest igaüks leiab ühe üksiku (väiksema) osa tulemustest, siis on parem stiil valida just see mitme funktsiooniga variant. Mida väiksemad ülesanded saate eri funktsioonide vahel ära jaotada, seda lihtsam on tavaliselt koodist aru saada, sealt vigu otsida ja seda hallata.

3.2 HULK

Veel üks Pythoni andmestruktuur, kuhu saab mitmeid elemente paigutada, on *hulk* (ingl *set*). Hulga tähistamiseks kasutatakse loogelisi sulgusid `{` ja `}`. Teise andmestruktuuri põhjal saab hulga teha funktsiooniga `set`.

```
h1 = {8, 2, 3, 6, 7}
h2 = set([6, 4, 5])
h3 = set('Tere hommikust')
```

Tühja hulga loomiseks tuleb kasutada `set()`, sest tühjad loogelised sulud `{}` tähistavad hoopis tühja sõnastikku, mida vaatame järgmises peatükis.

Nagu matemaatilises hulgaski, pole Pythoni hulgas elementide järjekord oluline. Nii võib ka Python hulga elemendid teises järjekorras esitada kui need sisestatud on.

```
>>> {8, 6, 4}
{8, 4, 6}
```

See tähendab ka muuhulgas, et kaks hulka on võrdsed siis, kui nad sisaldavad samu elemente. Need ei pea olema samas järjekorras (tegelikult polegi elementidel järjekorda).

```
>>> {42, 11} == {11, 42}
True
```

Kuna elementide järjekord pole määratud, siis ei saa ka üksikut elementi indeksi abil kätte. Küll aga saame for-tsükli abil hulga kõigi elementidega midagi ette võtta, hoolimata sellest, "mitmendat" elementi hetkel käsitleme:

```
for el in h3:
    print(el)
```

Teine oluline hulga tunnus pärineb samuti matemaatilistelt hulkadelt: hulgas ei ole korduvaid elemente. Kui püüame elemente hulga sees korrata, siis arvestatakse neid ikkagi ühekordselt.

```
>>> {4, 6, 7, 4}
{4, 6, 7}
```

Hulgad on muteeritavad: neid saab muuta elemente lisades ja eemaldades. Kui järjendi puhul oli elemendi juurde lisamiseks käsk `append`, mis tähendab lõppu lisamist, siis hulga puhul "lõpp" ei tähenda midagi ning seepärast on võetud elementide lisamiseks kasutusele hoopis käsk `add`. See lisatakse sarnaselt `append`-ile hulga lõppu punktiga:

```
>>> h = {42, 11}
>>> h.add(10)
>>> h
{10, 42, 11}
```

Veel mõned hulkade peal töötavad käsud on toodud järgmises tabelis:

Käsk	Selgitus
`h.remove(el)`	eemaldab elemendi `el` hulgast `h`
`h.update(h1)`	täiendab hulka `h` teise hulga `h1` elementidega
`h.pop()`	eemaldab ja tagastab hulgast `h` (juhusliku) elemendi
`h.clear()`	tühjendab hulga `h`
`h.copy()`	tagastab hulga `h` koopia

Hulga suurust saab `len` funktsiooni abil nagu järjendeid ja ennikuidki. Samuti saab hulgas sisalduvust kontrollida analoogselt:

```
>>> h = {42, 11}
>>> len(h)
2
>>> 42 in h
True
```

Ka paljud muud funktsioonid nagu `min` ja `max` töötavad hulkadega samamoodi nagu järjendite või ennikutega. Proovige ise järgi!

Hulkadega saab ka selliseid operatsioone teha, mida järjendite või ennikutega ei saa. Tähtsaimad neist on toodud järgnevas tabelis. Kõik need operatsioonid on binaarsed (kahekohalised) ja vastava operatsiooni märk tuleb panna kahe hulga vahele.

Operatsioon	Selgitus
`&`	ühisosa ehk hulk elementidest, mida sisaldavad mõlemad hulgad
`|`	ühend ehk hulk elementidest, mida sisaldavad üks või teine või mõlemad hulgad
`-`	vahe ehk hulk elementidest, mida sisaldab esimene, aga ei sisalda teine hulk
`<=`	on (mitterange) alamhulk ehk kas esimene hulk on saadud teisest elemente eemaldades
`>=`	on (mitterange) ülemhulk ehk kas teine hulk on saadud esimesest elemente eemaldades

Proovige järgi, kuidas need operatsioonid töötavad!

3.3 SÕNASTIK

Oleme tuttavad järjendite ja ennikutega, mille puhul igal elemendil on indeks. Nii on näiteks järjendi `a = [2, 4, 7]` puhul elemendi 2 indeks 0, elemendi 4 indeks on 1 ja elemendi 7 indeks on 2. Elemente saab indeksite abil kätte: `a[0]`, `a[1]` ja `a[2]`. Enniku puhul on indekseerimine samasugune - näiteks `b = (2, 4, 7)` puhul `b[0]`, `b[1]` ja `b[2]`.

Selline täisarvudega indekseerimine on väga kasulik, aga nii mõnelgi juhul oleks hea, kui elementide asukohta struktuuris ei määraks mitte järjekorranumber, vaid mingi muu väärtus. Näiteks võiksime tekitada struktuuri, kus "elementideks" (väärtusteks) on inimeste vanused ja "indeksiteks" (võtmeteks) nende inimeste nimed. Sel juhul saaksime näiteks `vanused["Kersti"]` väärtuseks 46 ja `vanused["Jüri"]` väärtuseks 38. Selline andmestruktuur on Pythonis täiesti olemas ja selle nimi on *sõnastik*.

Sõnastik (ingl *dictionary*) on dünaamilise (muutuva) pikkusega andmestruktuur, mille elemendid on tegelikult erilised paarid. Nende paaride esimest liiget nimetatakse *võtmeks* ja teist liiget *väärtuseks*. Koodis esitatakse sõnastik loogeliste sulgude `{` ja `}` vahel nii, et võti-väärtus paarid (*kirjed*) on komadega eraldatud. Paaride sees on võti ja väärtus eraldatud kooloniga.

```
>>> vanused = {"Kersti": 46, "Jüri": 38, "Jevgeni": 30, "Mart": 67}
>>> vanused["Kersti"]
46
>>> vanused["Jüri"]
38
```

Võtmed ja väärtused võivad olla erinevat tüüpi. Nii ongi meil ka eelmises näites võtmed sõned ja väärtused täisarvud. Sõnastiku väärtused saavad olla mistahes tüüpi, kuid võtmed peavad tehnilistel põhjustel olema mittemuteerivat tüüpi. See tähendab, et võtmeteks sobivad näiteks arvud, sõned ja ennikud, aga ei sobi järjendid, hulgad ega teised sõnastikud.

Kui tahame sõnastikku andmeid lisada, siis saame seda teha lihtsa omistamisega:

```
>>> vanused["Margus"] = 33
>>> vanused
{'Jüri': 38, 'Mart': 67, 'Jevgeni': 30, 'Kersti': 46, 'Margus': 33}
```

Kui püüda lisada uut väärtust võtmega, mis on juba sõnastikus olemas, siis eelnev väärtus kirjutatakse üle:

```
>>> vanused["Jevgeni"]
30
>>> vanused["Jevgeni"] = 31
>>> vanused["Jevgeni"]
31
```

Sõnastikku kasutataksegi niimoodi, et otsitakse võtme järgi väärtust. Vastupidi väärtuse järgi võtit kätte saada pole sõnastikust otseselt võimalik.

Sõnastikus on võti-väärtus paarid järjestamata just nagu hulga puhulgi. Seega ei saa ka nendele paaridele kuidagi indeksite kaudu ligi.

Sõnastikuga saame tsükli abil opereerida mitut moodi. Sõnastike kirjetest saab for-tsükliga üle käia niimoodi:

```
vanused = {"Kersti": 46, "Jüri": 38, "Jevgeni": 30, "Mart": 67}
for voti, vaartus in vanused.items():
    print(voti, vaartus)
```

Siin `vanus.items()` tagastab hulga tüüpi objekti, mis koosneb paaridest (ennikutest), kus esimesel kohal on võti ja teisel sellele vastav väärtus. Tsükli sama sammu ajal saame seega kasutada nii võtit kui vastavat väärtust.

Tihti kasutatakse for-tsüklit sõnastiku peal hoopis nii, et igal tsükli sammul saab tsüklimuutuja väärtuseks järjekordse võtme sõnastikust:

```
vanused = {"Kersti": 46, "Jüri": 38, "Jevgeni": 30, "Mart": 67}
for voti in vanused:
    print(voti)
```

Kui viimast varianti kasutada, siis saaksime vastavale väärtusele ligi tavapäraselt: `vanused[voti]`. See on täpselt analoogiline tsüklitele, mis töötasid järjendite indeksite peal.

Sõnastiku kirjete arvu saab sarnaselt eelnevatele struktuuridele kätte `len` funktsiooni abil. Võtme sõnastikku kuuluvust saab kontrollida `in` operaatoriga:

```
vanused = {"Kersti": 46, "Jüri": 38, "Jevgeni": 30, "Mart": 67}
kolmikud = ("Sofia", "Maria", "Kersti")
for nimi in kolmikud:
    if nimi not in vanused:
        vanused[nimi] = 13

print(vanused)
```

See programm lisas sõnastikku uusi kirjeid ainult juhul kui need olemasolevat väärtust üle ei kirjuta.

Sõnastikust saab kirjeid kustutada `del` operaatori abil:

```
>>> vanused = {"Kersti": 46, "Jüri": 38, "Jevgeni": 30, "Mart": 67}
>>> del vanused["Jüri"]
>>> vanused
{'Mart': 67, 'Jevgeni': 30, 'Kersti': 46}
```

3.4 SÕNE

Sõne kui andmestruktuur

Oleme sõnedega juba oma kursuste päris algusosast alates tuttavad. Teame, et sõnet raamistavad ülakomad või jutumärgid. Teame, et sõnetüüpi tähistatakse `str` ning näiteks arvmuutuja väärtusele vastava sõnetüüpi väärtuse saame funktsiooniga `str`. Seda viimast on meil olnud tarvis, kui pikemas tekstis tahame arvmuutuja väärtust kasutada.

```
a = 16
print("Arvu väärtus on " + str(a))
```

Siin materjalis aga vaatleme sõnet kui andmestruktuuri ja võrdleme seda teiste andmestruktuuridega, eriti järjendi ja ennikuga. Nimelt on sõne järjendi ja ennikuga päris mitmes mõttes sarnane.

Järjendi ja enniku puhul saame teatud elemendi indeksi abil kätte.

```
järjend = [1, 2, 5, 6]
ennik = (1, 2, 5, 6)
print(järjend[3])
print(ennik[3])
```

Proovime sama ka sõne korral.

```
sõne ="1256"
print(sõne[3])
```

Mis tüüpi aga on sõne elemendid?

Enesetest:

Mis ilmub ekraanile?

```
sõne = "1256"
print(sõne[3])
```

Vali `<class 'char'>`

Vali `<class 'int'>`

Vali `<class 'str'>`

Vali Veateade

Mitmetes keeltes on tõesti olemas eraldi andmetüüp sümbolite/märkide jaoks (nt *char*). Pythonis, aga pole ja nii käsitletakse sõne elemente ühemärgiliste sõnedena.

Sarnaselt järjendile ja ennikule saame ka sõne puhul for tsükliga ühekaupa elementidele ligi, näiteks väljastamiseks.

```
for element in järjend:
    print(element)

for element in ennik:
    print(element)

for element in sõne:
    print(element)
```

Sama saab korraldada ka indeksite abil.

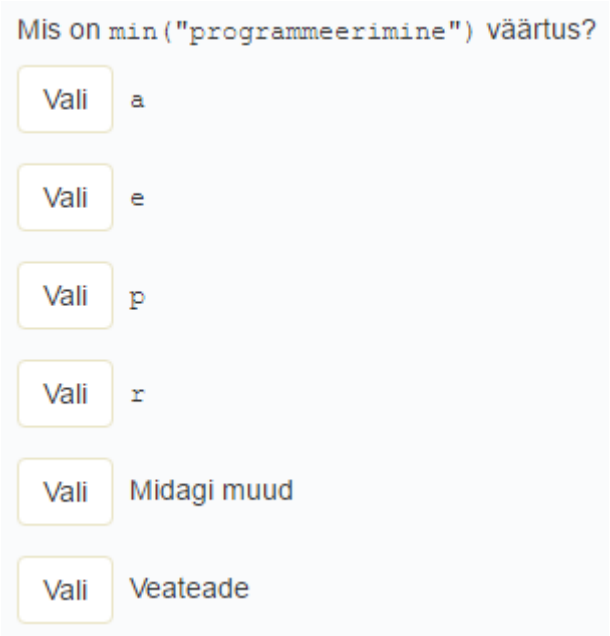
```
for i in range(len(järjend)):
    print(järjend[i])

for i in range(len(ennik)):
    print(ennik[i])

for i in range(len(sõne)):
    print(sõne[i])
```

Tsüklis võime elemente ka muidugi muul moel kasutada, kui ainult ekraanile toomiseks.

Näiteks võime iga elemendi korral leida, mitu korda ta üldse selles sõnes leidub.

```
sõne = "Kui Arno isaga koolimajja jõudis, olid tunnid juba alanud"
for märk in sõne:
    print(märk + " on sõnes " + str(sõne.count(märk)) + " korda")
```

Enesetest:



Sarnaselt saame järjendeid, ennikuid ja sõnesid ka viilutada.

```
print(järjend[1:3])
print(ennik[1:3])
print(sõne[1:3])
```

Kõik eelnevad näited töötasid sarnaselt sõne, järjendi ja enniku korral. Proovime nüüd neid võimalusi, kus järjend ja ennik erinevad, sõne puhul. Nimelt on erinevused selles osas, kas elemente saab muuta ja kas saab neid lisada ja eemaldada. Järjendi puhul saab muuta, lisada ja eemaldada, enniku puhul mitte.

Kuidas on lood sõnes elemendi väärtuse muutmisega?

Enesetest:

Mis ilmub ekraanile?

```
nimi = "Gerd"
nimi[3] = "t"
print(nimi)
```

- Vali `Gerd`
- Vali `Gert`
- Vali Midagi muud
- Vali Veateade

Proovigem nüüd sõne pikkust muuta.

Enesetest:

Mis ilmub ekraanile?

```
sõiduk = "Bus"
sõiduk.append("s")
print(sõiduk)
```

- Vali `Bus`
- Vali `Buss`
- Vali Midagi muud
- Vali Veateade

Samuti ei õnnestu sõne ja enniku puhul elemendi eemaldamine käsu `del` või funktsiooni `remove` abil.

Seega võib öelda, et sõne kui andmestruktuur sarnaneb pigem enniku kui järjendiga. Tegemist on muutmatute (mittemuteeritavate) andmestruktuuridega.

Selle muutmatuse väite valguses võib tunduda vastuolulisena näiteks funktsiooni `replace` tegevus, millega nagu saaks sõnet elemente asendades muuta.

```
nimi = "Gerd"
nimi2 = nimi.replace("d", "t")
print(nimi)
print(nimi2)
```

Tegelikult siiski esialgne sõne ei muutu.

3.5 ANDMED JA ANDMEBAASID

Senised kokkupuuted andmetega

Juba oma kursuste algusest peale oleme andmetega tegelenud - muutujad, erinevad andmetüübid olid praktiliselt esimesed teemad. Ka selle nädala üldteema on andmetega seotud - andmestruktuuridega nimelt. Käesolevas silmaringi materjalis käsitleme andmeid andmebaaside kontekstis.

Mäletatavasti kirjutasime algul andmed programmi sisse, hiljem juba kasutasime andmete jaoks eraldi faile. Juba sellel kursusel oli näiteid reaalsete andmete kasutamisest, konkreetselt tegelesime keskmiste temperatuuridega. Käesoleva kursuse osas 2.3 saime reaalseid andmeid statistikaameti veebilehelt ja Haridussilmast. Oluline on, et seda tabelit, mille me lõpuks allalaadisime ja oma programmiga avasime, ei olnud esialgu üldse sellisel kujul olemas. See alles konstrueeriti vastavalt meie valikutele olemasolevaid andmeid kasutades. Need olemasolevad andmed peavad olema mõistlikult organiseeritud, et kiiresti ja vigadeta vajalik üles leida.

Andmete hoidmiseks ja käsitsemiseks on kasutusel andmebaasid. Andmetega täidetud andmebaasidel põhinevad infosüsteemid ja registrid, mis tänapäeval kipuvad olema väga olulisel kohal. Näiteks Riigi Infosüsteemi otsingulehelt võib infosüsteeme, registreid jmt leida sadu.

Käesolevas silmaringimaterjalis tutvumegi pisut lähemalt andmete ja andmebaasidega. Tegemist on väga mitmekesise ja huvitava valdkonnaga, millest siin vaid põgusalt saame rääkida. Alustame andmebaaside ülesehitusest, siis räägime päringukeeltest ja lõpuks mõtiskleme üldse andmete ja informatsioonist meie ümber.

Üks või mitu tabelit

Andmete organiseerimiseks on väga erinevaid viise. Üks loomulik viis on püüda neid kujutada tabelina. Seda on ajalooliseltki palju kasutatud. On ju näiteks raamatu sisukord tabel, teame korrutustabelit ja keemiliste elementide perioodilisuse tabelit.

Kui on ühesuguseid andmeid mitmete objektide kohta, siis saab neid andmeid hõlpsasti ette kujutada tabelina. Kujutame ette meie kursuse hinnete tabelit.

Eesnimi	**Perenimi**	**Nädal1**	**Nädal2**	**Nädal3**	**Nädal4**	**Nädal5**	**Nädal6**	**Nädal7**	**Nädal8**
Andres	Paas	A	A	A	A				
Pearu	Murakas	A	A			A			A
Indrek	Paas	A				A			

Selle tabeli alusel saab välja panna näiteks kursuse lõpuarvestused või vaadata, millised nädalad on veel arvestamata. Teatud tasemeni on tabel juba päris sobiv andmete hoidmise viis.

Sageli on aga tahtmine käsitleda andmeid komplekssemalt. Näiteks võime ette kujutada, et õppija kohta on veel mingeid andmeid, mis konkreetse kursusega ei pruugi otseselt seotud olla. Samuti on iga nädala arvestamiseks vaja mingeid ülesandeid lahendada.

Nii saame vastava andmebaasi andmemudeli esitada näiteks järgmiselt. Peame siin silmas, et lahendamine tähendab, et konkreetse nädala kõik vajalikud arvestused on saadud.

- Õppija
  - matrikli_nr : text
  - eesnimi : text
  - perenimi : text
  - isikukood : text
- Nädal
  - nädala_jrk : int
  - temaatika : text
  - kohustuslike_arv : int
  - valik_arv : int
  - test_arv : int
- Lahendamine
  - nädala_jrk : int
  - matrikli_nr : text
  - kuupäev : date

Konkreetse asja või nähtuse esitust mudelis nimetatakse **olemiks** (e objektiks). Näiteks on olem konkreetne *õppija*, konkreetne *nädal*, aga ka konkreetne *lahendamine*. Iga olem on mingit tüüpi ja just olemitüübid on need, mida andmebaasi loomisel kirjeldatakse. Antud juhul on vajalikud olemitüübid *õppija*, *nädal* ja *lahendamine*.

Olemitüübis määratakse erinevad tunnused. Näiteks olemitüübi *õppija* puhul on tekstilist tüüpi tunnused matrikli numbri, eesnime, perenime ja isikukoodi jaoks. Iga konkreetse olemi (antud juhul siis *õppija*) puhul antakse tunnustele väärtused. Oluline on, et tunnuste valik

oleks selline, et iga olem oleks teistest eristatav. Õppija puhul ei pruugi eristamiseks piisata ees- ja perenimest, sest võib olla sama nimega õppijaid. Ühe ülikooli piires on sobiv tunnus matrikli number, mis on unikaalne. Eesti tasemel on üldiselt sobivaks isikukood. Näiteks meie kursustel see aga ei sobi, sest osaleb ka inimesi, kellel isikukoodi ei ole. Isegi mitte mõne teise riigi oma, sest kõigil riikidel isikukoodide süsteemi polegi.

Nende tunnuste komplekti, mille abil saame ühe olemi olemitüübis (ühe kirje tabelis) unikaalselt tuvastada, nimetatakse **primaarseks võtmeks**. Olemitüübis *Õppija* on selleks *matrikli_nr* ja olemitüübis *Nädal* on primaarvõtmeks *nädala_jrk*. Neid võtmeid kasutatakse olemitüübis *Lahendamine* viitamaks nädalale ja lahendajale (õppijale).

Iga olemitüübi jaoks saame oma tabeli, kus võib olla info paljude seda tüüpi olemite kohta.

Õppija

matrikli_nr	eesnimi	perenimi	isikukood
A034	Andres	Paas	34105212737
A037	Pearu	Murakas	34206122154
B124	Indrek	Paas	37111012443
B043	Andres	Paas	36801152732

Nädal

nädala_jrk	temaatika	kohustuslike_arv	valik_arv	test_arv
1	Kahemõõtmeline järjend	1	0	2
2	Kahekordne tsükkel	4	2	1
3	Andmestruktuurid	3	2	1

Lahendamine

Nädal	lahendaja_matrikkel	kuupäev
1	A034	07.04.2017
1	A037	08.04.2017
2	B124	10.04.2017
2	A037	12.04.2017

Kui nüüd nendest tabelitest sobivalt pärida, siis võib näiteks saada tulemuseks tabeli, milles on ühe konkreetse osaleja (Pearu Muraka) kõik arvestatud nädalad koos nädala teemadega ja kuupäevadega.

Kahemõõtmeline järjend 08.04.2017
Kahekordne tsükkel 12.04.2017

Päringukeel

Andmebaaside loomiseks ja andmetega tegutsemiseks on spetsiaalsed andmebaaside juhtsüsteemid. Nimetame siin vaid mõnda: *Oracle Database*, *MySQL*, *Microsoft SQL Server*, *Sybase SQL Anywhere*, *PostgreSQL*.

Neis süsteemides kasutatakse päringukeele SQL (*Structured Query Language*) erinevaid dialekte. Tutvume siin mõne tavalisema käsuga.

Tabeli loomiseks saab kasutada käsku `CREATE TABLE`.

```
CREATE TABLE õppijad (matrikli_nr text, eesnimi text, perenimi text, isikukood text)
```

Kirjeid saab lisada käsu `INSERT` abil.

```
INSERT INTO õppijad VALUES ('A034','Andres','Paas')
```

Kui tahame andmeid näha, siis on abiks käsk `SELECT`.

```
SELECT * FROM õppijad WHERE perenimi = 'Paas'
```

Tore koht, kus ise saab päringuid proovida on sqlzoo.net. Alustame seal päris algusest. Võite lahendada sealtoodud ülesande, aga võime ka proovida kogu tabelit näha

```
SELECT * FROM world
```

või kõigi Euroopa riikide pindalasid

```
SELECT name, area FROM world WHERE continent = 'Europe'
```

Tegelikult on ka Pythoniga võimalik andmebaasiga tegeleda kasutades moodulit sqlite3. Kes tahab, võib katsetada järgmisi programme.

Programm, mis teeb tabeli ja täidab seda. Tuleb märkida, et selle programmi töö ei väljasta ekraanile midagi.

```
import sqlite3
ühendus = sqlite3.connect('kursus.db')
c = ühendus.cursor()

c.execute('''CREATE TABLE õppijad
             (matrikli_nr text, eesnimi text, perenimi text)''')

c.execute("INSERT INTO õppijad VALUES ('A034','Andres','Paas')")
c.execute("INSERT INTO õppijad VALUES ('A037','Pearu','Murakas')")
c.execute("INSERT INTO õppijad VALUES ('B124','Indrek','Paas')")
c.execute("INSERT INTO õppijad VALUES ('B043','Andres','Paas')")

ühendus.commit()

ühendus.close()
```

Programm, mis otsib tabelist neid, kelle perenimi on Paas.

```
import sqlite3
ühendus = sqlite3.connect('kursus.db')
c = ühendus.cursor()
c.execute("SELECT * FROM õppijad WHERE perenimi = 'Paas'")
print(c.fetchall())
ühendus.commit()

ühendus.close()
```

Lõpetuseks

Oleme nüüd vaadanud andmebaasi loomise ja andmetega tegutsemise tehnilisemat poolt. Materjali lõpuosas mõtiskleme natuke üldisemalt, mida see kõik meile annab või anda võiks. Ühelt poolt on andmebaasides olevate andmete põhjal võimalik teha paremaid otsuseid. Tegelikult paljud inimesed erinevates organisatsioonides erinevate andmete analüüsiga tegelevadki. Kui palju neid analüüse otsuste tegemisel arvestatakse, on juba järgmine küsimus.

Siin saabki välja tuua, et normaalse infokasutuse olulised etapid on

- kvaliteetne andmete kogumine ja hoidmine.
- mõtlemispõhine andmeanalüüs ja otsustus.

Andmebaasidest, kus on tohutult andmeid, saab sobivate päringutega filtreerida konkreetses olukorras vajalikke andmeid. Sarnaselt peaksime filtreerima kogu seda infot, mis meile väga erinevaid kanaleid pidi pakutakse ja võib-olla isegi peale surutakse. Info hulgast valiku tegemine ei pruugi aga üldse kerge olla. Nagu ka toitumise puhul, kui laud või kaupluse letid on lookas.

Siinkohal on sobiv soovitada kolleeg Vambola Lepingu osalusel korraldatavat e-kursust “Söömine, joomine ja informatsioon”. Vambola teeb seal väga toreda sissejuhatuse andmete ja informatsiooni teemadel.