LUT Python ohjelmointiopas 2023 - osa 10
Sisällysluettelo

Luku 10: Data analytiikka ja uusia tietorakenteita

Tässä oppaassa käsitellään data analytiikan perusteita ohjelmoinnin näkökulmasta, koska se on yksi tyypillinen Pythonin sovellusalue. Kuten olemme nähneet, Pythonilla on helppo lukea tietoja tekstitiedostosta, jakaa ne olion jäsenmuuttujiin ja laittaa oliot sen jälkeen listaan jatkokäsittelyä varten. Usein data kannattaa tallentaa sellaiseen tietorakenteeseen, joka korostaa datan kiinnostavia piirteitä. Esimerkkejä tyypillisistä tietorakenteista on tähän mennessä ollut luokka ja lista, joiden lisäksi tässä luvussa tutustumme sanakirjaan ja matriisiin. Listan idea oli laittaa tietoalkioita peräkkäin samaan paikkaan, jossa niitä voi siirtää helposti. Sanakirja muistuttaa listaa, mutta sanakirjassa on tietoalkion lisäksi jokaisella alkiolla hakuavain, jolloin tietoon pääsee käsiksi nopeasti. Matriisi perustuu myös lista-ideaan, mutta nyt listoja voi olla useita, eli listan sisällä on listoja, jotta tiedot voidaan tallettaa 2-ulotteiseen tietorakenteeseen taulukkolaskennan tyylisesti ja tietoalkioin paikka matriisissa määräytyy rivi- ja sarake-tietojen perusteella. Rajoitumme tässä oppaassa 2-ulotteiseen tietorakenteeseen eli matriisiin, mutta käytännössä ulottuvuuksia voi olla enemmänkin. Listassa alkio löytyy yhdellä indeksillä (1-ulotteinen tietorakenne), matriisissa alkio löytyy kahdella indeksillä (rivi ja sarake), 3-ulotteisessa taulukossa alkio löytyy 3 indeksillä (esim. 3-ulotteinen koordinaatisto ja x, y sekä z koordinaatit) jne. Uusien tietorakenteiden lisäksi tutustumme tässä luvussa tarkemmin lajittelun, joka onnistuu myös listaa monipuolisemmille tietorakenteille, mutta on myös vastaavasti työläämpää.

Data-analyysi on helppo aloittaa taulukkolaskentaohjelmalla kuten Excelillä, mutta tehokkaiden analyysiominaisuuksien toisena puolena Excelillä on omat rajoitteensa. Yksi näistä on Excelin kyky käsitellä korkeintaan 1 048 576 riviä sisältäviä tiedostoja. Vaikka tämä onkin usein riittävästi, automaattinen datan keruu tuottaa helpolla paljon tätä suurempia datamääriä. Esimerkiksi vuodessa on noin 364 päivää, 8736 tuntia, 524 160 minuuttia ja 31 449 600 sekuntia, joten minuuttitasolla kirjatut tiedot tuottavat parissa vuodessa niin paljon dataa, ettei Excel pysty käsittelemään niitä. Toinen esimerkki datan määrästä on New Yorkin Yellow Cab eli ”keltainen taksi”, joka julkaisee taksien laskutustietoja kuukausittain noin 600 MB tiedostoina, joissa on siis miljoonia tietorivejä. Suomessa Traficom julkaisee liikennekäytössä olevista ajoneuvoista dataa tekstitiedostoina, jossa oli 31.3.2019 yhteensä 5 054 746 riviä. Python sopii tällaisten tiedostojen käsittelyyn hyvin, sillä se pystyy lukemaan tiedostoja varsin pienellä ohjelmointityöllä, valitsemaan kiinnostavat tietoalkiot sekä suorittamaan niille perusanalyysin.

Pythoniin on saatavilla monipuolisia analytiikka-kirjastoja, joissa on paljon valmiita rutiineja data-analyysien tekemiseen ja visualisointiin. Yksi tunnetuimmista tällaisista kirjastoista on Pandas, joka hyödyntää tehokkaasti aiemmin puhuttuja kirjastoja kuten numpy ja matplotlib. Data analytiikasta laajemmin kiinnostuneet voivat siis jatkaa asiaan tutustumista näiden työkalujen avulla.

Listan kertaus

Listan perusoperaatiot näkyvät kertauksen vuoksi esimerkissä 10.1, sillä sanakirja muistuttaa monelta osin listaa. Esimerkissä määritellään ensin lista-tyyppinen muuttuja ja sen jälkeen listaan lisätään 4 alkiota `append()`-jäsenfunktiolla. Listan alkiot voidaan käsitellä kahdella erilaisella toistorakenteella, joista `for Alkio in Lista` -ratkaisu on looginen ja selkeä tapa esim. alkioiden tulostamiseen. Toinen vaihtoehto on selvittää ensin listan pituus `len()`-funktiolla ja sen jälkeen käydä listan alkiot läpi indekseihin perustuen. Vaikka tämä tapa on hieman edellistä työläämpi, saamme tällöin selvitettyä tietoalkion arvon lisäksi sen paikan listassa eli sen indeksin. Ja indeksi taas mahdollistaa esim. alkion poistaminen ja muita alkioon perustuvia operaatiota eli jossain tapauksissa tämä on edellistä tapaa parempi ratkaisu. Listan käytön lopuksi se tulee tyhjentää `clear()`-jäsenfunktiolla.

Esimerkki 10.1. Listan luominen, alkioiden lisäys, käyttö ja tyhjennys

```
import random
# Listan luominen ja alkioiden lisäys append-jäsenfunktiolla
Lista = []
for i in range(4):
    Lista.append(random.randint(0,100))

print("Listan alkioiden läpikäynti for Alkio in -rakenteella:")
for Alkio in Lista:
    print(Alkio, end=" ")
print("\n")

print("Listan alkioiden läpikäynti indekseillä:")
print("Indeksi: Listan alkion arvo")
for i in range(len(Lista)):
    print("{}: {}".format(i, Lista[i]))
Lista.clear()
```

Tuloste

```
Listan alkioiden läpikäynti for Alkio in -rakenteella:
16 5 24 26

Listan alkioiden läpikäynti indekseillä:
Indeksi: Listan alkion arvo
0: 16
1: 5
2: 24
3: 26
```

Sanakirja (eng. Dictionary)

Sanakirja poikkeaa meille tutusta lista-sarjamuuttujista toimintalogiikan osalta, sillä se muistuttaa enemmän puhelinluetteloa, tai yllättäen sanakirjaa, kuin kauppalistaa. Sanakirja tunnetaan myös monella muulla nimellä kuten hakurakenne, luettelo ja assosiaatiotaulu (eng. associative array). Sanakirja sisältää kaksi erilaista tietoalkiota, jotka ovat avain (key) ja arvo (value). Sanakirjan avaimen tulee olla ainutlaatuinen, eli sanakirja ei voi sisältää kahta samaa avainta. Toinen avaimiin liittyvä rajoite on, että avaimeksi käy ainoastaan vakioarvoiset tietotyypit kuten merkkijonot tai luvut. Käytännössä tämä tarkoittaa sitä, että avaimina voidaan käyttää ainoastaan yksinkertaisia tietotyyppejä.

Esimerkissä 10.3 näkyy sanakirjan perusoperaatiot eli luominen, alkioiden lisäys sekä käyttö ja lopuksi sanakirjan tyhjennys. Sanakirja määritellään aaltosuluilla `{}`, kun lista määritellään hakasuluilla `[]` (ja tuple kaarisuluilla `()`). Sanakirjan luomisen jälkeen siihen voidaan lisätä alkioita, mutta listan `append()`-jäsenfunktion sijasta sanakirjassa sijoitetaan annetulle avaimelle arvoksi haluttu tieto tyyliin ’`Sanakirja[Avain] = random.randint(0,100)`’. Kun sanakirjassa on avaimia ja arvoja, voidaan sanakirjan avaimet käydä läpi `for Avain in Sanakirja` -rakenteella. Huomaa, että sanakirjan tapauksessa on oleellista huomioida sekä avain että sen arvo, kuten esimerkin tulosteessa näkyy. Käytön jälkeen sanakirja tyhjennetään tuttuun tapaan `clear()`-jäsenfunktiolla.

Esimerkki 10.2. Sanakirjan luominen, alkioiden lisäys, käyttö ja tyhjennys

```
import random
# Sanakirjan luominen ja alkioiden lisäys avain-arvo -pareina
Sanakirja = {}
for Avain in range(0, 4):
    Sanakirja[Avain] = random.randint(0,100)
print("Sanakirjan alkioiden läpikäynti avaimilla")
print("Avain: Sanakirjan alkion arvo")
for Avain in Sanakirja:
    print("{}: {}".format(Avain, Sanakirja[Avain]))
Sanakirja.clear()
```

Tuloste

```
Sanakirjan alkioiden läpikäynti avaimilla
Avain: Sanakirjan alkion arvo
0: 74
1: 37
2: 66
3: 84
```

Toisessa sanakirja esimerkissä 10.3 näkyy muutama muu sanakirjan tyypillinen käyttötapaus. Nyt sanakirjan määrittelyn yhteydessä siihen sisällytetään 3 avain-arvo -paria. Tämän sanakirjan avaimena toimii merkkijonona annettu opiskelijanumero ja tietoalkioina ovat opiskelijoiden nimet. Koska sanakirjassa on avain-arvo -parit, on nimet annettu yhtenä merkkijonona yksinkertaisuuden vuoksi. Huomaa, että avaimen ja arvon välissä on kaksoispiste (:) ja näitä pareja erottaa aina pilkku. Sanakirjan alustuksen jälkeen siihen lisätään vielä kaksi uutta alkiota sijoittamalla eli sanakirja on dynaaminen rakenne, johon voidaan lisätä alkioita sekä poistaa niitä. Sanakirja käydään seuraavaksi läpi edellä olleella `for Avain in` -rakenteella tulostaen avain ja sitä vastaava arvo. Seuraavaksi esimerkissä tarkastetaan, onko haluttu avain sanakirjassa `in`-operaattorilla ja jos on, niin tulostetaan taas avain ja arvo. Seuraavaksi tämä avain ja siihen liittyvä arvo poistetaan sanakirjasta `del`-käskyllä, kerrotaan asiasta käyttäjälle ja tulostetaan sanakirjassa oleva opiskelijoiden määrä `len()`-funktion avulla. Sanakirja on listan tavoin sarjallinen muuttuja, joten niitä käyttäessä voi usein käyttää samoja operaatioita, vaikka tietysti online-dokumentaatiosta operaatiot ja niiden toiminta on helppoa tarkastaa. Käytön loputtua sanakirja tulee tyhjentää.

Esimerkki 10.3. Sanakirjan tyypillinen avain-alkio -rakenne

```
# Sanakirjan alustus eli alkioiden lisäys määrittelyvaiheessa
Opiskelijat = {"0123456" : "Brian Kottarainen",
               "0654321" : "Liisa Opiskelija",
               "0666111" : "Ella Esimerkki"}

# Alkioiden lisäys olemassa olevaan sanakirjaan
Opiskelijat["0567432"] = "Olli Opiskelija"
Opiskelijat["0987654"] = "Iiro Ikiteekkari"

print("Sanakirjan alkioiden läpikäynti avaimilla:")
for Avain in Opiskelijat:
    print("{}: {}".format(Avain, Opiskelijat[Avain]))
print()

Avain = "0654321"
if (Avain in Opiskelijat):
    print("Opiskelijanumero '{}' liittyy nimeen '{}'.".
              format(Avain, Opiskelijat[Avain]))
    del Opiskelijat[Avain]
    print("Opiskelija poistettu sanakirjasta.")
    print("Opiskelijoita on vielä {}.".format(len(Opiskelijat)))
Opiskelijat.clear()
```

Tuloste

```
Sanakirjan alkioiden läpikäynti avaimilla:
0123456: Brian Kottarainen
0654321: Liisa Opiskelija
0666111: Ella Esimerkki
0567432: Olli Opiskelija
0987654: Iiro Ikiteekkari

Opiskelijanumero '0654321' liittyy nimeen 'Liisa Opiskelija'.
Opiskelija poistettu sanakirjasta.
Opiskelijoita on vielä 4.
```

Sanakirjaa käyttäessä tulee muistaa seuraavat asiat:

- Sanakirjassa on avain-arvo -pareja, joista avaimen tulee olla edellä kerrotun mukaisesti yksinkertainen tietotyyppi eli luku tai merkkijono. Tietoalkio voi olla myös rakenteinen tietorakenne, vaikka tässä oppaassa rajoitumme yksinkertaisiin tietotyyppeihin myös arvon osalta.
- Avaimet eivät ole järjestyksessä sanakirjan sisällä. Mikäli järjestyksellä on väliä, on käyttäjän huolehdittava siitä itse arvojen lisäysvaiheessa. Toinen vaihtoehto on käyttää collections-kirjastosta löytyvää sanakirjan kehittyneempää muotoa, OrderedDict, joka säilyttää siihen laitettujen alkioiden järjestyksen. Kolmas vaihtoehto on lajitella sanakirja itse, johon tutustumme seuraavassa kohdassa.

Lajittelu

Käsiteltäessä isoja tietomääriä tulee usein tarve järjestää tietoa siten, että se tuo esille jonkun tiedon ominaispiirteen. Esimerkiksi numerot voivat kasvaa, tai pienentyä, systemaattisesti ja tämä on helppo huomata yksinkertaisesti lajittelemalla numerot sopivalla tavalla. Lajittelu on yksi ohjelmoinnin perinteisiä kiinnostuksen kohteita ja kuten Tietorakenteet ja Algoritmit -kurssilla nähdään, on lajitteluun kehitetty useita erilaisia ratkaisuja, jotka poikkeavat toisistaan merkittävästi erityisesti nopeuden ja muistintarpeen osalta. Tässä oppaassa emme paneudu asiaan näin tarkasti, mutta katsomme muutamaa keskeistä asiaa tietojen lajitteluun liittyen erilaisten tietorakenteiden yhteydessä. Nämä asiat eivät ole tämän oppaan ydinsisältöä vaan demonstroivat asiasta kiinnostuneille, miten Pythonilla voi tehdä erilaisia asioita tutustumalla tarkemmin erilaisiin rakenteisiin ja ratkaisuihin.

Lista on Pythonin perustietorakenne käsiteltäessä useita tietoalkioita ja tämä näkyy mm. siinä, että listan lajittelu on tehty helpoksi esimerkin 10.4 mukaisesti. Listan ja sen alkioiden määrittelyn jälkeen lista voidaan lajitella `sort()`-jäsenfunktiolla alla olevan esimerkin mukaisesti ja tiivistäen kaikki toimii odotetulla tavalla niin kuin aiemminkin. Huomaa, että tällä kertaa listat tulostetaan Pythonin esitysmuodossa, sillä tässä yhteydessä tulostuvat sulut jne. tuovat meille lisäarvoa, kun yritämme ymmärtää lajittelun periaatteita.

Esimerkin 10.4 toinen tietorakenne on lista, joka sisältää listoja. Aiemmin puhutun mukaisesti tämä ei ole tämän oppaan normaali tietorakenne, mutta tässä tapauksessa se sopii esittämään useita tietoalkioita sisältäviä rakenteita ja niiden lajittelun periaatteet. Käytännössä `ListojaListassa`-rakenteen ytimenä on listan alkioina olevat opiskelijatiedot eli opiskelijanumero, etunimi ja sukunimen ensimmäinen kirjain. Jokaisesta opiskelijasta on nämä samat tiedot omassa listassa, ja tiedot ovat aina samassa järjestyksessä esimerkin mukaisesti. `ListojaListassa`-rakenteessa on kolmen opiskelijan tiedot eli se sisältää 3 samanlaista listaa. Lajittelemme tämän rakenteen kahdella eri tavalla ja siksi teemme listasta ensin kopion.

Kuten muuttujaparametrien yhteydessä oli puhetta, rakenteisten tietorakenteiden toiminta poikkeaa yksinkertaisista tietorakenteista. Esimerkissä 10.4 tämä näkyy siinä, että

tietorakenteen kopio pitää tehdä `copy()`-jäsenfunktiolla. Tämä kopioinnin jälkeen meillä on kaksi erillistä listaa, eikä yhteen listaan tehdyt muutokset näy toisessa listassa. Esimerkin 10.4 alussa tehty merkkijonoja sisältävän listan lajittelu on selkeää eli tietoalkiot laitetaan suuruusjärjestykseen. `ListaListoja`-rakenteen kohdalla listan tietoalkiot sisältävät 3 erillistä tietoalkiota, joten herää kysymys, minkä tietoalkion mukaan tiedot lajitellaan? Ja esimerkin mukaisesti `sort()`-jäsenfunktio lajittelee ensimmäisen alkion perusteella nousevaan järjestykseen. Tässä vaiheessa on siis selvinnyt, että `ListaListoja`-rakenne on rakenteinen tietorakenne, jonka vuoksi sen kopiointi on tehtävä `copy()`-jäsenfunktiolla ja lajittelu `sort()`-jäsenfunktiolla perustuu listojen ensimmäisten tietoalkioiden järjestykseen.

Esimerkki 10.4 jatkuu tulostamalla alkuperäinen `ListaListoja` ja sen jälkeensama rakenne `sorted()`-funktiokutsun jälkeen. Tulosteista näkyy, että rakenne on ensimmäisessä tulosteessa alkuperäisessä järjestyksessä ja toisessa se on lajiteltu samalla tavalla kuin aiemmin `sort()`-jäsenfunktiolla. `sorted()`-funktion osalta oleellinen osa tulee seuraavaksi, kun funktiokutsun toisena parametrina on `key`-parametri, jossa määritellään lajitteluperusteeksi `ListaListoja`-tietorakenteen tieto indeksillä 1, 2 ja 0. Tässä yhteydessä joudumme palaamaan opiskelijan tietoihin, jotka olivat esim. `["0123456","Brian","K"]` eli indeksillä 0 on opiskelijanumero, 1 on etunimi ja 2 on sukunimen ensimmäinen kirjain. Tutkimalla tulostetta tarkemmin voimme todeta, että `sorted()`-funktio lajitellut on `ListaListoja`-listan tiedot näiden mukaiseen järjestykseen.

Esimerkki 10.4. Listan lajittelu

```
print("Lista ja sen lajittelu:")
Lista = ["Liisa", "Ella", "Brian"]
print(Lista)
Lista.sort()
print(Lista)
print()

print("Tietorakenteena lista, jossa on listoja:")
ListaListoja = [["0123456","Brian","K"],
                ["0654321","Liisa","O"],
                ["0066011","Ella","E"]]

print("Lista listoja -rakenteen kopion lajittelu .sort()-jäsenfunkiolla:")
Kopio = ListaListoja.copy()
print(Kopio)
Kopio.sort()
print(Kopio)
print()

print("Lista listoja -rakenteen lajittelu sorted()-funkiolla:")
print(ListaListoja)
print(sorted(ListaListoja))
print(sorted(ListaListoja, key=lambda ListaListoja:ListaListoja[1]))
print(sorted(ListaListoja, key=lambda ListaListoja:ListaListoja[2], reverse=True))
print(sorted(ListaListoja, key=lambda ListaListoja:ListaListoja[0]))
```

Tuloste

```
Lista ja sen lajittelu:
['Liisa', 'Ella', 'Brian']
['Brian', 'Ella', 'Liisa']

Tietorakenteena lista, jossa on listoja:
Lista listoja -rakenteen kopion lajittelu .sort()-jäsenfunkiolla:
[['0123456', 'Brian', 'K'], ['0654321', 'Liisa', 'O'], ['0066011', 'Ella', 'E']]
[['0066011', 'Ella', 'E'], ['0123456', 'Brian', 'K'], ['0654321', 'Liisa', 'O']]

Lista listoja -rakenteen lajittelu sorted()-funkiolla:
[['0123456', 'Brian', 'K'], ['0654321', 'Liisa', 'O'], ['0066011', 'Ella', 'E']]
[['0066011', 'Ella', 'E'], ['0123456', 'Brian', 'K'], ['0654321', 'Liisa', 'O']]
[['0123456', 'Brian', 'K'], ['0066011', 'Ella', 'E'], ['0654321', 'Liisa', 'O']]
[['0654321', 'Liisa', 'O'], ['0123456', 'Brian', 'K'], ['0066011', 'Ella', 'E']]
[['0066011', 'Ella', 'E'], ['0123456', 'Brian', 'K'], ['0654321', 'Liisa', 'O']]
```

Edellä lajittelimme listoja sisältävän listan ja lähdimme siitä, että samoja periaatteita voi soveltaa myös sanakirjan lajitteluun. Esimerkissä 10.5 lähtee liikkeelle sanakirjan määrittelystä ja se tulostetaan ensin sellaisenaan sekä `sorted()`-funktiokutsun jälkeen, jolloin huomaamme, että `sorted()`-funktiokutsun jälkeen tulostuu avaimet lajiteltuna ja tietorakenteena onkin lista. Käytännössä toiminta vastaa aiempaa `for Avain in` -rakenteen toimintaa, sillä sekin käy läpi sanakirjan avaimet ja arvojen haku tehdään erikseen `Sanakirja[Avain]` -rakenteella.

Sanakirja voidaan lajitella `sorted()`-funktiolla usealla eri tavalla. Lähtökohtana on, että sanakirjassa on 2 tietoalkiota eli avain ja arvo, ja sanakirja voidaan lajitella näiden tietojen mukaan nousevaan tai laskevaan järjestykseen. Lajittelu tällä tavoin edellyttää, että sanakirja jaetaan tuple-pareihin `items()`-jäsenfunktiolla, määritellään lajitteluavain `key`-parametrilla ja lopuksi muutetaan `reverse`-parametrin arvo `True`:ksi, jos haluamme lajitella tiedot käänteiseen järjestykseen. Esimerkin 10.5 tulosteissa näkyy, miten sanakirja on hävinnyt eli tulosteissa ei ole enää aaltosulkuja vaan `sorted()`-funktio on palauttanut listan, jossa on tällä kertaa tuple-pareja halutussa järjestyksessä.

Kuten edellä mainittiin, nämä asiat eivät ole oppaan ydinsisältöä vaan asiasta kiinnostuneille esimerkkejä siitä, miten tietoa voidaan muokata tarpeen mukaiseen muotoon ja tehdä erilaisia asioita.

Esimerkki 10.5. Sanakirjan lajittelu

```
Sanakirja = {"0123456" : "Brian", "0654321" : "Liisa", "0066011" : "Ella"}

print(Sanakirja)
print(sorted(Sanakirja))
print(sorted(Sanakirja.items(), key=lambda Sanakirja:Sanakirja[0]))
print(sorted(Sanakirja.items(), key=lambda Sanakirja:Sanakirja[1]))
print(sorted(Sanakirja.items(), key=lambda Sanakirja:Sanakirja[1], reverse=True))
```

Tuloste

```
{'0123456': 'Brian', '0654321': 'Liisa', '0066011': 'Ella'}
['0066011', '0123456', '0654321']
[('0066011', 'Ella'), ('0123456', 'Brian'), ('0654321', 'Liisa')]
[('0123456', 'Brian'), ('0066011', 'Ella'), ('0654321', 'Liisa')]
[('0654321', 'Liisa'), ('0066011', 'Ella'), ('0123456', 'Brian')]
```

Muutama huomio lajitteluun liittyen:

- `sorted()` ei ole jäsenfunktio eikä muuta tietorakennetta vaan palauttaa muutetun kopion paluuarvona.

- `lambda` on Pythonin tapa ilmaista *anonyymi funktio*, eikä tästä tarvitse tietää tämän enempää, vaan asiaan palataan myöhemmillä ohjelmointikursseilla. Riittää, että hahmotat sanakirjan lajittelun tietyn kentän mukaan, jos haluat sitä tehdä.
- Lajittelusta kiinnostuneet voivat tutustua näihin asioihin tarkemmin esimerkiksi Python dokumentaation ”Sorting HOW TO” –osuuden avulla. Lähtökohtaisesti tämän oppaan taso on listojen lajittelu `sort()`-jäsenfunktion avulla.

Huomioita sarjallisten muuttujien vertailusta

Pythonin omien sarjallisten muuttujien (lista, tuple, sanakirja; eng. sequence) vertailussa on joitakin erityispiirteitä, jotka vaikuttavat niiden käyttäytymiseen. Yhtäsuuruutta testattaessa tulkki vertaa keskenään ensin ensimmäisiä tietueita, sitten toisia, sitten kolmansia jne., kunnes päädytään sarjan viimeiseen tietueeseen. Tässä vaiheessa astuu kehään joukko erikoissääntöjä, jotka voivat aiheuttaa ongelmia vertailuja suoritettaessa.

Yksinkertaisimmassa tapauksessa suurempi sarja on se, jolla on arvoltaan suurempi alkio lähempänä alkua. Jos kuitenkin sarjojen kaikki alkiot ovat samanarvoisia, on se sarja, jolla niitä on enemmän, suurempi. Jos taas sarjat ovat samanpituisia sekä alkioiltaan identtisiä, ovat ne samanarvoisia, mutta vain jos alkioiden järjestys on sama.

Alkioita keskenään verrattaessa se alkio, jonka ensimmäinen eroava merkki on merkistötaulukon arvoltaan suurempi, on myös lopullisesti suurempi, vaikka alkio olisi lyhempi. Tämän takia esimerkiksi isot kirjaimet ovat aina pieniä kirjaimia arvollisesti pienempiä. Alla olevasta taulukosta näet joitain esimerkkejä näiden sääntöjen käytännön tulkinnoista; kaikki vertailut tuottavat tuloksen `True`:

```
(1, 2, 3)              < (1, 2, 4)
(1, 2, 3)              < (1, 3, 2)
[1, 2, 3]              < [1, 2, 4]
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Eri tietotyyppien vertailu keskenään ei ole sallittua. Listaa ei siis voida verrata kokonaislukuun, eikä merkkijonoa desimaalilukuun. Tämä ei kuitenkaan päde numeroarvoihin: kokonaisluku (integer) 3 on samanarvoinen kuin desimaaliluku (float) 3.00, eli numeroita verrataan keskenään nimenomaisesti numeroarvoina tallennustyypistä huolimatta. Huomaa kuitenkin, että kokonaisluku ja desimaalilukuna ovat yhtä suuria vain, kun liukuluvun desimaaliosa on tasan 0. Kuten olet varmaan huomannut, käytännössä lasketun desimaaliluvun arvossa on usein pientä epätarkkuutta ja palaamme tähän tarkemmin pyöristyksen yhteydessä myöhemmin. Tiivistetysti lasketuille desimaaliluvuille ei kannata tehdä yhtäsuuruusvertailu.

Tyypillisiä sanakirjan lajittelutarpeita

Esimerkissä 10.6 näkyy tyypillisiä lajittelutehtäviä sanakirjassa olevalle datalle. Tässä esimerkissä autoja on luokiteltu vuosiluvun mukaan ja saatu sanakirjassa näkyvät määrät eri vuosien autoja. Kuten edellä oli puhetta, avaimen eli vuosiluvun mukaan tiedot on helppo lajitella `sorted()`-funktiolla, mutta datan eli lukumäärän perusteella lajittelu edellyttää sanakirjan rakenteen muutosta `items()`-jäsenfunktiolla ja lajitteluavaimen määrittelyä `key`-parametrilla. Tässä oppaassa näitä asioita ei tarvitse opetella, vaan jos niitä tarvitsee, ne voi katsoa esimerkeistä.

Esimerkki 10.6. Sanakirjan lajittelu avaimen ja tiedon perusteella

```
Autot = {2019:23, 2017:31, 2016:42, 2018:38, 2015:29}

print(Autot)
print("Järjestettynä avaimen eli vuosiluvun mukaan nousevasti:")
Lajiteltu = sorted(Autot)
print("Vuosi: Autoja")
for Vuosi in Lajiteltu:
    print("{}: {}".format(Vuosi, Autot[Vuosi]))
print()

print("Järjestettynä lukumäärän mukaan laskevasti:")
Lajiteltu = sorted(Autot.items(), key=lambda Autot:Autot[1], reverse=True)
print("Vuosi: Autoja")
for Alkio in Lajiteltu:
    print("{}: {}".format(Alkio[0], Alkio[1]))
```

Tuloste

```
{2019: 23, 2017: 31, 2016: 42, 2018: 38, 2015: 29}
Järjestettynä avaimen eli vuosiluvun mukaan nousevasti:
Vuosi: Autoja
2015: 29
2016: 42
2017: 31
2018: 38
2019: 23

Järjestettynä lukumäärän mukaan laskevasti:
Vuosi: Autoja
2016: 42
2018: 38
2017: 31
2015: 29
2019: 23
```

Matriisi eli numpy-moduulin taulukko

numpy-moduuli on numeerisen laskennan laajennus Pythoniin eli Numerical Python – paketti, joka tarjoaa työkaluja isojen moniulotteisten taulukoiden ja matriisien käsittelyyn. Tässä oppaassa katsomme vain 2-ulotteista taulukkoa, matriisia, joka on tyypillinen tapa esittää dataa taulukkolaskentaohjelmien riveinä ja sarakkeina. Pythonissa ei ole omaa matriisi-tietotyyppiä, mutta vastaavan rakenteen pystyy tekemään lisäämällä listoja listaan. Tämän rakenteen riski on kuitenkin se, että listat ovat dynaamisia rakenteita ja matriisi käsitellään lähtökohtaisesti staattisena rakenteena, jossa on vakiomäärä rivejä ja sarakkeita. Listat tekevät helpoksi tämän oletuksen rikkomisen, joka johtaa tyypillisesti myös ohjelman rikkoutumiseen. Siksi matriiseja käsiteltäessä kannattaa käyttää matriisi-rakenteita seuraavan esimerkin mukaisesti. numpy-moduulin asennus käytiin läpi luvussa 8 ja seuraavien ohjelmien suorittaminen edellyttää sitä.

Esimerkin 10.7 alussa on kirjaston tuonti ohjelmaan `import`-käskyllä ja sen jälkeen on määritelty matriisin rivien ja sarakkeiden määrät kiintoarvoina. Matriisin voi luoda kirjoittamalla sen alkiot koodiin (`MatriisiA`) tai käyttämällä numpyn `zeros()`-funktiota, joka alustaa matriisin nollilla (`MatriisiB`). Matriisin alkiot voi käydä läpi kahdella silmukalla ja esimerkissä jokaiseen matriisin alkioon kirjoitetaan uusi arvo eli rivi- ja sarakeindeksien summa.

numpy tarjoaa valmiita työkaluja matriisien käsittelyyn, joista tässä esimerkissä on käytetty yhteenlaskua ja tulostusta. Lopussa näkyvä `delete()`-funktio mahdollistaa matriisin alkioiden, rivien ja sarakkeiden poistamisen ja siksi sillä tulee vapauttaa matriiseille varattu

muisti. Esimerkissä matriisi tulostetaan yhdellä `print`-käskyllä numpyn toteutuksen mukaisesti ja lisäksi matriisin tiedot tulostetaan puolipisteillä eroteltuna, jolloin data on helppo kopioida esim. Exceliin visualisointia varten.

Esimerkki 10.7. numpy-kirjaston matriisin peruskäyttö

```
import numpy
RIVEJA = 3
SARAKKEITA = 3

# Matriisin luominen, 2 tapaa luoda ja alustaa, molemmissa kokonaislukuja
MatriisiA = numpy.array( # alustus halutuilla arvoilla
    [[1,2,4],
     [5,6,7],
     [8,9,10]])

MatriisiB = numpy.zeros((RIVEJA, SARAKKEITA), int) # alustus nolliksi

# Matriisin alkioiden käsittely indekseillä alustamisen jälkeen
for Rivi in range(RIVEJA):
    for Sarake in range(SARAKKEITA):
        MatriisiB[Rivi][Sarake] = Rivi+Sarake

# Laskentaa matriiseilla numpyn avulla
MatriisiC = MatriisiA + MatriisiB

# Matriisin tulostaminen indekseillä, puolipiste Excel-siirtoa varten
for Rivi in range(RIVEJA):
    for Sarake in range(SARAKKEITA):
        print(MatriisiB[Rivi][Sarake], end=';')
    print()
print()

# Matriisin tulostaminen numpyn avulla
print(MatriisiC)
# Matriisin tuhoaminen, rivin/sarakkeen/alkion poistaminen mahdollista
MatriisiA = numpy.delete(MatriisiA, numpy.s_[:], None)
MatriisiB = numpy.delete(MatriisiB, numpy.s_[:], None)
MatriisiC = numpy.delete(MatriisiC, numpy.s_[:], None)
```

Tuloste

```
0;1;2;
1;2;3;
2;3;4;

[[ 1  3  6]
 [ 6  8 10]
 [10 12 14]]
```

Yhteenveto

Osaamistavoitteet

Tämän luvun keskeiset asiat on nimetty alla olevassa listassa. Tarkista sen avulla, että tunnistat nämä asiat ja sinulla on käsitys siitä, mitä ne tarkoittavat. Mikäli joku käsite tuntuu oudolta, käy siihen liittyvät asiat uudestaan läpi. Nämä käsitteet ja asiat on hyvä muistaa ja tunnistaa, sillä seuraavaksi teemme niihin perustuvia ohjelmia.

- Tietorakenteita
    - Listan kertaus (E10.1)
    - Sanakirja, Dictionary (E10.2, 10.3)
    - Matriisi, numpy-moduulin moniulotteinen taulukko
- Lajittelu
    - Listan lajittelu, sort()-jäsenfunktio (E10.4)
    - Rakenteisen tietorakenteen lajittelu, listassa listoja (E10.4)
    - Sanakirjan lajittelu (E10.5)
    - Sarjallisten muuttujien vertailun periaatteita
    - Tyypillisiä sanakirjan lajittelutarpeita (E10.6)
- numpy-matriisin käyttö (E10.7, 10.8)
- Yhteenveto, kokoava esimerkki (E10.8)

Pienen Python-perusohjelman tyyliohjeet

Tähän lukuun liittyvät tyyliohjeet on koottu alle. Alla olevia ohjeita noudattamalla vältät yleisimmät virhetilanteisiin ja poikkeuksiin liittyvät ongelmat, ja perustellusta syystä niistä voi ja tulee poiketa.

1. Rakenteiset tietorakenteet lista, matriisi ja sanakirja tulee tyhjentää sen pää/aliohjelman lopussa, jossa ne luodaan
2. Dynaamisia rakenteita, lista tai sanakirja, tulee käyttää, jos samanlaisia muuttujia on 5 tai enemmän
3. Dynaamisissa rakenteissa olevien tietoalkioiden tulee olla samaa tyyppiä
4. Sanakirja on perus- ja tavoitetason rakenne, jota kannattaa käyttää tietyissä tehtävissä
5. `numpy`-matriisin käsittelyssä tulee käyttää vakioita
6. Listan ja sanakirjan lajitteluun voi käyttää `sorted()`-funktioita ja `lambda`:a
7. `lambda`:a käytetään vain `sorted()`-funktion yhteydessä

Luvun asiat kokoava esimerkki

Kokoavassa esimerkissä 10.8 käsitellään samaa askeldatatiedostoa kuin esimerkissä 8.18. Tällä kertaa tiedostoa luettaessa ohjelmassa on poikkeustenkäsittely ja tiedoston tiedoista tehdään oliolista, jossa on tehtävän kannalta oleelliset tiedot eli päivämäärä ja siihen liittyvä askelmäärä vastaavasti aika-tietona ja kokonaislukuna. Analyysi käy oliolistan läpi ja lisää askelmäärän matriisiin päivämäärän mukaisen kuukauden ja viikonpäivän kohdalle. Näin tulostusvaiheessa saamme taulukon, jossa on jokaista vuoden kuukautta kohden yksi rivi, jolla on aina jokaista viikonpäivää kohden yksi sarake. Nämä tiedot voi siirtää Exceliin visualisointia varten alla olevan kuvan 10.1 mukaisesti.

Esimerkki 10.8. Moniulotteista ohjelmointia

```
# Sama datatiedosto kuin esimerkissä 8.18, L08E18D1.txt
# Datatiedoston rivi: ”21-01-2017;3001;14624;10,81;16;507;346;70;26;1826”
# Varsinainen koodi kokonaisuudessa seuraavalla sivulla
```

```
import sys
import time
import numpy

KUUKAUSIA = 12
PAIVIA = 7
KUUKAUSI = ["Tammi", "Helmi", "Maalis", "Huhti", "Touko", "Kesä", "Heinä",
            "Elo", "Syys", "Loka", "Marras", "Joulu"]
PAIVA = ["Maanantai", "Tiistai", "Keskiviikko", "Torstai", "Perjantai",
             "Lauantai", "Sunnuntai"]

class DATA:
    Paivamaara = None
    Askelia = None

def lue(Nimi, Tiedot):
    Tiedot.clear()
    try:
        Tiedosto = open(Nimi, "r")
        Rivi = Tiedosto.readline()
        while (len(Rivi) > 0):
            Sarakkeet = Rivi[:-1].split(';')
            Data = DATA()
            Data.Paivamaara = time.strptime(Sarakkeet[0], "%d-%m-%Y")
            Data.Askelia = int(Sarakkeet[2]) # askeltiedot 2. sarake
            Tiedot.append(Data)
            Rivi = Tiedosto.readline()
        Tiedosto.close()
    except OSError:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(Nimi))
        sys.exit(0)
    return Tiedot

def analysoi(Tiedot, Matriisi):
    for Alkio in Tiedot: # Analyysi: tiedot listalta matriisiin
        Kuukausi = Alkio.Paivamaara.tm_mon - 1 # month-arvot ovat 1-12
        Paiva = Alkio.Paivamaara.tm_wday # viikonpäivä 0-6, maanantai 0
        Matriisi[Kuukausi][Paiva] += Alkio.Askelia
    return Matriisi

def tulosta(Matriisi):
    for Paiva in range(PAIVIA): # Otsikkorivin tulostus
        print(';'+PAIVA[Paiva], end='')
    print(';')

    for Kuukausi in range(KUUKAUSIA): # Tulokset excel-visualisointia varten
        print(KUUKAUSI[Kuukausi], end=';')
        for Paiva in range(PAIVIA):
            print(Matriisi[Kuukausi][Paiva], end=';')
        print()
    return None

def paaohjelma():
    Lista = [] # Alustuksia ja alkutoimenpiteitä
    Matriisi = numpy.zeros((KUUKAUSIA, PAIVIA), int)
    Tiedostonimi = "L08E18D1.txt" # input("Anna luettavan tiedoston nimi: ")
    Lista = lue(Tiedostonimi, Lista)
    Matriisi = analysoi(Lista, Matriisi)
    tulosta(Matriisi)
    Matriisi = numpy.delete(Matriisi, numpy.s_[:], None)
    Lista.clear()
    return None
paaohjelma()
# eof
```

Tuloste

```
;Maanantai;Tiistai;Keskiviikko;Torstai;Perjantai;Lauantai;Sunnuntai;
Tammi;28121;22191;11079;14148;23540;29487;20851;
Helmi;36843;37863;45092;49141;50866;50576;47468;
Maalis;38026;46399;59381;59265;54319;57333;43540;
Huhti;36048;35092;59421;47029;54332;53912;76597;
Touko;44889;40780;40992;37191;33262;37018;48360;
Kesä;40810;33772;41719;40482;46613;38424;27454;
Heinä;72741;61368;74987;51343;49371;64710;53880;
Elo;40495;50103;49441;51763;40201;51156;47082;
Syys;40558;32127;39916;42890;31902;60488;52177;
Loka;45154;47938;39192;45703;41581;47656;59852;
Marras;37066;44534;51230;35780;37863;37500;52049;
Joulu;44880;50602;41514;41980;48646;64563;56016;
```

Kuva 10.1. Esimerkin 10.8 tulostedata visualisoituna Excelillä