Struktuursed andmetüübid

Ühe kaupa väärtuste muutujatesse salvestamine on suuremate andmehulkade puhul praktiliselt võimatu. Eriti tülikas on aga nende järgnev töötlemine. Kui suuremat kogust andmeid on vaja korraga meeles pidada, sobivad selleks nn struktuursed andmetüübid, st ei ole üksikut väärtust, vaid on palju väärtuseid, mis on ühendatud ühtsesse struktuuri ja lisaks võib andmete paiknemine struktuuris anda edasi täiendavat infot (nt nimed tähestiku järjekorras).

Vanim struktuurne andmetüüp on **massiiv** *(array)*, kus hoitakse ühte tüüpi väärtuseid (näiteks ainult täisarve) ja kus igale väärtusele, nn massiivi elemendile „pääseb ligi“ tema **indeksi** kaudu (nö kaugus massiivi algusest ehk *offset*). Loogilises plaanis tuleks ka massiivis hoida ühte tüüpi väärtuseid, näiteks ainult üliõpilaste pikkuseid. Selline kord ja süsteem on vajalik selleks, et andmete töötlemine massiivis oleks võimalikult lihtne. Klassikaliselt hoiti massiive ka mälus jadamisi, mis tegi kiireks ja mugavaks nende andmetele ligipääsu (oli võimalik välja arvutada vajaliku elemendi mälupesa aadress, kui massiivi algus ja andmetüüp ning koos sellega vajatab baitide hulk teada on).

Teine klassikaline, kuid massiivist palju noorem struktuurne andmetüüp on **kirje** *(record)*, mis on ettenähtud erinevat tüüpi andmete koos hoidmiseks. Kirje abil tuleks seostada sellised andmed, mis reaalsuses kuidagi omavahel seotud on, näiteks ühe objekti juurde kuuluvad (inimese nimi, sünniaasta, pikkus, kinga number jms).

Python ei kasuta klassikalist **massiivi** *(array)* mõistet, vaid selle keele puhul räägitakse **jadast** (*sequence*)

Jadasid eristatakse Pythonis kolme tüüpi:

- string
- list
- tuple ehk ennik

Klassikalise massiivina ei käitu neist ükski, indeksite kasutamise võimalus on aga omane kõigile neile.

Jada (sequence)

Jada elemendid on organiseeritud järjestikku ja nendele pääseb ligi indeksi kaudu, mis näitab elemendi kaugust jada algusest ehk elemendi järjekorranumbrit. Kogu selline struktuur on ülesehitatud jadamisi – üks element järgneb teisele. Ja teda salvestatakse samuti jadamisi. Elementide (jada liikmete) nummerdamine algab 0-st ja seega viimase elemendi number on `N-1` (kui jadas on N elementi). Jada pikkuse saab teada `len()`-funktsiooniga: `N = len(jada)`

Järgneb näide indekseerimisest (jada pikkus N)

23	56	875	45	567					
0	1	2	3	4	5	6	...	N-2	N-1

Alternatiivne indekseerimine tehakse negatiivsete indeksitega. Sel juhul on N-elemendilise jada indeksid vahemikus -N kuni -1.

23	56	875	45	567					
-N	-(N-1)	-(N-2)	...	-6	-5	-4	-3	-2	-1

Operaatorid

Sõltuvalt jadasse pandud elementide andmetüüpidest saab nendele rakendada kõiki **tavalisi operaatoreid**, mis sellist tüüpi andmetega opereerida oskavad, st kui jadas on täisarvud, saab nendega arvutada (kasutada aritmeetikaavaldistes), neid võrrelda (kasutada loogikaavaldistes) ja loomulikult sisestada ning välja trükkida. Jada elemente saab mõnel puhul kasutada korraga, mõnel puhul on aga vaja sealt elemente ükshaaval kätte saada.

Spetsiaalselt jada jaoks kehtivad operaatorid on nii jada elementide väljalõikamiseks (nende ükshaaval kasutamiseks) kui ka teheteks terve jadaga.

Lõige

```
jada[i1:i2]   jada[i]
```

Jadast lõigatakse välja elementide alamjada, mis algab `i1` elemendiga (kaasaarvatud) ja lõppeb `i2` elemendiga (väljaarvatud). Inglisekeelne mõiste *slicing*. Kui jätta üks indeksitest ära, siis algusindeksi puudumisel tehakse lõige jada algusest ning lõpuindeksi puudumisel tehakse lõige viimase elemendini.

Sarnaselt saab jadast välja lõigata üksikut elementi. Jadast eraldatakse järjekorranumbriga `i` näidatud element, täpsemalt küll `i-1` esimene element, sest nummerdamine algab 0-ga. Lõike jaoks peab indeks olema täisarvuline ja mahtuma vahemikku `0 <= i <= N-1`, kus N on jada elementide arv.

Lõiked võivad olla ka keerulisemad: **täiendatud lõige** *(extended slicing)* lubab lisaks lõike pikkusele määrata ka sammu (näiteks võtta stringist märke üle ühe, kui sammuks on 2). Samm lisatakse kandilistesse sulgudesse kolmandaks arvuks. Sammu -1 puhul pööratakse jada tagurpidi.

Kordamine

```
jada * täisarvuline_avaldis
```

Jada korratakse nii mitu korda, kui nõuab avaldis (lihtsamal juhul on avaldise kohal täisarv või täisarvuline muutuja). Tehte tulemusena tekib uus jada.

Liitmine ehk konkateneerimine

```
jada + jada
```

Kaks jada liidetakse uueks jadaks, kõigepealt on 1. jada liikmed ja seejärel 2. jada liikmed. Jadad peavad olema sama tüüpi, st stringi ja listi liita ei saa. Tõsisemaks tööks liitmistehet siiski ei soovitata, sest väidetavalt on tegemist aeglase operatsiooniga ja tuleks otsida/kasutada pigem vastavate andmetüüpide meetodeid.

Kuulumine ja mittekuulumine jadasse

```
asi in jada  asi not in jada
```

`in` operaatoriga saab kontrollida kuulumist jadasse. **Stringi** puhul saame teada, kas vastav märk esineb jadas (sest string on jada märkidest). Vastus on tõeväärtustüüpi *true* või *false.* **Listi** ja **enniku** puhul saab teada objekti kuulumise jadasse. Operaator `not in` kontrollib vastupidi „mittekuulumist“ jadasse.

Standardfunktsioonid

Kõigi jada tüüpide jaoks on olemas mitmed **sisefunktsioonid** *(build-in functions)*:

Teisendusfunktsioonide abil saab muuta ühte tüüpi muutujad teist tüüpi muutujateks. Näiteks saab suvalisest arvust teha stringi või stringist listi. Vastavad funktsioonid on `str()`, `list()` ja `tuple()`.

Töötavad ka funktsioonid, nagu

`max(jada)` - tagastab suurima väärtuse

`min(jada)` - tagastab vähima väärtuse

`sorted(jada, reverse = True)` - sorteerib jada, tulemuseks on uus jada (vana säilub); lisades parameetrite loetellu `reverse = True`, sorteeritakse jada tagurpidi.

`sum(jada, init = algväärtus)` – summeerib jadas olevade väärtused, kui seda teha saab, `init` abil saab summa algväärtustada.

Stringid

Stringide eraldamiseks muust programmitekstist kasutatakse jutumärke ja/või ülakomasid. Kasutamisel on nad ühesuguste omadustega. Märgitüüp (üks suvaline sümbol) Pythonis puudub. Seda asendab string pikkusega 1. Stringi saab käsitleda ühe skalaarse väärtusena, kuid samas on ka võimalik vaadelda stringi sümboleid jadana ja nende jaoks kasutada jada operaatoreid.

Pythonis eristati varem **harilikke stringe** *(regular string)* ja **Unicode stringe** *(Unicode string)*. Praeguseks on see eristus kadunud ja kõik stringid on nn Unicode stringid. Andmeid üldisemalt jaotatakse aga tekst (*Text*) ja andmed (*Data*)

Stringist osade väljalõikamine toimub eespool kirjeldatud loogika kohaselt indeksite abil:

Näide:

```
nimi = „Maali“
print(nimi[1]) -> 'a'
print(nimi[2:]) -> 'ali'
```

Stringi „puhastamiseks“ saab talle omistada tühja stringi: `nimi = ''` või kasutada funktsiooni `del(nimi)`

Stringe võrreldakse omavahel toetudes ASCII-tabelile. Võib võrrelda kahte tervet stringi korraga. Ei ole vajadust stringi sümboleid ükshaaval võrdlema hakata:

```
„sääsk“ > „elevant“ -> True
```

Tuleb ka arvestada, et suur- ja väiketähed on erinevad.

Operaatoriga `in` saab küsida märgi kuulumist stringi.

Lisavõimalusi stringide töötlemiseks annab moodul `string`, mida saab programmis kasutusele võtta käsuga `import string`. Mõned näited võimalustest on eeldefineeritud stringid (nn stringkonstandid), mida saab programmides kasutada (näiteks selleks, et kontrollida mõne sümboli kuulumist suurtähtede hulka (tugineb ASCII-tabelile ning seega ei ole lokaliseeritav):

`string.ascii_uppercase` – suurtähed `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`

`string.ascii_lowercase` – väiketähed `'abcdefghijklmnopqrstuvwxyz'`

`string.ascii_letters` – kõik tähed
`'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`

`string.digits` – numbrid `'0123456789'`

Lisaks string-konstantidele on selles moodulis ka vajalikke funktsioone, mida tasuks tervikuna uurida „Reference manualist“. Stringi on juba eelmistes versioonides soovitatud kasutada nö objekt-orienteeritult, st string on klass ja tal on meetodid, mitte funktsioonid. Näiteks:

`asi = asi.upper()` muudab stringis `asi` olevad tähed suurtähtedeks, tekitades samal ajal uue stringi. Näites pannakse uus string nö vanale kohale tagasi..

Näited stringist sümbolite ükshaaval eraldamiseks:

```
# Kasutatakse indeksit, järjest kõik tähed eraldada,
# indeksi piirid on 0 kuni stringi pikkus (viimane väljaarvatud)
s6na = 'abcde'
for i in range(0, len(s6na)):
     print(s6na[i])
```

„Pütoonilisemalt“ sobiks aga kirjutada nii:

```
# Kui indeks ei anna lisaväärtust, ei ole mõtet teda kasutada
s6na = 'abcde'
for t2ht in s6na:
    print(t2ht)
```

Kui string panna kolme paari ülakomade või jutumärkide vahele, võib seal sees näiteks kasutada tavalisi reavahetusi teksti vormindamiseks (ei ole vaja lisada \n või kirjutada korduvaid print-lauseid). Selline võimalus on hea, kui oleks vaja esitada teksti võimalikult WYSIWYG-i kujul – näiteks soovides oma programmiga veebi jaoks HTML-i esitada.

Pythoni stringid on **mittemuudetavud** *(immutable)*. See tähendab, et kui stringi muudetakse (mida saab loomulikult teha), siis tegelikult tekib mällu uus string. Õnneks programmikirjutaja seda reaalselt ei näe, kuid see võib muuta programmi aeglasemaks. Mida aga ei luba interpretaator teha, on ühes stringis tähtede või lõikude asendamist.: omistamine `Nimi[2] = „a“` lõppeb veateatega.

String kui objekt

Python on objekt-orienteeritud keel. Stringi vaadeldakse Pythonis kui objekti. Päris objekt-orienteerituks me oma programme siiski tegema ei hakka. Tehnilist vahet võrreldes mooduli kasutamisega väga palju ei ole. Sisuline vahe (mis nii väga välja ei paista) on suurem.

Objektist võime me mõtelda (esialgu) kui muutujast. Tema jaoks on Pythonis kirjeldatud meetodid (ehk tegevused), mida objekt teha oskab. Meetodi käivitamine toimub liitnime abil: `objekti_nimi.meetodi_nimi`. Kui moodustada string, siis on kohe olemas ka meetodid tema töötlemiseks.

Stringimeetodite nimekirja leiad veebist Pythoni kodulehelt materjalide hulgast:

http://docs.python.org/3.1/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range

Näide:

```
# -*- coding: cp1257 -*-
# Stringi esitäht tehakse suureks ja loendatakse „a" esinemine stringis
nimi =  "maali maasikas"
nimi = nimi.capitalize()   # Esitähe suurendamise meetod
loendur = nimi.count("a")     # Alamstringide loendamise meetod
print("Nimi on ", nimi, "ja selles on ",loendur, " a-d.")
```

Selgitus:

Stringi-tüüpi muutujaks ehk objektiks on `nimi`. Stringi kui objektiga on seotud meetodid `capitalize`, `count` jms. Et nime esitäht suureks teha, käivitatakse meetod nii: `nimi.capitalize()`. Kui meil oleks eraldi string `perekonnanimi`, saaks tema esitähe suureks muuta `perekonnanimi.capitalize`() abil. Muutujanimi liitnime alguses ütleb, millise stringiga tuleb vastav toiming teha.

NB! Võrreldes Python 2.x-ga on keelest kadunud mitmed stringifunktsioonid ja asemele tulnud (või ainsana kasutusse jäänud) stringi kui objekti meetodid. Seega ei tasu taas imestada, kui mõni veebist leitud näide keeldub tööle minemast.

Listid

List on andmeüksuste jada, kus üksikule elemendile pääseb ligi indeksi kaudu. Sarnaselt stringiga näitab indeks elemendi paiknemise kaugust listi alguspunktist. Listi elemendid saavad olla suvalist andmetüüpi ja nad saavad olla erinevad. Listi saab elemente lihtsalt lisada ja sealt neid ka kustutada. Liste saab ühendada ja tükkideks jagada.

Listi loomiseks saab kasutada omistuslauset ja listi määramiseks kandilisi sulge [].

```
asjade_list = ['hobune', 12, 3.45, 'pann', ['list', 'listis']]
```

Näites toodud listi viimane element on list ja tema elementide kätte saamiseks tuleb kasutada kahte indeksit (vt näide allpool). Üksikute elementide ja lõikude eraldamine listist töötab sarnaselt stringile:

```
asjade_list[3] -> 'pann'
asjade_list[4][1] -> 'listis'
```

Listi elemendile omistamine asendab ühe väärtuse teisega:

```
asjade_list[3] = 'kastrul' -> ['hobune',12,3.45,'kastrul',['list',
'listis']]
```

Listist saab kustutada del käsuga indeksi järgi:

```
del asjade_list[2] -> ['hobune',12,'kastrul',['list', 'listis']]
```

Operaatorid

Liste saab võrrelda. Liste võrreldakse element haaval, kuni leitakse erinevus ja selle järgi otsustakse võrdumine, mittevõrdumine jms.

Listist saab eraldada üksikut liiget või lõiku, kusjuures indeksitega manipuleerimine on samasugune kui stringide puhul. Erinevuseks on, et igale indeksile vastab terve element. Üheks listi elemendiks võib olla teine list. Sel juhul tekib kahemõõtmeline struktuur. Listid on **muudetavad** *(mutable)*, st neid saab muuta osade kaupa, näiteks mõnda elementi välja vahetada. See on erinevuseks võrreldes stringiga.

`In`- ja `not in`-operaatorite abil saab kontrollida elemendi kuulumist või mittekuulumist listi (sarnaselt stringile)

```
'pott' in asjade_list -> false
'kastrul' in asjade_list -> true
```

Sisefunktsioonidest *(built-in)* sobib kasutada (ja töötavad sarnaselt nagu stringidegi juures):

`len()` - pikkuse jaoks, st listi elementide arvu leidmiseks;

`max()` ja `min()` - suurima ja vähima väärtusega elemendi leidmiseks; see töötab normaalselt siis, kui listi elemendid on ühte tüüpi

`sorted()` ja `reversed()` - sorteerimine toimub ASCII tabeli järgi, st mitte tähestiku järgi. St suured tähed on kõik väikestest tähtedest eespool.

Listi meetoditega võib tutvuda eespool mainitud Pythoni raamatus. Kasutamispõhimõte on sarnane stringide jaoks kirjeldatuga. Ka siin on tegemist objekti meetodiga ja mitte funktsiooniga. Link Safari kodulehele:

http://proquest.safaribooksonline.com/0132269937/ch06lev1sec14

Tuple ehk ennik

Tuple on väga sarnane listile. Eestikeelse vastega on veidi raskusi. Mõned variandid, mis internetist silma jäid: **rida**, **järjend**, **ennik** (standardile vastav), **korteež** (inglise-eesti sõnastiku tõlge, kasutusel pigem matemaatikas). Püüan teda vastavalt standardile edaspidi ennikuks kutsuda. Programmis kasutatakse enniku kirjeldamiseks ümarsulge (erinevalt listist, kus oli kandilised sulud). Ennikute moodustamine ja temast lõigete tegemine on sarnane listiga:

```
ennik = ('lill', 'liblikas', 13, 'ohakas', 1234, 65.345)
ennik[2] -> 13
```

Ennik on **muudetamatu** *(immutable)*. St tema elemente ükshaaval muuta ei saa. See ongi peamine (ja oluline erinevus võrreldes listiga). Enniku muutmiseks tuleb moodustada uus ennik, näiteks:

```
uus_ennik = (ennik[0], ennik[4], ennik[3]) -> ('lill','ohakas',13)
```

Operaatoreid

Ennikuid saab võrrelda ja nendest lõikeid teha sarnaselt listile. Sisefunktsioonidest töötavad nö info pärimise funktsioonid (`len(), max(), min() ..`), kuid mitte need, mis enniku sisu muuta tahaksid (muudetamatu endmetüüp).

Näiteprogramm:

Jada kõigi elementide töötlemiseks sobib kasutada for-tsüklit. Eespool oli näide stringi kohta. Sarnaselt saab toimida listiga:

```
# Kõigepealt luuakse list sisestatud nimedest (tühi string lõpetab
sisestuse)
# ja seejärel trükitakse see nimekiri välja
# Sisesatmiseks while-tsükkel, väljundiks for-tsükkel

nimed = []                  # Teeme tühja listi
uus_nimi = input("Sisesta nimi ")
while uus_nimi != "":       # Tsükkel töötab kuni sisendiks midagi on
    nimed += [uus_nimi]     # Nimi lisatakse listi
    uus_nimi = input("Sisesta nimi, lõpetamiseks Enter ")
for nimi in nimed:          # Trükitakse välja kõik nimed
    print(nimi)
print("Tänan tähelepanu eest!")
```

Lisaks väljundile võib elementidega teha ka muud.

Kasutamine

Ehkki ühte listi saab koondada erinevat tüüpi andmeid, ei ole selliste listidega eriti palju pihta hakata. Andmete süstemaatiliseks töötlemiseks on eelistatud sarnase struktuuriga andmestikud. Seega tasub oma liste planeerida ja

kasutada suurema hulga sarnaste andmete hoidmiseks. List saab koosneda teistest listidest ja selliselt moodustub kahemõõtmeline struktuur (mida võib endale ka maatriksina ette kujutada).