CSE-A1111
Ohjelmoinnin peruskurssi Y1
Opetusmoniste syksy 2015

Kerttu Pollari-Malmi

Sisältö

Luku 1

Tietokoneista ja ohjelmista

1.1 Lyhyesti tietokoneen rakenteesta

Tietokoneen perusosia ovat prosessori eli suoritin, keskusmuisti, syöttölaitteet ja tulostuslaiteet. Prosessori varsinaisesti suorittaa tietokoneohjelmat. Se pystyy esimerkiksi laskemaan kaksi lukua yhteen, tutkimaan onko jokin luku suurempi kuin nolla, hakemaan käsiteltäviä lukuja keskusmuistista jne.

Keskusmuisti on se tietokoneen osa, jossa sijaitsee parhaillaan suoritettava ohjelma ja sen käsittelemä data, esimerkiksi ne luvut, joille ohjelmassa pitää tehdä laskutoimituksia. Keskusmuisti koostuu peräkkäisistä muistipaikoista, joista jokaisella on oma tunnus, osoite. Yhteen muistipaikkaan voidaan tallentaa yksittäinen merkki tai osa yhdestä luvusta. Kun tietokone sammutetaan, sen keskusmuistissa oleva tieto katoaa.

Syöttölaitteella käyttäjä voi antaa käskyjä ja dataa tietokoneelle. Tyypillisiä syöttölaitteita ovat esimerkiksi näppäimistö ja hiiri. Tulostuslaitteilla, esimerkiksi näyttöpäätteellä ja tulostimella taas tietokone antaa tietoa käyttäjälle.

Keskusmuistin lisäksi tietokoneeseen on yleensä liitetty ulkoista muistia, esimerkiksi kovalevy ja väliaikaisesti muistitikku. Tällainen ulkoinen muisti voi toimia tietokoneessa sekä syöttö- että tulostuslaitteena.

1.2 Mikä on tietokoneohjelma?

Tietokoneen toimintaa ohjataan tietokoneohjelman avulla. Ohjelma on jono yksinkertaisia käskyjä, joita tietokoneen prosessori suorittaa järjestyksessä yksi kerrallaan. Tilannetta voi verrata siihen, että kokki tekee ruokaa keittokirjan ohjeen avulla. Kokki lukee reseptiä ja suorittaa siinä olevia käskyjä järjestyksessä. Kun keittokirjassa on käskyjä "vatkaa munat ja sokeri vaahdoksi", "lisää jauhot munasokerivaahtoon", niin tietokoneohjelmassa voi olla esimerkiksi seuraavanlaisia käskyjä: "pyydä käyttäjältä luku ja lue se", "vähennä luvusta 32", "kerro erotus viidellä", "jaa tulos yhdeksällä" ja "tulosta kuvaruudulle näin saatu lopputulos".

Tietokoneohjelmalla ja keittokirjan reseptillä on myös eroja. Reseptissä osa käskyistä voi olla epämääräisiä, esimerkiksi "lisää suolaa maun mukaan". Tietokoneohjel-

massa kaikkien käskyjen täytyy olla täsmällisiä ja täysin yksiselitteisiä. Reseptin käskyjä suoritetaan yleensä siinä järjestyksessä kuin ne on ohjeessa annettu. Tietokoneohjelmassa on usein käskyjä, jotka aiheuttavat sen, että ohjelman suorituksessa hypätään kokonaan toiseen paikkaan ohjelmassa.

Tietokoneen ymmärtämät käskyt ovat hyvin yksinkertaisia, kuten "Laske kaksi lukua yhteen", "Hyppää ohjelmassa kohtaan X". Tietokoneen tehokkuus ja monikäyttöisyys perustuu siihen, että tietokone pystyy suorittamaan näitä käskyjä todella nopeasti.

Lisäksi käskyt pitää esittää tietokoneelle bittijonoina eli erilaisina ykkösten ja nollien yhdistelminä. Tätä kutsutaan konekieleksi. Ensimmäisiä rakennettuja tietokoneita ohjelmoitiinkin niin, että koneessa olevien kytkimien avulla koneelle annettiin ykkösten ja nollien jonoja, jotka muodostivat käskyjä. Tällainen ohjelmointi oli luonnollisesti hyvin hidasta ja virhealtista. Jos yksikin kytkin oli väärässä asennossa, koko ohjelman suoritus meni sekaisin.

Ohjelmoinnin helpottamiseksi kehitettiin symbolinen konekieli, Assembler. Siinä käskyjä ei esitetä enää bittijonoina, vaan jokaista käskyä varten on sovittu määrätty sana, esimerkiksi yhteenlaskukäsky on yleensä add ja vähennyslaskukäsky sub. Lisäksi käskyt sisältävät tiedon siitä, missä ovat ne luvut, joille operaatio tehdään. Tietokone ei suoraan ymmärrä symbolisen konekielen käskyjä, vaan jotta symbolisella konekielellä kirjoitettu tietokoneohjelma voitaisiin suorittaa, pitää ensin muuttaa symbolisen konekielen käskyt tietokoneen ymmärtämiksi bittijonoiksi. Tätä muunnosta ei tarvitsee kuitenkaan suorittaa käsin, vaan voidaan kirjoittaa tietokoneohjelma, joka suorittaa muunnoksen. Jokaista symbolisen konekielen käskyä vastaa suoraan yksi määrätty bittijono, joten muunnoksen suorittaminen on hyvin suoraviivaista.

Symbolisenkin konekielen ongelmana on kuitenkin se, että ohjelmoija joutuu sitä käyttäessään ajattelemaan asioita huomattavasti yksityiskohtaisemmin kuin mikä on tarkoituksenmukaista. Ohjelmoijan pitää esimerkiksi koko ajan olla selvillä siitä, missä kohdassa tietokoneen muistissa käsiteltävät luvut täsmällisesti sijaitsevat. Jos haluataan laskea kaksi lukua yhteen, vaatii tämä usein monta konekielen käskyä, kun käsiteltäviä lukuja on ensin siirrettävä keskusmuistista prosessorin sisällä oleviin muistipaikkoihin. Näiden yksityiskohtien ajatteleminen tekee ohjelmoinnin työlääksi ja lisää myös virhemahdollisuuksia. Lisäksi jokaiselle prosessorityypille on oma konekieli. Yhdelle tietokoneelle symbolisella konekielellä kirjoitettu ohjelma ei toimi toisentyyppisessä tietokoneessa.

Symbolisen konekielen ongelmien ratkaisemiseksi ruvettiin vähitellen kehittämään lausekieliä. Lausekielisissä ohjelmissa monta peräkkäistä konekielen käskyä on korvattu yhdellä lausekielen käskyllä. Käskyt on suunniteltu niin, että ne vastaisivat sellaisia loogisia kokonaisuuksia, joita ohjelmoija ajattelee. Python on yksi lausekieli. Muita lausekieliä ovat esimerkiksi Cobol, Fortran, Pascal, C, C++ ja Java.

Tietokone ei kuitenkaan ymmärrä lausekielistä ohjelmaa sellaisenaan, vaan lausekielinen ohjelma on muutettava konekieliseksi, jotta se voitaisiin suorittaa. Tähän tarkoitukseen käytetään jälleen toista ohjelmaa. Tarkoitukseen käytettävät ohjelmat voidaan jakaa kahteen luokkaan, kääntäjiin ja tulkkeihin.

Kääntäjä ottaa koko lausekielisen ohjelman, muuttaa sen konekielelle ja tallentaa konekielisen ohjelman tiedostoon. Tämän jälkeen konekielinen ohjelma voidaan ajaa mielivaltaisen monta kertaa ilman, että ohjelmaa tarvitsee kääntää uudelleen.

Tulkki käy läpi lausekielistä ohjelmaa käsky kerrallaan, muuttaa käskyn konekielelle ja suorittaa konekielisen käskyn tai käskyt. Tämän jälkeen tulkki siirtyy lausekielisen ohjelman seuraavaan käskyyn. Konekielisiä käskyjä ei tallenneta minnekään, vaan jos ohjelma halutaan suorittaa uudelleen, pitää se myös tulkita kokonaan uudelleen. Python-kielisen ohjelman suorittamiseen käytetään tulkkia.

Lausekielten etu symboliseen konekieleen verrattuna on se, että ohjelmointi niillä on nopeampaa ja vähemmän virhealtista. Lisäksi ohjelmointi ei enää riipu siitä tietokoneesta, jolla ohjelma aiotaan suorittaa. Lausekielellä kirjoitettu ohjelma voidaan ajaa millä tietokoneella tahansa, kunhan tälle tietokoneelle on olemassa kääntäjä tai tulkki, joka muuttaa kyseisellä lausekielellä kirjoitetun ohjelman konekieliseksi.

Python-ohjelmointikielestä on eri versioita sen mukaan, kuinka uutta Python-tulkkia käytetään. Tällä kurssilla käytetään Python-versiota 3.4, koska tämä versio on tällä hetkellä asennettu Aalto IT:n Linux-tietokoneisiin. Kotikoneeseen voi hyvin asentaa myös version 3.3, sillä näiden versioiden välillä ei ole sellaisia eroja, joilla olisi merkitystä tällä kurssilla. Sen sijaan versiossa 2.7 ja sitä vanhemmissa versioissa on olennaisia muutoksia uudempiin verrattuna, eikä näiden versioiden käyttö ole tällä kurssilla enää mahdollista.

1.3 Mihin teekkari tai diplomi-insinööri tarvitsee ohjelmointia?

Suurin osa tietokoneen käytöstä on valmiiden ohjelmien (esimerkiksi tekstinkäsittely, taulukkolaskenta, www-selain) käyttöä. Näitä ohjelmia kirjoittavat yleensä ohjelmointialan ammattilaiset, eikä muun alan diplomi-insinöörin tarvitse tehdä niitä itse. Toisenkin alan diplomi-insinöörille ja teekkarille tulee kuitenkin vastaan tilanteita, joissa on hyödyllistä osata kirjoittaa oma tietokoneohjelma.

Yksi tyypillinen esimerkki on tilanne, jossa tarvitaan pientä laskentasovellusta. Pitäisi lukea lähtötiedot tiedostosta, tehdä niille joitakin laskutoimituksia, jotka saattavat sisältää esimerkiksi optimointia tai iterointia, ja tulostaa tulokset kuvaruudulle tai tiedostoon. Laskutoimituksia tarvitaan niin paljon, että niiden tekeminen taskulaskimella tai taulukkolaskentaohjelmalla on liian työlästä.

Toinen esimerkki on tilanne, jossa pitäisi liittää mittalaite tai vastaava tietokoneeseen. Usein tällaisen laitteen mukana tulee joukko aliohjelmia, joiden avulla voidaan esimerkiksi käynnistää laite, lopettaa sen toiminta ja muuttaa laitteen asetuksia. Laitteen käyttäjän on kuitenkin kirjoitettava itse pieni ohjelma, joka käynnistää laitteen mukana tulleet aliohjelmat halutussa järjestyksessä.

Myös erilaisia taulukko- ja matriisilaskentaohjelmia käytettäessä ohjelmointitaidosta on paljon hyötyä.

Ohjelmointitaito auttaa myös valmiiden kaupallisten ohjelmien käytön oppimisessa ja niiden toiminnan ymmärtämisessä. Kun on käsitys niistä periaatteista, jotka ovat ohjelmoinnin taustalla, on paljon helpompi ymmärtää, miksi valmis ohjelma toimii niin kuin se toimii.

Jos on kiinnostunut opiskelemaan ohjelmointia enemmän, niin työelämässä on paljon paikkoja sellaisille henkilöille, jotka hallitsevat hyvin jonkin insinöörialan ja osaavat

lisäksi ohjelmoida hyvin. Tällaiset henkilöt kirjoittavat yleensä omaan alaansa liittyviä ohjelmia. Tällaisiin työtehtäviin ei tosin tällä kurssilla opetettava ohjelmointitaito riitä, vaan silloin ohjelmointia on opiskeltava selvästi enemmän. Sen sijaan tässä luvussa aiemmin mainitut tavoitteet ovat saavutettavissa jo tämän kurssin jälkeen.

Luku 2

Python-ohjelmoinnin alkeita

2.1 Yksittäisten käskyjen antaminen Python-tulkille

Python on tulkattava kieli. Python-tulkki ottaa suoritettavasta ohjelmasta yhden käskyn kerrallaan, muuttaa sen konekielelle ja suorittaa käskyn saman tien. Tämä aiheuttaa sen, että Python-tulkilla ei tarvitse olla ohjelman loppua tiedossaan vielä silloin, kun se suorittaa ohjelman alkua. Niinpä Python-tulkin toimintaa voi aluksi kokeilla siten, että antaa tulkille käskyn kerrallaan. Aalto-yliopiston Linux-tietokoneissa Python-tulkki voidaan käynnistää kirjoittamalla komentoikkunassa käsky `python3`. Kuvaruudulle ilmestyy kehote `>>>`, jonka perään käyttäjä voi kirjoittaa haluamansa käskyn. Tulkki suorittaa käskyn saman tien ja tulostaa sen tuloksen kuvaruudulle. Alla on esimerkki istunnosta, jossa käyttäjä on antanut tulkille kaksi käskyä:

```
~ % python3
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 5 + 7
12
>>> print("kaunis ilma")
kaunis ilma
>>>
```

Ensimmäinen käsky `5 + 7` tarkoittaa sitä, että käyttäjä pyytää Python-tulkkia laskemaan annetun yhteenlaskun. Tulkki laskeekin sen ja seuraavalla rivillä antaa tuloksen. Jälkimmäinen käsky `print("kaunis ilma")` taas pyytää tulkkia tulostamaan tekstin "kaunis ilma". Tässä käskyssä sana print merkitsee pyyntöä tulostaa jotain ja tulostettava teksti on annettu sen jälkeen sulkujen sisällä. Koska tulostettava teksti halutaan tulostaa sellaisenaan, se on annettu lainausmerkeissä (muussa tapauksessa tulkki lähtisi tutkimaan, liittyykö sanoihin kaunis ja ilma jokin arvo, joka pitäisi ottaa huomioon). Kuten nähdään, tulkki on seuraavalla rivillä tulostanut pyydetyn tekstin.

Tulkin toiminta lopetetaan antamalla käsky `exit()` tai vaihtoehtoisesti tiedoston loppu -merkki, joka on Unix/Linux -käyttöjärjestelmässä Ctrl-D.

Huomautus: Python-tulkin käynnistävän käskyn tarkka muoto vaihtelee eri ympäristöissä. Usein se on vain pelkkä `python`. Aalto IT:n Linux-koneissa tämä käsky käynnistää kuitenkin vanhemman version Python-tulkista. Tällä kurssilla käytettävä versio käynnistetään edellä esitetyllä käskyllä `python3`.

2.2 Ohjelman kirjoittaminen tiedostoon

Käskyjen antaminen Python-tulkille yksitellen on kätevää silloin, jos haluaa vain tutkia yksittäisten käskyjen toimintaa. Menetelmässä on kuitenkin se huono puoli, että suoritetut käskyt eivät jää minnekään talteen. Jos samat käskyt haluaa suorittaa uudelleen, pitää ne kirjoittaa joka kerta yksi kerrallaan. Jos sama ohjelma halutaan suorittaa monta kertaa, kannattaa ohjelmaan kuuluvat käskyt kirjoittaa tiedostoon. Kun ohjelma on tallennettu tiedostoon, voidaan se suorittaa kuinka monta kertaa hyvänsä.

Esimerkiksi ohjelma, joka tulostaa kuvaruudulle "kaunis ilma", voidaan tehdä seuraavasti. Käynnistetään mikä tahansa tekstieditori (esimerkiksi Emacs tai Notepad), jonka avulla voidaan helposti tallentaa pelkkää tekstiä. Tavalliset tekstinkäsittelyohjelmat, esimerkiksi Word, eivät ole hyviä tähän tarkoitukseen, koska ne tallentavat tiedostoon itse tekstin lisäksi tekstin muotoiluun liittyvää tietoa, joka sekoittaa Python-tulkin. Kirjoitetaan tiedostoon rivi

```
print("kaunis ilma")
```

Jos kirjoitettava ohjelma sisältää useamman kuin yhden rivin, ne kirjoitetaan kaikki tähän samaan tiedostoon. Tämän jälkeen tiedosto pitää tallentaa. Siinä yhteydessä tiedostolle annetaan nimi. Tiedoston nimessä pitää olla pääte ".py", josta tunnistetaan, että tiedosto sisältää Python-ohjelman. Jos edellinen ohjelma on tallennettu tiedostoon tulostuskokeilu.py, voi sen ohjelman suorittaa käskyllä

```
python3 tulostuskokeilu.py
```

Ohjelman suoritus näyttää silloin seuraavalta:

```
> python3 tulostuskokeilu.py
kaunis ilma
```

Huomautus: Jälleen ohjelman suorituksessa tarvittava käsky voi olla pelkästään `python tulostuskokeilu.py` jossain toisessa ympäristössä kuin Aalto IT:n Linux-koneessa.

2.3 Ohjelman kirjoittaminen ja ajaminen Eclipsellä

Edellisessä esimerkissä neuvottiin, miten Python-ohjelma voidaan kirjoittaa millä tahansa tekstieditorilla ja ajaa sitten käyttöjärjestelmän puolella annettavalla

`python3`-käskyllä. Toinen vaihtoehto on käyttää ohjelman kirjoittamiseen integroitua kehitysympäristöä (engl. Integrated Development Environment, IDE). Se on työkalu, joka tyypillisesti sisältää välineet sekä ohjelmatiedoston kirjoittamiseen että ohjelman ajamiseen sekä muita apuvälineitä (esimerkiksi debuggerin, joka avulla voi helposti etsiä ohjelmasta virheitä). Tällä kurssilla ohjelmien kirjoittaminen käytetään Eclipse-työkalua. Tässä kappaleessa selitetään, miten ohjelma voidaan kirjoittaa ja ajaa Eclipse-työkalun avulla. Tarkemmat ohjeet Eclipsen käyttöön on tämän kurssin kotisivulla. Niinpä tässä opetusmonisteessa on esitetty vain lyhyesti periaatteet, ja yksityiskohdat on syytä lukea kurssin kotisivulta.

Jotta Eclipseä voisi käyttää Python-ohjelmointiin, pitää siihen ensin ladata sopiva lisäosa, plugin. Tällä kurssilla käytetään *Pydev*-nimistä pluginia. Tarkemmat ohjeet Pydevin asennukseen on kurssin kotisivulla.

Käynnistä Eclipse antamalla käyttöjärjestelmän puolella komento `eclipse` tai graafisessa ympäristössä kaksoisklikkaamalla Eclipsen kuvaketta. Pienen odottelun jälkeen näkyviin tulee Eclipsen ikkuna.

Eclipsessä kaikki ohjelmat kuuluvat johonkin projektiin. Jos kirjoitettavalle ohjelmalle sopivaa projektia ei ole aikaisemmin luotu, pitää se ensin luoda valitsemalla `File->New->Pydev Project`. Eclipse kysyy projektin nimeä. Eclipse kysyy myös kohdassa Grammar Version, mitä Pythonin versiota käytetään. Valitse tässä 3.0.

Kun projekti on luotu, pitää vielä luoda tiedosto, johon ohjelma kirjoitetaan. Tällaista tiedostoa kutsutaan *moduuliksi*. Se luodaan valitsemalla ensin projekti, johon moduuli tulee, ja sen jälkeen valikosta `File->New->Pydev Module`. Eclipse kysyy luotavan tiedoston (moduulin) nimen. Tässä yhteydessä olisi myös mahdollista määritellä pakkaus, mihin moduuli tulee. Samaan projektiin kuuluvat moduulit voidaan jakaa useisiin pakkauksiin. Tällä kurssilla tehtävät ohjelmat ovat kuitenkin niin pieniä, että pakkausten käyttö ei ole tarpeellista.

Kun moduulin nimi on annettu ja painettu finish-painiketta, ikkunan keskiosaan ilmestyy tila, johon ohjelman koodin voi kirjoittaa. Python-tulkki tutkii koodia samalla kuin ohjelmoija kirjoittaa sitä ja ilmoittaa havaitsemistaan virheistä punaisilla merkeillä. Jos kursorin vie punaisen merkin kohdalle, tulkki antaa tarkemman kuvauksen havaitsemastaan virheestä. Aina ei kuitenkaan ole kysymys varsinaisesta virheestä, vaan vain siitä, että kirjoitettava rivi tai ohjelma on vielä keskeneräinen, eikä sen vuoksi sisällä kaikkia tarpeellisia asioita. Tällöin virhemerkintä poistuu itsestään ohjelman kirjoituksen edetessä.

Kun ohjelma on kirjoitettu valmiiksi, eikä Eclipse löydä siitä enää virheitä, ohjelman voi ensin tallentaa `File->Save`-käskyllä ja sitten ajaa valitsemalla `Run->Run as->Python Run`. Ohjelman tulosteet tulevat nyt ohjelmakoodin sisältävän ikkunan osan alla olevaan konsoliosaan.

Eclipse tallentaa kaikki sillä kirjoitetut ohjelmatiedostot `moduulinnnimi.py`-nimisinä tavallisina teksitiedostoina hakemistoon `työhakemisto/projektinnimi/`. Näin Eclipsellä kirjoitettuja ohjelmia pystyy ajamaan myös ilman Eclipseä missä tahansa ympäristössä, jossa vain on Python-tulkki käytössä. Näin yleensä tehdäänkin valmiille ohjelmalle. Eclipse on nimenomaan ohjelman kehitysvaiheeseen tarkoitettu apuväline.

Edellä siis `moduulinnimi` tarkoittaa ohjelmoijan tälle moduulille antamaa nimeä. Kun ohjelmoija käyttää Eclipseä ensimmäistä kertaa, Eclipse pyytää häntä kerto-

maan työhakemiston, jonka alle kaikki ohjelmatiedostot tallennetaan. Tästä hakemistossa on edellä käytetty nimeä `työhakemisto`.

2.4 Muuttujat, sijoituskäsky ja tiedon lukeminen käyttäjältä

Yleensä tietokoneohjelmissa halutaan, että ohjelma voi lukea jotain tietoa käyttäjältä ja käyttää tätä tietoa hyväkseen. Ajatellaan esimerkiksi ohjelmaa, joka pyytää käyttäjältä lämpötilan fahrenheit-asteina ja tulostaa saman lämpötilan celsius-asteina. Ohjelmassa täytyy tällöin olla jokin tapa tallentaa käyttäjän antama fahrenheit-arvo ja käsitellä sitä.

Tällaiseen arvojen tallentamiseen voidaan käyttää *muuttujia*. Jokaisella muuttujalla on nimi ja muuttujalle voi antaa arvon. Muuttujan nimen avulla pääsee käsiksi tähän muuttujan arvoon myöhemmin ohjelmassa. Muuttujalle voi antaa arvon sijoituskäskyn avulla. Sijoituskäskyssä on vasemmalla sen muuttujan nimi, jolle arvo halutaan antaa, sitten sijoitusoperaattori `=`, ja oikealla muuttujalle annettava arvo. Esimerkiksi käsky

```
fahrenheit = 78.5
```

sijoittaa muuttujan `fahrenheit` arvoksi 78.5. Sijoitettavan arvon ei tarvitse olla suoraan jokin vakioarvo, vaan sijoitusoperaattorin oikealla puolella voi olla jokin lauseke, jonka arvo lasketaan ensin. Tämän jälkeen laskettu arvo annetaan arvoksi vasemmalla puolella olevalle muuttujalle, esimerkiksi käsky

```
celsius = (fahrenheit - 32) * 5.0 / 9.0
```

vähentää muuttujan `fahrenheit` arvosta 32, kertoo näin saadun tuloksen 5.0:lla ja jakaa sen 9.0:lla ja sijoittaa näin saadun tuloksen muuttujan `celsius` arvoksi.

Ennen kuin lähdetään tutkimaan fahrenheit-celsius-muunnoksen tekevää ohjelmaa ja lämpötilatiedon lukemista käyttäjältä tarkemmin, tarkastellaan vielä yksinkertaisempaa ohjelmaa. Se pyytää käyttäjää antamaan nimensä, lukee käyttäjän antaman nimen ja tulostaa sitten käyttäjälle osoitetun tervehdyksen.

Käyttäjän syötettä voidaan lukea käskyn `input` -avulla. Esimerkiksi käsky

```
nimi = input("Kerro nimesi: ")
```

tulostaa ensin kuvaruudulle tekstin "Kerro nimesi: ", lukee sitten käyttäjän antaman tekstin ja sijoittaa sen muuttujan `nimi` arvoksi. Käskyssä on siis `input`in jälkeen tulevien sulkujen sisällä se teksti, joka käyttäjälle halutaan ensin tulostaa. Luettu arvo annetaan jonkin muuttujan arvoksi sijoituskäskyn avulla.

Seuraavaksi koko ohjelma, joka lukee käyttäjän tekstin ja tulostaa tervetulotoivotuksen:

```
nimi = input("Kerro nimesi: ")
print("Hei,", nimi)
print("Tervetuloa Python-kurssille!")
```

Nimen lukemisen jälkeen ohjelmassa on siis kaksi tulostuskäskyä. Ensimmäinen niistä tulostaa tekstin "Hei," sellaisenaan ja sen jälkeen samalle riville muuttujan `nimi` arvon. Sana Hei on lainausmerkkien sisällä, koska se halutaan tulostaa sellaisenaan. Sana `nimi` sen sijaan ei ole lainausmerkkien sisällä, koska ohjelman ei haluta tulostaa tekstiä nimi, vaan muuttujan nimi arvo. Jos samassa tulostuskäskyssä halutaan tulostaa enemmän kuin yksi asia (esimerkkitapauksessa sekä teksti että muuttujan nimi arvo), erotetaan tulostettavat asiat toisistaan pilkulla. Tulostettavien asioiden välille lisätään tulostuksessa yksi välilyönti.

Ohjelman viimeinen rivi tulostaa tekstin "Tervetuloa Python-kurssille!". (Niiden, jotka ovat aikaisemmin ohjelmoineet jollain toisella kielellä kannattaa huomata, että Python-ohjelmissa muuttujia ei määritellä toisin kuin esimerkiksi Java- tai C-ohjelmissa.)

Kun ohjelma ajetaan, näyttää sen tulostus seuraavalta:

```
Kerro nimesi: Maija
Hei, Maija
Tervetuloa Python-kurssille!
```

Fahrenheit-Celsius-muunnoksen laskeva ohjelma on vähän monimutkaisempi. Tämä johtuu siitä, että `input`-käsky välittää käyttäjältä lukemansa syötteen aina merkkijonona eli tekstinä, joka koostuu peräkkäisistä merkeistä (näitä merkkejä voi olla yksi, useampi tai jopa nolla). Lämpötilamuunnoksessa tarvitaan kuitenkin yhteen-, kerto- ja jakolaskuja. Python-tulkki ei voi suorittaa näitä laskutoimituksia merkkijonoille, vaan laskutoimituksen operandin on oltava luku. Oletetaan, että käyttäjän antama merkkijono on tallennettu muuttujaan `rivi`. Merkkijonoa vastaava luku voidaan tällöin antaa muuttujan `fahrenheit` arvoksi käskyllä

```
fahrenheit = float(rivi)
```

Tässä käsky `float` muuttaa sille sulkujen sisällä annetun merkkijonon tai sulkujen sisällä annetun muuttujan arvona olevan merkkijonon vastaavaksi desimaaliluvuksi. Muutos onnistuu vain, jos merkkijono todella esittää jotain desimaalilukua. Seuraavaksi koko ohjelma:

```
rivi = input("Anna lampotila fahrenheit-asteina: ")
fahrenheit = float(rivi)
celsius = (fahrenheit - 32) * 5.0 / 9.0
print(fahrenheit, "F on", celsius, "C")
```

Ohjelman suoritus voi näyttää seuraavalta:

```
Anna lampotila fahrenheit-asteina: 79.5
79.5 F on 26.3888888889 C
```

Kuten esimerkkiajosta nähdään, celsius-lämpötila esitetään turhan tarkasti. Luvussa 3.2.5 kerrotaan, miten tulosta voi muotoilla siistimmäksi.

Jos käyttäjän antamaa merkkijono ei vastaa mitään lukua, ohjelma kaatuu eli se lopettaa toimintansa ja tulostaa jonkinlaisen virheilmoituksen. Alla esimerkki ohjelman tällaisesta suorituksesta:

```
Anna lampotila fahrenheit-asteina: ei mikaan luku
Traceback (most recent call last):
  File "/m/home/lampotilat1.py", line 2, in <module>
    fahrenheit = float(rivi)
ValueError: could not convert string to float: 'ei mikaan luku'
```

Virheilmoitus ei näytä kovin helppolukuiselta, mutta se kertoo, missä ohjelman kohdassa virhe on sattunut, millainen virhe oli kysymyksessä ja mikä virheen aiheutti.

Edellisessä esimerkissä ei ole käytetty ohjelman tulostuksissa skandinaavisia aakkosia ä, ö ja å, vaan ne on korvattu a:lla ja o:lla. Tämä ei ole mitenkään pakollista, vaan Python-ohjelmissa pystytään käsittelemään myös näitä kirjaimia. Käytännössä tällä kurssilla on kuitenkin havaittu, että ä- ja ö-kirjainten käyttäminen aiheuttaa opiskelijoille usein harmia kurssilla käytettävän automaattisen tarkastusjärjestelmän käytössä, kun opiskelijat tekevät harjoitustehtäviä omilla kotikoneillaan, mutta eivät osaa asettaa käytettäviä merkkien koodaustapoja oikein. Koska näistä asioista huolehtiminen ei ole tällä kurssilla keskeinen asia, on opiskelijoiden harjoitustehtävien tekemisen helpottamiseksi tällä kurssilla luovuttu skandinaavisten aakkosten käyttämisestä ohjelman tulostuksista.

2.5 Erilaisia tyyppejä

Monessa muussa ohjelmointikielessä pitää jo ennen muuttujan ensimmäistä käyttöä määritellä muuttuja ja kertoa, mikä muuttujan tyyppi on. Muuttujan tyyppi määrää, millaisen arvon muuttujalle voi antaa. Python-ohjelmissa asia ei ole näin, vaan muuttuja otetaan käyttöön ilman määrittelyä. Siitä huolimatta muuttujalle sijoitettavalla arvolla on jokin tyyppi, joka määrää esimerkiksi sen, millaisia operaatiota muuttujalle voidaan suorittaa. Kaksi kokonaislukua voidaan jakaa keskenään, mutta kahta merkkijonoa ei.

Seuraavaksi esitellään yleisimmät yksinkertaisissa ohjelmissa esiintyvät tyypit.

Merkkijonoja (englannniksi string) käytetään käsittelemään erilaisia tekstejä, esimerkiksi nimiä. Merkkijonoille ei voi suorittaa aritmeettisia laskutoimituksia, mutta merkkijonosta voi esimerkiksi selvittää sen pituuden ja kaksi merkkijonoa voi liittää peräkkäin yhteen. Merkkijonoille mahdollisista operaatioista kerrotaan enemmän luvussa 5. Merkkijonon tyyppi Pythonissa on `str`.

Kokonaisluvut ovat lukuja, joissa ei ole desimaalipistettä. Niiden arvon tyyppi on Pythonissa `int`. Jos lukuarvossa on desimaalipiste, sen tyyppi ei ole `int`, vaan `float`. Tätä tyyppiä käytetään desimaalilukujen arvojen tallentamiseen. On kuitenkin huomattava, että `float`-arvot eivät ole reaalilukuja matemaattisessa mielessä. Arvon

tallentamiseen käytettävissä oleva muistitila asettaa rajat sille, kuinka tarkasti desimaalilukuja voidaan esittää ja mikä on suurin tai pienin mahdollinen desimaaliluku. Käytännössä tämä voi johtaa yllättäviin pyöristysvirheisiin ohjelman toiminnassa. Jos esimerkiksi lasketaan yhteen hyvin suuri ja hyvin pieni positiivinen desimaaliluku, ei pienempi luku vaikuta välttämättä lopputulokseen lainkaan, koska summan tarkkuus ei riitä kattamaan pienemmän luvun vaikutusta.

Desimaaliluku käsitetään `float`-tyyppiseksi, vaikka sen desimaaliosa olisikin 0. Näin esimerkiksi arvon 5 tyyppi on Python-tulkin mielestä `int`, mutta arvon 5.0 tyyppi on `float`. Tällä on merkitystä, sillä `int`- ja `float`-tyyppiset arvot tallennetaan tietokoneen muistissa erilaisissa muodoissa ja myös jotkin operaatiot toimivat eri lailla kokonais- ja desimaaliluvuille.

Lisäksi Python-kielessä on tyyppi `bool`, jota käytetään totuusarvojen esittämiseen. Arvona on voi tällöin olla joko `True` (tosi) tai `False` (epätosi).

2.6 Laskutoimituksia

Python-kielessä kokonais- ja desimaaliluvuille on käytettävissä laskutoimituksia varten operaattorit `+`, `-`, `*`, `/`, `//` `%`, `*` ja `**`. Operaattori `+` tarkoittaa yhteenlaskua, `-` vähennyslaskua, `*` kertolaskua, ja `**` potenssiin korotusta. Nämä operaatiot toimivat aivan niin kuin arkikokemuksen perusteella voisi olettaa. Poikkeusena on edellisessä luvussa mainitut, mahdolliset pyöristysvirheistä johtuvat epätarkkuudet.

Operaattorit `/` ja `//` suorittavat jakolaskun. Operaattori `/` toimii, kuten arkikokemuksen perusteella jakolaskun voisi olettaa toimivan pyöristysvirheitä lukuunottamatta. Esimerkiksi laskutoimituksen `5 / 3` tulos on `1.6666666666666667`. Sen sijaan käytettäessä operaattoria `//` kahden positiivisen kokonaisluvun välillä tulokseksi tulee varsinaisen jakolaskun tuloksen kokonaisosa. Niinpä esimerkiksi laskutoimituksen `5 // 3` tulos on `1`. Tulosta laskettaessa ei suoriteta mitään pyöristyksiä, vaan jakolaskun tuloksen kokonaisosa otetaan tulokseksi sellaisenaan, tosin negatiivisilla tuloksilla otetaan tulosta pienempi kokonaisluku. Operaattori `//` katkaisee tuloksesta desimaalit pois myös silloin, kun jaettavana on desimaalilukuja. Tuloksen tyyppi on kuitenkin tällöin `float` eikä `int`.

Operaattori `%` on jakojäännös. Esimerkiksi lausekkeen `19 % 4` arvo on `3`. Operaattori `**` tarkoittaa potenssiin korotusta. Esimerkiksi lausekkeen `3 ** 4` arvo on `81` ja lausekkeen `3.0 ** 4` arvo on `81.0`.

Näiden operaatioiden lisäksi voidaan Python-ohjelmissa ottaa käyttöön valmiita moduuleita, jotka sisältävät paljon lisää laskutoimituksia, esimerkiksi trigonometrisia funktioita ja logaritmeja. Näistä kerrotaan tarkemmin myöhemmin.

2.7 Lisää sijoituskäskystä

Tarkastellaan vähän tarkemmin sijoituskäskyä. Sijoituskäskyssä on vasemmalla jonkin muuttujan nimi, sitten `=`-merkki ja tämän oikealla puolella lauseke, esimerkiksi

```
keskiarvo = (4.5 + 8.7) / 2
```

Sijoituskäsky suoritetaan aina niin, että ensin lasketaan oikealla puolella olevan lausekkeen arvo ja saatu tulos sijoitetaan vasemmalla puolella olevan muuttujan arvoksi.

Laskettavassa lausekkeessa voi esiintyä muuttujien nimiä. Tällöin lausekkeen arvoa laskettaessa käytetään näiden muuttujien arvoja, esimerkiksi

```
fahrenheit = 100;
celsius = (fahrenheit - 32) * 5.0 / 9.0
```

Sijoituskäskyn vasemmalla puolella olevalla muuttujalla voi olla arvo jo ennen sijoituskäskyä. Tällöin vanha arvo häviää sijoituskäskyä suoritettaessa, esimerkiksi

```
celsius = 25.0
fahrenheit = 100
celsius = (fahrenheit - 32) * 5.0 / 9.0
```

Muuttujan `celsius` arvo on ennen viimeisen käskyn suorittamista 25.0, mutta sen suorittamisen jälkeen noin 37.7777777778.

Muuttujan vanhaa arvoa voidaan käyttää myös hyväksi saman muuttujan uutta arvoa laskettaessa, esimerkiksi

```
luku = 5
luku = luku + 7
```

Jälkimmäinen sijoituskäsky suoritetaan seuraavasti: lasketaan ensin oikealla puolella olevan lausekkeen arvo. Otetaan siis muuttujan `luku` arvo (5), lisätään siihen 7, jolloin lausekkeen arvoksi saadaan 12. Saatu arvo sijoitetaan muuttujan `luku` uudeksi arvoksi. Muuttujan `luku` arvo on siis ennen käskyn suorittamista 5 ja käskyn suorittamisen jälkeen 12. Käsky voi tuntua aluksi hämäävältä, mutta on muistettava, että merkki `=` tarkoittaa todellakin sijoitusta, eikä sillä ole Python-kielessä mitään tekemistä yhtäsuuruuden kanssa.

Sijoituskäskyt, joissa muuttujan vanhaa arvoa käytetään hyväksi uutta arvoa laskettaessa ovat niin yleisiä, että osalle niistä on sovittu lyhennysmerkintä. Sijoitus

```
luku += 7
```

tarkoittaa samaa kuin sijoituskäsky

```
luku = luku + 7
```

Vastaavasti toimivat merkinnät `-=`, `*=` ja `/=`.

2.8 Ohjelman jako funktioihin ja pääohjelma

Tähän asti esitetyt esimerkkiohjelmat ovat olleet korkeintaan muutaman rivin mittaisia. Käytännön elämässä tarvitaan kuitenkin usein ohjelmia, jotka sisältävät satoja, tuhansia ja jopa kymmeniä tuhansia rivejä. Jos tällöin koko pitkä ohjelma koostuisi vain yhdestä osasta, joka sisältäisi suoritettavat ohjelmarivit peräkkäin, olisi ohjelman rakenteen ja toiminnan hahmottaminen hyvin vaikeaa. Ohjelman rakennetta selkiytetään sillä, että ohjelma jaetaan funktioihin. Yhteen funktioon kirjoitetaan muutamasta rivistä muutamaan kymmeneen riviin ohjelmakoodia, joka tyypillisesti suorittaa jonkin tehtävän. Funktiolle annetaan sen tarkoitusta kuvaava nimi. Kun ohjelmassa sitten halutaan suorittaa tähän funktioon kuuluvat käskyt, kirjoitetaan ohjelmaan rivi, joka *kutsuu* tätä funktiota eli pyytää suorittamaan tämän funktion.

Sen lisäksi, että ohjelman jakaminen funktioihin selkeyttää ohjelmaa, niin se myös säästää työtä. Jos samanlainen ohjelman osa pitää suorittaa monta kertaa, riittää kirjoittaa tähän osaan kuuluvat käskyt yhden kerran yhdeksi funktioksi. Tämän jälkeen riittää kirjoittaa yksi käsky, funktion kutsu, aina kun kyseinen osa halutaan suorittaa.

Luvussa 4 kerrotaan tarkemmin, miten funktioita määritellään ja mitä kaikkea muuta asiaan liittyy. Tässä vaiheessa opetellaan kuitenkin yhden erityisen funktion, *pääohjelman* kirjoittaminen ja kutsuminen.

Tyypillisesti Python-ohjelmassa on yksi erityisasemassa oleva funktio, jota kutsutaan pääohjelmaksi. Tämän funktion nimeksi annetaan yleensä `main` ja se on se osa ohjelmasta, josta ohjelman suoritus aloitetaan. Kun ohjelma käynnistetään, suoritetaan pääohjelmassa olevia käskyjä järjestyksessä, kunnes jokin pääohjelmassa oleva käsky (esimerkiksi toisen funktion kutsu) aiheuttaa sen, että siirrytään ohjelmassa johonkin muualle. Pääohjelman määrittelyssä kerrotaan, mitä käskyjä pääohjelmassa on. Määrittely aloitetaan rivillä

```
def main():
```

Sana `def` kertoo, että ollaan määrittelemässä funktiota, `main` on määriteltävän funktion nimi ja itse määrittely tulee kaksoispisteen jälkeen. Nimen jälkeen olevat sulut liittyvät siihen, että määriteltäville funktioille on mahdollista määritellä myös parametreja, joista kerrotaan tarkemmin luvussa 4.

Tämän rivin jälkeen tulevat varsinaiset pääohjelmaan kuuluvat käskyt. Jokainen pääohjelmaan kuuluva rivi on sisennetty. Sisennysten avulla osoitetaan, mitkä rivit kuuluvat pääohjelman määrittelyyn. Ensimmäinen näiden rivien jälkeen tuleva sisentämätön rivi ei enää kuulu pääohjelmaan. Esimerkiksi jos aikaisemmin esitetty fahrenheit-celsius-muunnoksen tekevä ohjelma kirjoitetaan pääohjelmaksi, näyttää se seuraavalta:

```
def main():
    rivi = input("Anna lampotila fahrenheit-asteina: ")
    fahrenheit = float(rivi)
    celsius = (fahrenheit - 32) * 5.0 / 9.0
    print(fahrenheit, "F on", celsius, "C")
```

Pelkkä pääohjelman määrittely ei saa kuitenkaan ohjelmaa vielä tekemään mitään. Ohjelmassa pitää myös kutsua pääohjelmaa eli kirjoittaa käsky, joka saa aikaiseksi sen, että suoritetaan `main`-niminen funktio (pääohjelma). Tämä käsky on yksinkertaisesti seuraava:

```
main()
```

eli kutsuttavan funktion nimi ja sen perässä sulut. Pääohjelman kutsu voidaan kirjoittaa ohjelmatiedostossa pääohjelman määrittelyn jälkeen. Kutsua sisältävää riviä ei ole enää sisennetty, koska pääohjelman kutsu ei kuulu pääohjelman määrittelyyn.

Kokonaisuudessaan siis esimerkkiohjelman sisältävään tiedostoon kirjoitetaan seuraavat rivit:

```
def main():
    rivi = input("Anna lampotila fahrenheit-asteina: ")
    fahrenheit = float(rivi)
    celsius = (fahrenheit - 32) * 5.0 / 9.0
    print(fahrenheit, "F on", celsius, "C")

main()
```

2.9 Joitakin esimerkkiohjelmia

2.9.1 Huoneen pinta-ala

Seuraava ohjelma pyytää käyttäjältä suorakulmion muotoisen huoneen pituuden ja leveyden. Se laskee ja tulostaa huoneen pinta-alan. Esimerkistä nähdään, kuinka käyttäjältä voidaan lukea samassa ohjelmassa useampi arvo. Kukin luettu arvo (sen jälkeen, kun se on muutettu desimaaliluvuksi) tallennetaan omaan muuttujaansa.

```
def main():
    rivi = input("Anna huoneen leveys metreina: ")
    leveys = float(rivi)
    rivi = input("Anna huoneen pituus metreina: ")
    pituus = float(rivi)
    pinta_ala = leveys * pituus
    print("Huoneen pinta-ala on", pinta_ala, "neliometria")

main()
```

Alla on esimerkki ohjelman suorituksesta.

```
Anna huoneen leveys metreina: 4.5
Anna huoneen pituus metreina: 3.8
Huoneen pinta-ala on 17.1 neliometria
```

Tähänastisissa esimerkkiohjelmissa on käytetty lyhyissä Python-ohjelmissa varsin yleistä tapaa, jossa käyttäjälle annettava kehote (pyyntö) on samalla rivillä kuin mille käyttäjä kirjoittaa oman syötteensä (vastauksen ohjelman kysymykseen). Tällä kurssilla tehtävissä harjoitustehtävissä tavan käyttö aiheuttaa kuitenkin ongelmia, koska harjoitustehtävät automaattisesti tarkastava Goblin-järjestelmä tutkii ohjelmien tulostetta rivi kerrallaan. Ohjelman tulostuksen ja käyttäjän syötteen erottaminen tarkistuksessa on järjestelmälle ongelmallista, jos ohjelman tuloste ei ole välimerkkejä myöten juuri annetun mallin mukainen.

Turhien ongelmien välttämiseksi tällä kurssilla kirjoitetaan sen vuoksi palautettavat harjoitustehtävät niin, että ohjelman tulostukset päätetään aina rivinvaihtoon ja käyttäjä kirjoittaa mahdollisen syötteensä vasta seuraavalle riville. Käskyn `input` sulkujen sisällä olevan tulostettavan merkkijonon saa päättymään rivinvaihtoon sillä, että lisää merkkijonon loppuun erikoismerkin `\n`. Tämä merkki siis kirjoitetaan painamalla näppäimistöllä ensin näppäintä `\` ja näppäintä `n`, mutta merkkijonossa yhdistelmä käsitetään yhdeksi merkiksi, joka aiheuttaa rivinvaihdon merkkijonoa tulostettaessa. Käsky `print` lisää itse rivinvaihdon tulostuksen jälkeen, joten `print`-käskyä käytettäessä rivinvaihtoa ei tarvitse lisätä.

Alla pinta-alan laskeva ohjelma kirjoitettuna uudelleen niin, että kaikki ohjelman tulosteet päättyvät rivinvaihtoon ja käyttäjä syöte tulee omalle rivilleen.

```
def main():
    rivi = input("Anna huoneen leveys metreina.\n")
    leveys = float(rivi)
    rivi = input("Anna huoneen pituus metreina.\n")
    pituus = float(rivi)
    pinta_ala = leveys * pituus
    print("Huoneen pinta-ala on", pinta_ala, "neliometria")

main()
```

Esimerkki ohjelman suorituksesta:

```
Anna huoneen leveys metreina.
3.7
Anna huoneen pituus metreina.
5.3
Huoneen pinta-ala on 19.61 neliometria
```

Rivinvaihtojen lisääminen ei ole mikään Python-ohjelmoinnin vaatima toimenpide, vaan ne on lisätty ainoastaan tällä kurssilla käytettävän automaattisen tarkastusjärjestelmän toiminnan helpottamiseksi. Jatkossa esitettävissä esimerkeissä osassa on tulostuksiin lisätty rivinvaihdot, osassa ei. Kurssilla palautettavissa harjoitustehtävissä rivinvaihdot on kuitenkin syytä olla aina mukana (silloin, kun ei tulosteta `print`-käskyllä, joka lisää itse rivinvaihdon).

2.9.2 Puhelun hinta

Monessa matkapuhelinliittymässä puhelun hinta on koostuu aloitusmaksusta, joka ei riipu puhelun kestosta, sekä puhelun kestosta riippuvasta minuuttihinnasta. Seu-

raava ohjelma pyytää käyttäjältä aloitusmaksun, minuuttihinnan sekä puhelun keston minuutteina ja laskee puhelun hinnan. Yksinkertaisuuden vuoksi puhelun kesto annetaan desimaalilukuna. Toinen vaihtoehto olisi antaa kesto kokonaisina minuutteina ja sekunteina, joista sitten laskettaisiin vastaava kesto desimaalilukuna.

```
def main():
    print("Ohjelma laskee matkapuhelun hinnan.")
    rivi = input("Anna puhelun aloitusmaksu.\n")
    aloitusmaksu = float(rivi)
    rivi = input("Anna puhelun minuuttihinta.\n")
    minuuttihinta = float(rivi)
    rivi = input("Anna puhelun kesto minuutteina.\n")
    kesto = float(rivi)
    hinta = aloitusmaksu + kesto * minuuttihinta
    print("Puhelun hinta on", hinta, "euroa.")

main()
```

Esimerkki ohjelman suorituksesta:

```
Ohjelma laskee matkapuhelun hinnan.
Anna puhelun aloitusmaksu.
0.049
Anna puhelun minuuttihinta.
0.079
Anna puhelun kesto minuutteina.
8.5
Puhelun hinta on 0.7205 euroa.
```

2.9.3 Aikamuunnoksia

Välillä tulee vastaa tilanteita, joissa pitempi aika pitää ilmaista sekunteina esimerkiksi erilaisia laskutoimituksia varten. Tämä muunnos on helppo totetuttaa pienellä tietokoneohjelmalla. Toisaalta monesti on vaikea hahmottaa suoraan, kuinka pitkä aika on esimerkiksi 15600 sekuntia. Kun saman ajan ilmaisee tuntien ja minuuttien avulla, tulee aikajakson pituus heti selväksi. Myös tämä muunnos on helppo laskea pienen tietokoneohjelman avulla.

Ensimmäinen esimerkkiohjelma muuttaa tunteina, minuutteina ja sekunteina annetun aikamäärän pelkiksi sekunneiksi. Oletetaan, että aika annetaan sekuntien tarkkuudella (sekuntien osia ei käsitellä), jolloin nyt voidaan käyttää kokonaislukutyyppiä `int` annettujen lukujen käsittelyyn.

```
def main():
    print("Tama ohjelma muuttaa annetun aikajakson pituuden sekunneiksi.")
    rivi = input("Anna aikajakson tunnit.\n")
    tunnit = int(rivi)
    rivi = input("Anna aikajakson minuutit.\n")
    minuutit = int(rivi)
    rivi = input("Anna aikajakson sekunnit.\n")
    sekunnit = int(rivi)
    aika_sekunteina = tunnit * 3600 + minuutit * 60 + sekunnit
    print("Aikajakson pituus on", aika_sekunteina, "sekuntia.")

main()
```

Esimerkki ohjelman suorituksesta:

```
Tama ohjelma muuttaa annetun aikajakson pituuden sekunneiksi.
Anna aikajakson tunnit.
4
Anna aikajakson minuutit.
37
Anna aikajakson sekunnit.
14
Aikajakson pituus on 16634 sekuntia.
```

Muunnos sekunneista tunneiksi, minuuteiksi ja sekunneiksi on vähän monimutkaisempi. Tässä voidaan kuitenkin käyttää hyväksi sitä, että käsitellään kokonaislukuja. Kuten aikaisemmin on kerrottu, Pythonissa on tavallisen jakolaskuoperaattorin `/` vaihtoehtona operaattori `//`, jota käytettäessä kahden kokonaisluvun jakolaskun tulos on kokonaisluku, johon on otettu mukaan vain jakolaskun tuloksen kokonaisosa. Mitään pyöristyksiä ei siis tehdä. Näin johonkin sekuntimäärään sisältyvät täydet tunnit saadaan selville jakamalla tämä sekuntimäärä 3600:lla. Näiden tuntien jälkeen ylijääneet sekunnit saadaan taas selville ottamalla edellisen jakolaskun jakojäännös, joka voidaan tehdä `%`-operaattorin avulla. Vastaavalla tavalla saadaan selville jäljelle jääneisiin sekunteihin sisältyvät kokonaiset minuutit ja sekunnit.

```
def main():
    rivi = input("Anna aikajakson pituus sekunteina.\n")
    pituus_sekunteina = int(rivi)
    tunnit = pituus_sekunteina // 3600
    jaannossekunnit = pituus_sekunteina % 3600
    minuutit = jaannossekunnit // 60
    sekunnit = jaannossekunnit % 60
    print("Pituus on", tunnit, "h", minuutit, "min", sekunnit, "s.")


main()
```

Esimerkki ohjelman suorituksesta:

```
Anna aikajakson pituus sekunteina.
18756
Pituus on 5 h 12 min 36 s.
```

Alla on toinen esimerkki ohjelman suorituksesta. Tästä nähdään, että ohjelma ei osaa jättää pois tuntia, minuutteja tai sekunteja siinä tapauksessa, että jokin niistä on nolla. Ohjelman tulostus ei ole tässäkään tapauksessa virheellinen, mutta se voisi olla lyhyempi. Nollien poisjättämiseen tarvitaan kuitenkin valintakäskyä, joka opetetaan vasta seuraavassa luvussa.

```
Anna aikajakson pituus sekunteina.
7218
Pituus on 2 h 0 min 18 s.
```

2.10 Kommentit

Ohjelmaan on myös mahdollista lisätä selitystekstiä, joka ei vaikuta mitenkään ohjelman suoritukseen, mutta joka auttaa ohjelmatekstin lukijaa ymmärtämään koodia. Tällaisia selitystekstejä kutsutaan *kommenteiksi*. Python-ohjelmissa kommentit merkitään `#`-merkillä. Kun Python-tulkki ohjelmaa lukiessaan kohtaa `#`-merkin, se jättää ottamatta huomioon kaiken tämän merkin jälkeen tulevan tekstin rivin loppuun asti.

Kommenttien järkevä käyttö helpottaa huomattavasti ohjelmaa lukevaa ihmistä koodin ymmärtämisessä. Tällä on merkitystä erityisesti silloin, kun halutaan myöhemmin muuttaa ohjelmaa – esimerkiksi lisätä siihen uusia ominaisuuksia – tai käyttää aikaisemmin tehtyä ohjelmaa jonkin uuden ohjelman pohjana.

Ohjelman alkuun kannattaa aina lisätä kommentti, joka kertoo, mitä ohjelma tekee, kuka sen on kirjoittanut ja koska ohjelmaa on viimeksi muokattu, esimerkiksi:

```
# Ohjelma, joka muuttaa kayttajan maileina antaman matkan kilometreiksi.
# Kirjoittanut Maija Meikalainen.
# Viimeksi muutettu 9.1.2013.

def main():
    syote = input("Anna matka maileina: ")
    mailit = float(syote)
    kilometrit = 1.6093 * mailit
    print("Matka on", kilometrit, "km.")


main()
```

Suuremmissa ohjelmissa kannattaa ohjelman sisälle kirjoittaa kommentteja kunkin funktion merkityksestä. Myös funktioiden sisällä voi kommentoida kohtia, joiden merkitys ei selviä helposti koodia lukemalla.

Kommentteja voi kirjoittaa myös varsinaisen ohjelmarivin loppuun, esimerkiksi

```
kilometrit = 1.6093 * mailit  #Muuta mailit kilometreiksi kertoimen avulla.
```

Pythonin tyyliopas `http://www.python.org/dev/peps/pep-0008/` kuitenkin neuvoo käyttämään rivin loppuun kirjoitettavia kommentteja hyvin säästeliäästi ja suosimaan omille riveilleen kirjoitettavia kommentteja aina, kun se on järkevää.

Merkillä `#`-alkavien kommenttien sijaan ohjelmassa olevien kokonaisuuksien kommentoimiseen voi käyttää *dokumentointimerkkijonoja*, (engl. documentation strings, docstrings). Niitä ei kuitenkaan käsitellä tässä opetusmonisteessa.

Luku 3

Kontrollirakenteet: valinta ja toisto

3.1 Valintakäsky if

Tähän asti esitetyt ohjelmat ovat aina suorittaneet samat käskyt samassa järjestyksessä. Usein kuitenkin haluamme, että ohjelma toimii eri tilanteissa eri tavoilla, esimerkiksi niin, että ohjelman toiminta riippuu käyttäjän antamista syötteistä.

Oletetaan, että erääseen tilaisuuteen myydään lippuja, jotka maksavat aikuisilta 10 euroa ja lapsilta 3 euroa. Alle 18-vuotiaat pääsevät lasten lipulla. Haluamme kirjoittaa ohjelman, joka kysyy käyttäjältä tämän iän ja tulostaa sitten lipun hinnan. Hintaa varten voimme määritellä muuttujan `hinta`, mutta sille asetettava arvo riippuu käyttäjän antamasta iästä. Jos ikä on pienempi kuin 18, pitää suorittaa käsky `hinta = 3` ja muussa tapauksessa käsky `hinta = 10`.

Tällaista tilannetta varten Python-kielessä on `if-else`-rakenne. Rakenteen yleinen muoto on

```
if ehto:
  käsky1
else:
  käsky2
```

Tämä suoritetaan seuraavasti: Ensin tutkitaan, onko `ehto` tosi vai ei. Jos ehto on tosi, suoritetaan `käsky1`. Jos ehto on epätosi, suoritetaan käsky `käsky2`. Toinen käskyistä `käsky1` ja `käsky2` jää siis aina suorittamatta. Tässä `ehto` on jokin lauseke, jonka totuusarvo voidaan tutkia, esimerkiksi `ika < 18`.

Lipun hinnan tulostava ohjelma voidaan siis kirjoittaa if-käskyn avulla seuraavasti:

```
def main():
    rivi = input("Kerro ikasi: ")
    ika = int(rivi)
    if ika < 18:
        hinta = 3
```

```
    else:
        hinta = 10
    print("Lipun hinta on", hinta, "euroa")

main()
```

Esimerkki ohjelman suorituksesta:

```
Kerro ikasi: 17
Lipun hinta on 3 euroa
```

Toinen esimerkki:

```
Kerro ikasi: 22
Lipun hinta on 10 euroa
```

If-käskyn ei ole pakko sisältää else-osaa. Jos else-osa puuttuu, if-rakenteeseen kuuluva käsky suoritetaan vain, jos ehto on tosi. Jos ehto on epätosi, siirrytään suoraan ohjelman seuraavaan käskyyn.

Käytetty ehto voi olla mikä tahansa lauseke, jonka arvo on `True` (tosi) tai `False` (epätosi). Tällaisia lausekkeita voi muodostaa esimerkiksi vertailuoperaattoreiden avulla. Python-kieli tarjoaa seuraavat vertailuoperaattorit:

`>`	suurempi kuin
`<`	pienempi kuin
`==`	yhtäsuuri kuin
`!=`	erisuuri kuin
`>=`	suurempi tai yhtäsuuri kuin
`<=`	pienempi tai yhtäsuuri kuin

Huomaa, että yhtäsuuri kuin -operaattori kirjoitetaan kahden yhtäsuuruusmerkin avulla. Yksi yhtäsuuruusmerkki tarkoittaa sijoitusta, jolla ei ole mitään tekemistä vertailun kanssa.

Lauseketta

```
ika == 15
```

suoritettaessa siis tutkitaan, onko muuttujan `ika` arvo yhtäsuuri kuin `15`. Jos on, niin lausekkeen arvo on `True`, jos taas ei ole, niin lausekkeen arvo on `False`.

Sen sijaan käskyä

```
ika = 15
```

suoritettaessa sijoitetaan muuttujan `ika` arvoksi `15`.

Lisäksi kannattaa huomata, että kahden desimaaliluvun yhtäsuuruutta ei yleensä kannata tutkia, koska pyöristysvirheet voivat aiheuttaa yllätyksiä.

Edellä olevassa esimerkissä oli täsmälleen yksi käsky, joka piti suorittaa, jos if-käskyn ehto oli tosi. Usein kuitenkin halutaan, että tässä tilanteessa suoritetaan useampi käsky.

Esimerkki: henkilön painoindeksi lasketaan siten, että paino (kiloissa) jaetaan pituuden (metreissä) neliöllä. Painoindeksin avulla voidaan päätellä, onko henkilö ali-, ylivai normaalipainoinen. Haluamme kirjoittaa ohjelman, joka pyytää käyttäjältä pituuden sekä painon ja tulostaa sitten henkilön painoindeksin.

Periaatteessa ohjelma on hyvin yksinkertainen: pyydetään ja luetaan käyttäjän paino ja pituus, lasketaan painoindeksi ja tulostetaan se. Ongelma on kuitenkin siinä, että jos käyttäjä antaa vahingossa pituudekseen 0, jakolaskussa syntyy virhetilanne — nollalla jako — ja ohjelma kaatuu antaen käyttäjälle kummallisen virheilmoituksen. Haluamme kuitenkin ohjelman kertovan käyttäjälle tässä tilanteessa selvästi, miksi painoindeksiä ei voi laskea. Ohjelman rakenne on seuraava:

```
pyydä ja lue käyttäjän paino ja pituus;
if pituus != 0:
  laske painoindeksi
  tulosta painoindeksi
else:
  ilmoita, että painoindeksiä ei voi laskea
```

Haluamme siis suorittaa kaksi eri käskyä, jos if-käskyn ehto on tosi. Tämä saadaan aikaiseksi käyttämällä sisennyksiä juuri siten kuin yllä olevasta esimerkistä näkyy. Kaikki ne käskyt, jotka on sisennetty sisemmälle tasolle kuin if-käskyn ehdon sisältävä rivi, katsotaan kuuluvaksi samaan if-käskyyn aina niitä seuraavaan sisentämättömään riviin saakka. Koska pituus ei voi olla myöskään negatiivinen, on ohjelmassa muutettu ehto `pituus != 0` muotoon `pituus > 0`. Näin se tulostaa virheilmoituksen aina silloin, kun pituus ei ole nollaa suurempi.

```
def main():
    rivi = input("Anna painosi kiloina: ")
    paino = float(rivi)
    rivi = input("Anna pituutesi metreina: ")
    pituus = float(rivi)
    if pituus > 0.0:
        painoindeksi = paino / (pituus * pituus)
        print("Painoindeksisi on", painoindeksi)
    else:
        print("Virheellinen pituus - painoindeksia ei voi laskea")

main()
```

Esimerkki ohjelman suorituksesta:

```
Anna painosi kiloina: 74.0
Anna pituutesi metreina: 1.81
Painoindeksisi on 22.5878330942
```

Toinen esimerkki:

```
Anna painosi kiloina: 57.5
Anna pituutesi metreina: 0.0
Virheellinen pituus - painoindeksia ei voi laskea
```

3.1.1 Loogiset operaattorit

Edellisessä painoindeksin laskevassa ohjelmassa tarkistettiin, että käyttäjän antama pituus ei ole nolla. Voi olla hyvinkin järkevää tarkistaa lisäksi, että pituus ei ole aivan liian suuri. Ei ole mitenkään epätavallista, että käyttäjä ei huomaa sitä, että pituus pitäisi antaa metreinä. Jos käyttäjä antaa vahingossa pituuden senttimetreinä, painoindeksistä tulee järjettömän pieni. Tämä on helppo estää lisäämällä ohjelmaan yksi tarkistus lisää. Jos esimerkiksi käyttäjän antama pituus on vähintään 3, voimme olla täysin varmoja siitä, että pituudessa on jokin virhe.

Aikaisemman ohjelman ehto *jos pituus on suurempi kuin 0* pitäisi siis muuttaa muotoon *jos pituus on suurempi kuin 0 ja pituus on pienempi kuin 3*. Ehto *pituus on pienempi kuin 3* on helppo kirjoittaa

```
pituus < 3.0
```

Nyt tarvitsemme kuitenkin ehdon, joka on tosi silloin, kun molemmat ehdoista *jos pituus on eri kuin 0* ja *pituus on pienempi kuin 3* ovat tosia ja epätosi aina, kun toinen tai molemmat näistä ehdoista ovat epätosia. Tällainen ehto saada aikaiseksi käyttämällä `and`- eli ja-operaattoria ehtojen välissä. Aikaisemman ohjelman ehdon

```
if pituus > 0:
```

sijasta ohjelmaan kirjoitetaankin nyt

```
if pituus > 0 and pituus < 3.0:
```

Muu ohjelma on täysin sama. Operaattori `and` on yksi *loogisista operaattoreista*. Tärkeitä loogisia operaattoreita on esitelty alla.

operaattori	nimi	merkitys
`and`	ja	tosi, jos molemmat lausekkeet tosia
`or`	tai	tosi, jos vähintään toinen lausekkeista tosi
`not`	ei	tosi, jos seuraava lauseke on epätosi

Esimerkiksi lausekkeen

```
x > 0 or y > 0
```

Arvo on `True` (tosi), jos joko muuttujan `x` arvo on nollaa suurempi tai muuttujan `y` arvo on nollaa suurempi tai niiden molempien arvo on suurempi kuin nolla.

Operaatioiden `and` ja `or` tapauksessa lasketaan ensin ensimmäisen (operaattoria edeltävän) lausekkeen arvo. Jos jo sen perusteella voidaan päätellä, että koko lausekkeen arvon on pakko olla `True` tai `False`, ei jälkimmäisen lausekkeen arvoa lasketa lainkaan. Käytännössä tämä tarkoittaa sitä, että jos `and`-operaattorin ensimmäisen operandin arvo on `False`, ei toisen operandin arvoa lasketa, vaan koko lausekkeen arvoksi tulee joka tapauksessa `False`. Vastaavasti, jos `or`-operaattorin ensimmäisen operandin arvo on `True`, ei jälkimmäisen operandin arvoa lasketa, vaan koko lausekeen arvoksi tulee aina `True`.

Tällä on merkitystä erityisesti niissä tapauksissa, joissa ensimmäisen lausekkeen totuusarvosta riippuu se, voidaanko jälkimmäisen lausekkeen totuusarvoa laskea lainkaan. Esimerkiksi lausekkeessa

```
(x != 0) and (10 / x > 0)
```

laskutoimitus `10 / x` aiheuttaa nollalla jakamisen ja koko ohjelman kaatumisen, jos muuttujan `x` arvo on nolla. Python-ohjelmaan yllä olevan lausekkeen voi kuitenkin kirjoittaa huoletta, koska ensin tutkitaan ensimmäisen operandin `(x != 0)` totuusarvo. Jos kyseinen lauseke on epätosi (eli muuttujan `x` arvo on 0), ei jälkimmäisen operandin totuusarvoa tutkita lainkaan ja jakolasku jää näin suorittamatta.

Luettelon viimeisellä operaattorilla `not` on vain yksi operandi. Esimerkiksi lausekkeen

```
not (x < 0)
```

arvo on `True`, jos lausekkeen `x < 0` arvo on `False`, ja päinvastoin.

Huomautus: Painoindeksiesimerkissä `and`-operaattorin käyttö ei ole välttämätöntä, sillä Python-ohjelmissa muuttujan arvon sijoittumista jollekin välille pystyy tutkimaan myös kirjoittamalla vertailuoperaattorit muuttujan molemmin puolin. Rivi

```
if pituus > 0 and pituus < 3.0:
```

voidaan siis korvata rivillä

```
if 0 < pituus < 3.0:
```

Näin voidaan tehdä kuitenkin vain silloin, kun tutkitaan muuttujan sijoittumista halutulle välille. Operaattori `and` on selvästi monikäyttöisempi, koska sen avulla voidaan yhdistää kaksi mitä tahansa ehtoa, joiden ei tarvitse edes sisältää samaa muuttujaa.

3.1.2 Sisäkkäisiä if-käskyjä

Myös if-käskyn sisällä voi olla toinen if-käsky. Tarkastellaan esimerkkiä, jossa painoindeksin laskun yhteydessä halutaan antaa varoitus, jos käyttäjä on alipainoinen (painoindeksi alle 19) tai ylipainoinen (painoindeksi vähintään 25). Ohjelma voidaan tällöin kirjoittaa seuraavasti:

```
def main():
    rivi = input("Anna painosi kiloina: ")
    paino = float(rivi)
    rivi = input("Anna pituutesi metreina: ")
    pituus = float(rivi)
    if pituus > 0.0 and pituus < 3.0:
        painoindeksi = paino / (pituus * pituus)
        print("Painoindeksisi on", painoindeksi)
        if painoindeksi < 19.0:
            print("Olet alipainoinen")
        if painoindeksi >= 25.0:
            print("Olet ylipainoinen")
    else:
        print("Virheellinen pituus - painoindeksia ei voi laskea")

main()
```

Esimerkissä sisemmät if-käskyt suoritetaan vain silloin, jos ulomman if-käskyn ehto on tosi.

Esimerkki ohjelman suorituksesta:

```
Anna painosi kiloina: 48.6
Anna pituutesi metreina: 1.6
Painoindeksisi on 18.984375
Olet alipainoinen
```

Joskus if-käskyjä joudutaan ketjuttamaan monta peräkkäin. Täydennetään painoindeksin laskevaa ohjelmaa siten, että se tulostaa lopuksi, onko käyttäjä alipainoinen (painoindeksi alle 19), normaalipainoinen (painoindeksi vähintään 19, mutta alle 25), lievästi ylipainoinen (painoindeksi vähintään 25, mutta alle 30), merkittävästi ylipainoinen (painoindeksi vähintään 30, mutta alle 35), vaikeasti ylipainoinen (painoindeksi vähintään 35, mutta alle 40) vai sairaalloisesti ylipainoinen (painoindeksi vähintään 40).

Ohjelmassa on siis useita ehtoja, joista vain yksi kerrallaan voi olla tosi. Tällöin ohjelman voi toki kirjoittaa käyttämällä monta sisäkkäistä if-käskyä, mutta huomattavasti selvemmän ohjelman saa aikaiseksi käyttämällä if - elif - else -rakennetta. Rakenteen yleinen muoto on seuraava:

```
if ehto1:
    käsky1
elif ehto2:
    käsky2
elif ehto3:
    käsky3

#lisää elif-kohtia ja niihin liittyviä käskyjä

else:
    käskyN
```

Sana elif on lyhenne sanoista else if. Rakenteessa tutkitaan ensin, onko `ehto1` tosi. Jos se on, suoritetaan `käsky1` eikä muita ehtoja tutkita lainkaan. Jos taas `ehto1` on epätosi, siirrytään tutkimaan järjestyksessä seuraavien ehtojen totuusarvoja, kunnes löydetään ensimmäinen tosi ehto ja suoritetaan sitä vastaava käsky. Jos mikään käskyistä ei ole tosi, suoritetaan else-kohdassa oleva käsky. Else-kohta voi myös puuttua, jolloin ei suoriteta mitään käskyä, jos mikään ehdoista ei ole tosi.

Painoindeksin laskeva ohjelma on nyt seuraavanlainen:

```
def main():
    rivi = input("Anna painosi kiloina: ")
    paino = float(rivi)
    rivi = input("Anna pituutesi metreina: ")
    pituus = float(rivi)
    if pituus > 0.0 and pituus < 3.0:
        painoindeksi = paino / (pituus * pituus)
        print("Painoindeksisi on", painoindeksi)
        if painoindeksi < 19.0:
            print("Olet alipainoinen")
        elif painoindeksi < 25.0:
            print("Painosi on normaali")
        elif painoindeksi < 30.0:
            print("Olet lievasti ylipainoinen")
        elif painoindeksi < 35.0:
            print("Olet merkittavasti ylipainoinen")
        elif painoindeksi < 40.0:
            print("Olet vaikeasti ylipainoinen")
        else:
            print("Olet sairaalloisesti ylipainoinen")
    else:
        print("Virheellinen pituus - painoindeksia ei voi laskea")

main()
```

Ehdoissa ei nyt tarvitse lainkaan tutkia, onko painoindeksi suurempi kuin jokin alaraja, sillä edellisten if- ja elif-kohtien ehdot pitävät huolen siitä, että jotain ehtoa tutkitaan vain silloin, jos painoindeksi on edellisten kohtien rajaa suurempi. Esimerkisi ehdon `painoindeksi < 35.0` totuusarvoa tutkitaan vain siinä tapauksessa, että kaikki edelliset ehdot ovat olleet epätosia, jolloin painoindeksin on pakko olla vähintään 30.

Esimerkki ohjelman suorituksesta:

```
Anna painosi kiloina: 49.0
Anna pituutesi metreina: 1.6
Painoindeksisi on 19.140625
Painosi on normaali
```

3.2 Toistokäsky

Useissa tietokoneohjelmissa samaa toimenpidettä pitää toistaa monta kertaa. Ajatellaan tasalyhenteistä lainaa varten tehtävää asuntolainalaskuria, joka pyytää ensin käyttäjältä lainasumman, laina-ajan ja korkoprosentin. Tämän jälkeen ohjelman halutaan tulostavan luettelon eri kuukausina maksettavista lainanhoitokuluista (lyhennys + korko yhteensä). Jos laina on tasalyhenteinen, maksettava lyhennys on joka kuukausi sama, mutta korkoerä vaihtelee sen mukaan, paljonko lainaa on jäljellä.

Ohjelman on siis laskettava jokaista kuukautta kohti sen hetkinen pääoma, tälle pääomalle kuukauden aikana kertynyt korko ja lyhennyserän ja kuukauden korkoerän summa. Jos laina-aika on 10 vuotta, pitää tämä laskutoimitus toistaa 120 kertaa. Olisi ikävää kirjoittaa samat käskyt ohjelmaan 120 kertaa peräkkäin. Lisäksi tämä lähestymistapa ei edes toimisi, jos käyttäjä saisi antaa vapaasti haluamansa laina-ajan, koska kirjoitettavien käskyjen määrä riippuisi aina laina-ajasta.

Tällaisia tilanteita varten useimmissa ohjelmointikielissä on toistokäskyjä. Toistokäskyn avulla voidaan ohjelmaa pyytää toistamaan jotain toista käskyä tai käskyjonoa niin kauan kuin jokin ohjelmoijan antama ehto on tosi.

Python-kielessä on kaksi erilaista toistokäskyä: `while` ja `for`. Seuraavaksi tarkastellaan näiden käskyjen toimintaa. Ennen kuin lähdemme katsomaan asuntolainalaskurin toteuttamista näiden käskyjen avulla, tutkimme käskyjen rakennetta yksinkertaisemman esimerkin kautta.

3.2.1 Toistokäsky while

While-käskyn yleinen rakenne on

```
while ehto:
    käsky
```

Kun ohjelmassa suoritetaan tällaistä käskyä, toimitaan seuraavasti: Ensin tarkastetaan, onko `ehto` tosi. Jos se on epätosi, siirrytään suoraan ohjelmassa eteenpäin while-rakenteen sisällä olevan käskyn (tai käskyjen) ohi sen jälkeen tulevaan kohtaan. Jos taas `ehto` on tosi, suoritetaan `käsky`. Tämän jälkeen tarkastetaan uudelleen, onko `ehto` tosi. Jos se on tosi, suoritetaan `käsky` jälleen. Sen jälkeen tarkastetaan taas, onko `ehto` tosi. Näin jatketaan, kunnes tullaan siihen tilanteeseen, että `ehto` on epätosi. Silloin lopetaan `while`-käskyn suoritus ja siirrytään ohjelmassa eteenpäin.

Toistettavia käskyjä voi olla joko yksi tai useita peräkkäin. Jälleen sisennysten avulla osoitetaan, mitkä käskyt kuuluvat toistettavien käskyjen sarjaan. Ehdon totuusarvo tarkistetaan kuitenkin vain ennen ensimmäisen käskyn suoritusta ja aina koko käskysarjan suorituksen jälkeen, ei siis käskysarjaan kuuluvien käskyjen välissä.

Tarkastallaan ensimmäisenä esimerkkinä ohjelmaa, joka pyytää käyttäjältä 10 kokonaislukua ja laskee niiden keskiarvon.

```
def main():
    print("Anna 10 kokonaislukua, lasken niiden keskiarvon")
    i = 0
    summa = 0
    while i < 10:
        rivi = input("Anna seuraava luku: ")
        luku = int(rivi)
        summa = summa + luku
        i = i + 1
    keskiarvo = summa / 10
    print("Niiden keskiarvo on", keskiarvo)

main()
```

Ohjelmassa käytetään muuttujaa `i` pitämään kirjaa siitä, montako lukua on jo luettu. Ennen toistokäskyn alkua `i`:lle annetaan arvo 0 ja jokaisella kierroksella `i`:n arvoa kasvatetaan yhdellä. Koska lukuja halutaan lukea 10 kappaletta, on toistokäskyn suorittamista jatkettava niin kauan kuin `i`:n arvo on pienempi kuin 10.

Tarvitaan myös muuttuja, johon kerätään jo luettujen lukujen summa. Tähän tarkoitukseen ohjelmassa käytetään muuttujaa `summa`, joka myöskin alustetaan nollaksi ennen toistokäskyn alkua. Toistokäskyn jokaisella kierroksella käyttäjältä luettu luku lisätään muuttujan `summa` vanhaan arvoon.

Toistokäskyn päätyttyä lasketaan keskiarvo jakamalla summa luettujen lukujen määrällä.

Esimerkki ohjelman suorituksesta:

```
Anna 10 kokonaislukua, lasken niiden keskiarvon
Anna seuraava luku: 12
Anna seuraava luku: 15
Anna seuraava luku: 42
Anna seuraava luku: 33
Anna seuraava luku: 76
Anna seuraava luku: 45
Anna seuraava luku: 22
Anna seuraava luku: 12
Anna seuraava luku: 34
Anna seuraava luku: 33
Niiden keskiarvo on 32.4
```

Edellisessä ohjelmassa luettavien lukujen määräksi on määrätty 10. Jos lukumäärää halutaan muuttaa, pitää muuttaa itse ohjelmaa. Ohjelma on kuitenkin kirjoitettu niin, että muutos pitää tehdä kolmeen eri paikkaan: käyttäjälle annettavaan alkuohjeeseen, toistokäskyn ehtoon ja keskiarvon laskevaan lausekkeeseen. On paljon parempi kirjoittaa ohjelma niin, että luettavien lukujen määrä tallennetaan muuttujaan, jota sitten käytetään ohjelmassa aina siellä, missä määrää tarvitaan. Tällöin lukumäärää on tarvittaessa helppo muuttaa. Muutos tarvitsee tehdä vain yhteen paikkaan, ja tuon paikan löytäminen on helppoa silloinkin, kun ohjelma on pitkä.

Tällaista muuttujaa kutsutaan vakioksi ja sen nimi kirjoitetaan isoilla kirjaimilla kertomaan siitä, että ohjelman suorituksen ei ole tarkoitus muuttaa tämän muuttujan arvoa sen jälkeen, kun ohjelmassa on annettu muuttujalle vakioalkuarvo. Muutettu ohjelma on esitetty alla.

```
def main():
    LKM = 10
    print("Anna", LKM, "kokonaislukua, lasken niiden keskiarvon")
    i = 0
    summa = 0
    while i < LKM:
        rivi = input("Anna seuraava luku: ")
        luku = int(rivi)
        summa = summa + luku
        i = i + 1
    keskiarvo = summa / LKM
    print("Niiden keskiarvo on", keskiarvo)

main()
```

Seuraava versio ohjelmasta on sellainen, että käyttäjä voi itse kertoa, kuinka monta lukua hän antaa. Ohjelma pyytää aluksi käyttäjältä annettavien lukujen lukumäärän ja tallentaa sen muuttujaan `lukujen_maara`. Sen jälkeen ohjelma pyytää toistokäskyn avulla näin monta lukua ja laskee lopuksi niiden keskiarvon. Koska lukujen määrä pyydetään käyttäjältä, on ohjelmaan lisätty tarkistus, jolla vältetään ohjelman kaatuminen jakolaskussa siinä tapauksessa, että lukujen määrä on 0.

```
def main():
    print("Lasken keskiarvon antamistasi kokonaisluvuista.")
    rivi = input("Anna lukujen maara: ")
    lukujen_maara = int(rivi)
    i = 0
    summa = 0
    while i < lukujen_maara:
        rivi = input("Anna seuraava luku: ")
        luku = int(rivi)
        summa = summa + luku
        i = i + 1
    if lukujen_maara > 0:
        keskiarvo = summa / lukujen_maara
        print("Niiden keskiarvo on", keskiarvo)
    else:
        print("Et antanut yhtaan lukua.")

main()
```

```
Lasken keskiarvon antamistasi kokonaisluvuista.
Anna lukujen maara: 5
Anna seuraava luku: 2
Anna seuraava luku: 4
Anna seuraava luku: 8
Anna seuraava luku: 8
Anna seuraava luku: 10
Niiden keskiarvo on 6.4
```

Tässäkin versiossa on kuitenkin se ongelma, että käyttäjän on etukäteen tiedettävä, montako lukua hän haluaa antaa. Jos lukuja on paljon, niiden laskeminen voi olla työlästä. Tällöin käyttäjän kannalta olisi parempi, että hän voisi antaa lukuja niin kauan kuin niitä riittää ja sitten kun luvut ovat lopussa, osoittaa jollain sopivalla arvolla lukujen loppuneen.

Jos esimerkiksi tiedetään, että kaikki annettavat luvut ovat suurempia tai yhtäsuuria kuin nolla, voi käyttäjä osoittaa lukujen loppuneen antamalla negatiivisen luvun. Ohjelma kirjoitetaan niin, että se lopettaa lukujen lukemisen ensimmäisen tällaisen luvun saatuaan. Viimeiseksi luettua negatiivista lukua ei tällöin oteta mukaan keskiarvoa laskettaessa. Ohjelma toteutetaan siten, että ensimmäinen luku luetaan jo ennen toistokäskyn alkua ja toistokäskyä jatketaan niin kauan kuin viimeiseksi luettu luku on vähintään nolla. Koska luettavien lukujen määrää ei tiedetä etukäteen, on se laskettava.

```
def main():
    print("Lasken keskiarvon antamistasi ei-negatiivisista kokonaisluvuista.")
    print("Lopeta negatiivisella luvulla.")
    lukujen_maara = 0
    summa = 0
    rivi = input("Anna ensimmainen luku: ")
    luku = int(rivi)
    while luku >= 0:
        lukujen_maara = lukujen_maara + 1
        summa = summa + luku
        rivi = input("Anna seuraava luku: ")
        luku = int(rivi)
    if lukujen_maara > 0:
        keskiarvo = summa / lukujen_maara
        print("Niiden keskiarvo on", keskiarvo)
    else:
        print("Et antanut yhtaan ei-negatiivista lukua.")

main()
```

Esimerkki ohjelman suorituksesta:

```
Lasken keskiarvon antamistasi ei-negatiivisista kokonaisluvuista.
Lopeta negatiivisella luvulla.
Anna ensimmainen luku: 24
```

```
Anna seuraava luku: 12
Anna seuraava luku: 30
Anna seuraava luku: 13
Anna seuraava luku: -5
Niiden keskiarvo on 19.75
```

3.2.2 Esimerkki: valintakäsky toistokäskyn sisällä

Toistokäskyn sisällä voi olla mitä tahansa käskyjä, esimerkiksi toinen toistokäsky tai if-käsky. Seuraavassa esimerkissä toistokäskyn sisällä on if-käsky.

Tarkastellaan seuraavaa tilannetta: käyttäjä syöttää ohjelmalle eri päivien lämpötiloja ja sademääriä. Ohjelman halutaan laskevan kaikkien lämpötilojen keskiarvon sekä sadepäivien lukumäärän. Sadepäiväksi katsotaan päivä, jonka sademäärä on ollut vähintään 1.0 mm. Ohjelma pyytää aluksi niiden päivien lukumäärän, joiden tiedot syötetään ohjelmalle. Sitten ohjelma lukee while-käskyn sisällä yhden päivän lämpötilan ja sademäärän. If-käskyn avulla tutkitaan, onko viimeksi luettu sademäärä vähintään `1.0`. Jos se on, sadepäivien lukumäärää kasvatetaan yhdellä. Toistokäskyn jälkeen ohjelma tulostaa lämpötilojen keskiarvon ja sadepäivien lukumäärän. Näitä käskyjä ei enää kirjoiteta toistokäskyn sisälle, koska tulostusten halutaan tapahtuvan vasta siinä vaiheessa, kun kaikki päivät on jo käyty läpi ja toistokäsky päättynyt.

```
def main():
    print("Ohjelma keraa saatilastoja.")
    syote = input("Monenko paivan tiedot haluat antaa? ")
    paiva_lkm = int(syote)
    i = 0
    sadepaivien_lkm = 0
    lamposumma = 0.0
    while i < paiva_lkm:
        syote = input("Anna paivan lampotila (C): ")
        lampotila = float(syote)
        lamposumma += lampotila
        syote = input("Anna saman paivan sademaara (mm): ")
        sademaara = float(syote)
        if sademaara >= 1.0:
            sadepaivien_lkm += 1
        i += 1
    if paiva_lkm == 0:
        print("Et antanut yhdenkaan paivan tietoja.")
    else:
        keskiarvo = lamposumma / paiva_lkm
        print("Lampotilojen keskiarvo on", keskiarvo, "C.")
        print("Sadepaivia oli", sadepaivien_lkm, "kpl.")

main()
```

Esimerkki ohjelman suorituksesta:

```
Ohjelma keraa saatilastoja.
Monenko paivan tiedot haluat antaa? 8
Anna paivan lampotila (C): 12.0
Anna saman paivan sademaara (mm): 3.0
Anna paivan lampotila (C): 18.5
Anna saman paivan sademaara (mm): 0.0
Anna paivan lampotila (C): 22.1
Anna saman paivan sademaara (mm): 0.0
Anna paivan lampotila (C): 11.5
Anna saman paivan sademaara (mm): 1.5
Anna paivan lampotila (C): 9.0
Anna saman paivan sademaara (mm): 0.5
Anna paivan lampotila (C): 16.0
Anna saman paivan sademaara (mm): 0.0
Anna paivan lampotila (C): 13.3
Anna saman paivan sademaara (mm): 4.5
Anna paivan lampotila (C): 18.0
Anna saman paivan sademaara (mm): 0.0
Lampotilojen keskiarvo on 15.05 C.
Sadepaivia oli 3 kpl.
```

3.2.3 Iteraatioesimerkki

Insinöörimatematiikassa törmätään usein tilanteeseen, jossa jokin yhtälö pitää ratkaista *iteroimalla*, koska yhtälölle ei ole suoraa ratkaisukaavaa. Iteroinnissa aluksi arvataan yhtälön juurelle (ratkaisulle) jokin arvo. Tämän jälkeen arvaus sijoitetaan sopivaan lausekkeeseen ja näin saatua arvoa käytetään parempana arvauksena. Se sijoitetaan edelleen lausekkeeseen, jolloin saadaan vielä parempi arvaus ja niin edelleen, kunnes arvaus on riittävän lähellä oikeaa arvoa. Aina arvaus ei parane joka kierroksella, mutta jos iterointimenetelmä on oikein valittu, yleensä saadut arvaukset vähitellen lähenevät oikeaa ratkaisua.

Tarkastellaan esimerkkinä käyttäjän antaman luvun `x` neliöjuuren laskemista Newtonin iteraatiolla. Neliöjuuren laskemista varten Python-kirjastoon on kirjoitettu valmiiksi oma funktio, joten esimerkkiohjelmamme ei ole sen vuoksi mitenkään välttämätön. Se on tässä valittu kuitenkin esimerkiksi iteroinnista yksinkertaisuutensa vuoksi.

Haluamme siis löytää luvun y siten, että $y = \sqrt{x}$. Toisin sanoen on etsittävä funktion $f(y) = y^2 - x$ nollakohta. Newtonin iteraatio perustuu yleisesti siihen, että jos meillä on funktion $f(y)$ nollakohdalle arvaus y_n, niin parempi arvaus y_{n+1} saadaan kaavalla

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

missä $f'(y)$ tarkoittaa funktion f derivaattaa muuttujan y suhteen. Kaava on saatu siitä, että tarkastellaan funktion $f(y)$ tangenttia kohdassa y_n ja on laskettu, missä kohdassa tangenttisuora leikkaa x-akselin.

Kun annettuun kaavaan sijoitetaan funktio $f(y) = y^2 - x$, ja sievennetään saatu lauseke, saadaan iterointikaavaksi

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$$

Tämän perusteella voidaan siis laskea muuttujan x neliöjuurelle parempi arvaus y_{n+1}, kun vanha arvaus on y_n. Iteraatiossa lähdetään liikkeelle mistä tahansa positiivisesta alkuarvauksesta, esimerkiksi ykkösestä.

Ohjelma tarvitsee vielä tiedon siitä, koska saatu arvaus on riittävän hyvä eli koska iterointi pitää lopettaa. Yksi yleinen menetelmä on tarkastella kahden perättäisen arvauksen välistä eroa ja lopettaa silloin, kun tämä ero on tarpeeksi pieni. Menetelmä toimii vain silloin, kun arvaus joka kierroksella lähenee oikeaa arvausta. Jos arvaukset voivat lähentyä jotain paikallista maksimia tai minimiä, joka ei ole kuitenkaan oikean ratkaisun lähellä, tällä kriteerillä saatu tulos voi olla kaukana oikeasta. Neliöjuuren hakemiseen tämä kriteeri kuitenkin sopii.

Kun tutkitaan, kuinka lähellä uusi arvaus on lähellä edellistä arvausta, on kätevää käyttää apuna itseisarvon laskemista. Etukäteen ei ole tietoa siitä, onko uusi arvaus suurempi vai pienempi kuin edellinen arvaus. Tämän vuoksi on helpointa tehdä niin, että lasketaan kahden peräkkäisen luvun välinen erotus ja otetaan siitä itseisarvo. Itseisarvon laskemiseen Pythonissa valmiina funktio `abs`. Luvun `arvo` itseisarvo voidaan laskea kirjoittamalla `abs(arvo)`.

Neliöjuuren laskeva ohjelma on alla. Koska neliöjuurta ei ole määritelty negatiivisille luvuille, ohjelma tarkistaa ensin, että käyttäjän antama luku ei ole negatiivinen, ja laskee neliöjuuren vain ei-negatiivisille luvuille.

```
def main():
    TARKKUUS  = 0.0001
    rivi = input("Minka luvun neliojuuri lasketaan? ")
    x = float(rivi)
    if x < 0:
        print("Neliojuurta ei ole maaritelty negatiivisille luvuille")
    else:
        arvaus = 1.0
        uusi_arvaus = 0.5 * (arvaus + x / arvaus)
        while abs(uusi_arvaus - arvaus) > TARKKUUS:
            arvaus = uusi_arvaus
            uusi_arvaus = 0.5 * (arvaus + x / arvaus)
        print("Luvun", x, "neliojuuri on", uusi_arvaus)

main()
```

Esimerkki ohjelman suorituksesta:

```
Minka luvun neliojuuri lasketaan? 57.0
Luvun 57.0 neliojuuri on 7.54983443527
```

Toinen esimerkki:

```
Minka luvun neliojuuri lasketaan? -9.0
Neliojuurta ei ole maaritelty negatiivisille luvuille
```

3.2.4 Toistokäsky for ja range-funktio

Toinen Python-kielessä käytettävä toistokäsky on `for`. Se on tarkoitettu tilanteisiin, jossa käydään läpi jonkin lukujonon tai rakenteen kaikki alkiot järjestyksessä. Perehdymme `for`-käskyn käyttämiseen eri rakenteiden yhteydessä myöhemmin, kun näitä rakenteita (lista, merkkijonot, sanakirja) esitellään tarkemmin. Tässä vaiheessa tarkastellaan kuitenkin lyhyesti `for`-käskyn käyttöä lukujonojen yhteydessä.

Python-kielessä saa generoitua kokonaislukujonon haluamaltaan väliltä funktion `range` avulla. Käsky `range(n)` generoi lukujonon, joka sisältää järjestyksessä kaikki kokonaisluvut nollasta `n-1`:een saakka. Jos nyt kirjoitetaan käsky

```
for i in range(n):
    kasky
```

tarkoittaa se: suorita `kasky` vuorotellen jokaiselle `i`:n arvolle, joka kuuluu `range(n)`:n generoimaan lukujonoon. Ensimmäisellä kierroksella siis `kasky` suoritetaan niin, että `i`:n arvo on 0, toisella kierroksella `i`:n arvo on 1, kolmannella 2 ja niin edelleen, kunnes viimeisellä kierroksella `i`:n arvo on `n-1`. Yhden käskyn sijaan toistettavana voi jälleen olla useamman käskyn jono, ja sisennysten avulla osoitetaan, mikä ohjelman osa kuuluu toistettavaan osaan. Esimerkiksi alla oleva ohjelma tulostaa 6:n kertotaulun (niin, että myös laskun `0 * 6` tulos on mukana):

```
def main():
    for i in range(11):
        tulo = i * 6
        print(i, "* 6 =", tulo)

main()
```

Ohjelman tulostus siis näyttää seuraavalta:

```
0 * 6 = 0
1 * 6 = 6
2 * 6 = 12
3 * 6 = 18
4 * 6 = 24
5 * 6 = 30
6 * 6 = 36
7 * 6 = 42
8 * 6 = 48
9 * 6 = 54
10 * 6 = 60
```

Joka kierroksella laskettavaa tuloa ei tarvitse välttämättä tallentaa muuttujaan, vaan kertolasku voidaan antaa myös suoraan tulostuskäskyssä:

```
def main():
    for i in range(11):
        print(i, "* 6 =", i * 6)

main()
```

Ohjelman tulostus on täsmälleen sama kuin edellä.

Huomautus: käsite lukujono ei ole täysin oikea, kun puhutaan `range`-käskyn suorituksesta. Oikea Pythonissa käytettävä termi on *iteraattori*. Toistokäskyn `for` ja käskyn `range` suorituksen ymmärtämiseen riittää kuitenkin se, että kuvittelee `range`-käskyn tekevän lukujonon, jota `for`-käskyssä käydään läpi.

Toistettaviin käskyihin voi kuulua mitä tahansa Python-kielen käskyjä, esimerkiksi `if`-käskyjä tai toisia toistokäskyjä. Seuraavassa esimerkkiohjelmassa käyttäjä antaa haluamansa määrän lukuja (määrä kysytään ensin käyttäjältä), ja ohjelma laskee annetuista luvuista positiivisten lukujen summan. Negatiivisia lukuja ei siis oteta summaan mukaan. Toistokäskyn sisällä käytetään `if`-käskyä tarkistamaan, onko viimeksi luettu luku positiivinen.

```
def main():
    rivi = input("Anna lukujen maara.\n")
    lkm = int(rivi)
    summa = 0
    print("Anna luvut yksi kerrallaan.")
    for i in range(lkm):
        rivi = input()
        luku = float(rivi)
        if luku > 0:
            summa += luku
    print("Positiivisten lukujen summa on", summa)


main()
```

Tässä ohjelmassa muuttujaa `i` ei ole käytetty toistokäskyn sisällä mitenkään. Sen avulla pidetään vain huoli siitä, että kierroksia suoritetaan täsmälleen käyttäjän antama määrä, joka on ohjelman alussa tallennettu `lkm`-muuttujaan. Summan tulostavaa riviä ohjelman lopussa ei ole enää sisennetty `for`-käskyn sisään, koska tulostus tehdään vasta toistokäskyn päätyttyä, ei joka kierroksella.

Funktio `range` luo siis oletusarvoisesti lukujonon, jonka ensimmäinen alkio on 0. Jos lukujonon halutaan alkavan jostain muusta luvusta, voidaan se tehdä antamalla `range`-funktiolle sulkujen sisällä ensin haluttu lähtöarvo, esimerkiksi `range(1,n)` tuottaa lukujonon, joka sisältää kokonaisluvut luvusta 1 lukuun `n-1` asti. Seuraava ohjelma tulostaa kuuden kertotaulun niin, että siinä ei ole mukana riviä, joka sisältää kertolaskun `0 * 6`.

```
def main():
    for i in range(1, 11):
        print(i, "* 6 =", i * 6)

main()
```

Oletusarvoisesti `range`-funktio luo lukujonon, joka sisältää kaikki halutun välin kokonaisluvut järjestyksessä. Jos kuitenkin halutaan, että lukujonossa peräkkäisten lukujen erotus on suurempi kuin 1, voidaan haluttu väli antaa `range`-funktiolle kolmantena tietona. `range(alaraja, ylaraja, vali)` tuottaa lukujonon, jossa esiintyvät `vali`:n välein olevat luvut alkaen `alaraja`sta ja päättyen viimeiseen lukuun, joka on pienempi kuin `ylaraja`. Esimerkiksi käsky

```
for i in range(5, 19, 3):
    print(i)
```

Tulostaa

```
5
8
11
14
17
```

3.2.5 Tulostuksen muotoilusta

Ennen kuin jatketaan toistokäskyistä ja niitä käyttävistä esimerkeistä, katsotaan vähän sitä, miten ohjelmoija voi vaikuttaa ohjelman tulostuksen muotoiluun.

Tähänastissa esimerkeissä desimaaliluvut on tulostettu sillä tarkkuudella, mitä Python-tulkki on oletuksena käyttänyt. Käytännössä tämä on tarkoittanut sitä, että tulostustetuissa luvuissa on usein ollut tarpeettoman monta desimaalia. Tulostuksesta tulisi selvästi siistimpi, jos desimaalipisteen jälkeen tulevien numeroiden määrää rajoitettaisiin. Tämä voidaan tehdä tulostuksen muotoilun avulla.

Muotoilun avulla voidaan myös määritellä, miten leveään kenttään kokonaisluvut ja merkkijonot tulostetaan. Kentällä tarkoitetaan sitä aluetta, joka tulostuksessa käytetään jonkin asian esittämiseen. Jos kentän leveydeksi on määritelty esimerkiksi 8, niin luvulle varataan tulostuksessa 8 merkin levyinen alue, vaikka itse luku veisikin vain 2 merkkiä. Joko ennen lukua tai sen jälkeen kirjoitetaan ylimääräisiä välilyöntejä. (Jos luku on niin suuri, että se ei mahdu sille varattuun kenttään, osaa luvusta ei kuitenkaan jätetä tulostamatta, vaan luvun annetaan käyttää enemmän tilaa kuin mitä sille on muotoilussa määrätty.) Kentän leveyden määrittelyä tarvitaan esimerkiksi silloin, kun halutaan tulostaa useita arvoja peräkkäisille riveille niin, että vastaavat arvot tulevat rivillä aina samaan kohtaan.

Kun halutaan vaikuttaa `print`-käskyllä tulostettavan rivin muotoiluun, kirjoitetaan ensin lainausmerkkien sisälle tulostettavalle riville tuleva vakioteksti. Tämän tekstin keskelle lisätään aaltosulut ja niiden sisälle *muotoilumääre* niihin paikkoihin, joihin tulostuksessa halutaan tulevan jonkin muuttujan arvo. Aaltosulut ja niiden sisällä oleva muotoilumääre osoittavat, mihin paikkaan muuttujan arvo tulee tulostettavassa tekstissä, ja lisäksi ne kertovat, miten tulostettava arvo muotoillaan, esimerkiksi

kuinka leveä kenttä arvolle varataan ja montako desimaalia desimaaliluvusta tulostetaan. Lainausmerkeissä olevan tekstin jälkeen kirjoitetaan piste, käsky `format`, ja sen jälkeen luetellaan sulkujen sisällä niiden muuttujien nimet, joiden arvot rivillä halutaan tulostaa. Riviä tulostettaessa aaltosulut ja niiden sisällä olevat muotoilumääreet korvataan luettelossa olevien muuttujien arvoilla niin, että ensimmäiset aaltosulut sisältöineen korvataan luettelon ensimmäisen muuttujan arvolla, toiset aaltosulut sisältöineen luettelon toisen muuttujan arvolla jne.

Muotoilumääre alkaa kaksoispisteellä `:`. Sen jälkeen siinä kerrotaan ensin arvolle varattavan kentän leveys kokonaislukuna. Leveys voidaan jättää myös pois, jolloin arvolle varataan sen tarvitsema tila (esimerkiksi 4-numeroiselle kokonaisluvulle 4 merkin levyinen kenttä). Desimaalilukuja koskevissa muotoilumääreissä tulee seuraavaksi piste ja tämän jälkeen toinen luku, joka kertoo, kuinka monta numeroa esitetään desimaalipisteen jälkeen. Viimeisenä muotoilumääressä on *määrittelykirjain*, joka kertoo, minkä tyyppinen arvo kohtaan tulee, esimerkiksi `d`, kun on kysymyksessä kokonaisluku, `s`, jos kysymyksessä on merkkijono ja `f`, `e` tai `g`, jos kysymyksessä on desimaaliluku. (Muotoilumääreissä voi olla myös muita osia tai muita kirjaimia erilaisia tyyppejä varten. Niitä ei esitetä tässä sen tarkemmin, mutta kiinnostunut lukija voi tutkia asiaa tarkemmin Pythonin omasta dokumentaatiosta.)

Seuraavaksi joitakin esimerkkejä muotoilumääreiden käytöstä ja niiden selityksiä. Esimerkit eivät ole kokonaisia ohjelmia, vaan Python-tulkille on annettu yksittäisiä käskyjä kappaleessa 2.1 kuvatulla tavalla.

```
>>> kertoja = 5
>>> kerrottava = 8
>>> tulos = kertoja * kerrottava
>>> print("{:3d} kertaa {:3d} on {:6d}".format(kertoja, kerrottava, tulos))
  5 kertaa   8 on     40
```

Tässä on kerrottu, että tulostettavaan tekstiin tulee ensin kokonaislukuarvo, jolle varataan 3 merkin levyinen kenttä, sen jälkeen teksti "kertaa ", sitten kokonaislukuarvo, jolle varataan 3 merkin levyinen kenttä, sen jälkeen teksti "on "ja lopuksi kokonaislukuarvo, jolle varataan kuuden merkin levyinen kenttä. Tulostuksessa ensimmäisen muotoilukoodin paikalle sijoitetaan muuttujan `kertoja` arvo, toisen koodin paikalle muuttujan `kerrottava` arvo ja kolmannen koodin paikalle muuttujan `tulos` arvo.

Huomautus: oikeastaan muotoilussa on kysymys siitä, että `print`-käskyssä annetulle merkkijonolle kutsutaan `format`-nimistä metodia, joka suorittaa tulostettavan merkkijonon muotoilun. Tätä ei kuitenkaan tarvitse vielä ymmärtää tässä vaiheessa, vaan asiaan palataan sitten, kun käsitellään olioita.

```
>>> enimi1 = "Matti"
>>> snimi1 = "Virtanen"
>>> palkka1 = 2800
>>> enimi2 = "Tiina-Maija"
>>> snimi2 = "Pomo"
>>> palkka2 = 11000
>>> print("{:15s} {:15s} {:5d} euroa".format(snimi1, enimi1, palkka1))
Virtanen        Matti            2800 euroa
```

```
>>> print("{:15s} {:15s} {:5d} euroa".format(snimi2, enimi2, palkka2))
Pomo            Tiina-Maija     11000 euroa
```

Tässä on muotoilukoodien avulla määritelty merkkijonojen ja kokonaisluvun kentän leveys. Näin peräkkäin tulostettavilla riveillä sarakkeet menevät päällekkäin, vaikka merkkijonojen pituus ja lukujen suuruus vaihtelevat. Oletuksen mukaisesti merkkijonoja sisältävät sarakkeet on tasattu niin, että niiden vasen reuna on samassa kohdassa, ja lukuja sisältävät sarakkeet niin, että niiden oikea reuna on samassa kohdassa. Jos halutaan säätää lukujen tulostus kentän vasempaan reunaan, lisätään muotoilumääreeseen `:`-merkin jälkeen merkki `<`, ja jos halutaan säätää merkkijonojen tulostus kentän oikeaan reunaan, lisätään muotoilumääreeseen `:`-merkin jälkeen merkki `>`. Alla on tästä esimerkkejä:

```
>>> print("{:>15s} {:>15s} {:<5d} euroa".format(snimi1, enimi1, palkka1))
       Virtanen           Matti 2800  euroa
>>> print("{:>15s} {:>15s} {:<5d} euroa".format(snimi2, enimi2, palkka2))
           Pomo     Tiina-Maija 11000 euroa
```

Seuraavaksi esimerkki desimaalilukujen tulostamisesta. Ensimmäisessä esimerkissä luvut on tulostettu määrittelykirjaimen `f` avulla. Esimerkissä tulostuskäsky jatkuu seuraavalle riville, koska se ei mahtunut siististi yhdelle riville. Seuraavan rivin alussa oleva `...` ei ole ohjelmoijan kirjoitusta, vaan Python-tulkki lisää sen automaattisesti seuraavan rivin alkuun silloin, kun edellistä riviä jatketaan.

```
>>> sivu = 4.5678945
>>> pinta_ala = sivu * sivu
>>> print("Sivu on {:5.2f} m ja pinta-ala {:8.2f} m2".format(sivu,
... pinta_ala))
Sivu on  4.57 m ja pinta-ala    20.87 m2
```

Jos muotoilumääreessä ei anneta kentän leveyttä, luvulle varataan tarpeellinen määrä tilaa. Muotoilumääreellä voi tällöinkin vaikuttaa esimerkiksi siihen, kuinka monta desimaalia desimaaliluvusta tulostetaan:

```
>>> print("Sivu on {:.2f} m ja pinta-ala {:.2f} m2".format(sivu,
... pinta_ala))
Sivu on 4.57 m ja pinta-ala 20.87 m2
```

Seuraavassa esimerkissä samat luvut on tulostettu määrittelykirjaimen `e` avulla. Tällöin desimaaliluku esitetään niin, että desimaalipisteen edessä on vain yksi numero ja lisäksi kerrotaan, millä kymmenen eksponentilla luku pitää kertoa. Esimerkiksi `5.87e+01` tarkoittaa $5.87 \cdot 10^1$. Ensimmäinen luku ei enää mahdu sille varattuun kenttään, mutta kuten esimerkistä nähdään, kentän leveyttä kasvatetaan tarvittaessa muotoilukoodissa annettua leveyttä suuremmaksi.

```
>>> print("Sivu on {:5.2e} m ja pinta-ala {:8.2e} m2".format(sivu,
... pinta_ala))
Sivu on 4.57e+00 m ja pinta-ala 2.09e+01 m2
```

Jos ei ole tietoa tulostettavan desimaaliluvun suuruusluokasta, voi käyttää määrittelykirjainta `g`. Tällöin luku tulostetaan ilman eksponenttiosaa, jos näin voidaan siististi tehdä ja eksponenttiosan kanssa, jos luku on itseisarvoltaan hyvin pieni tai suuri. Määrittelykirjainta `g` käytettäessä pisteen jälkeen tuleva luku ei kuitenkaan enää määrää desimaalipisteen jälkeen tulevien lukujen lukumäärää, vaan tulostuksessa esiintyvien merkitsevien numeroiden lukumäärää.

```
>>> pieni_luku = 1.2308e01
>>> suuri_luku = 14.5e20
>>> tulo = pieni_luku * suuri_luku
>>> print("Kun kerrotaan {:5.4g} ja {:8.4g}, saadaan {:10.4g}".format(
... pieni_luku, suuri_luku, tulo))
Kun kerrotaan 12.31 ja 1.45e+21, saadaan  1.785e+22
```

Tulostusta muotoiltaessa tulostettavan arvon ei tarvitse olla suoraan muuttujan arvona, vaan format-osan sulkujen sisällä voidaan antaa tulostettavaksi mitä tahansa lausekkeita, esimerkiksi

```
>>> korko = 2.5
>>> paaoma = 12000
>>> print("Saat korkoa {:.2f} eur".format(korko / 100.0 * paaoma))
Saat korkoa 300.00 eur
```

3.2.6 Asuntolainaesimerkki

Tarkastellaan alussa mainittua esimerkkiä. Halutaan kirjoittaa ohjelma, joka tulostaa tasalyhenteisen lainan maksusuunnitelman eli kuukausittain maksettavat korkoja lyhennyserät. Ohjelma pyytää käyttäjältä laina-ajan vuosina, lainasumman sekä korkoprosentin. Yksinkertaisuuden vuoksi oletetaan, että korkoprosentti pysyy samana koko laina-ajan.

Kuukausittainen lyhennyserä saadaan jakamalla lainasumma laina-aikaan kuuluvien kuukausien määrällä. Lyhennys pysyy koko laina-ajan samana. Kuukausittain maksettava korkoerä taas vaihtelee sen mukaan, kuinka paljon lainan pääomaa on vielä jäljellä. Jokaista kuukautta kohti lasketaan jäljellä olevan pääoman aiheuttama korkoerä ja tämän jälkeen vähennetään lyhennyserä pääomasta. Ohjelma on kirjoitettu alla `while`-käskyn avulla. Yksinkertaisuuden vuoksi pyöristysvirheiden mahdollisuutta ei ole otettu huomioon, vaan yhteensä takaisin maksettava lainapääoma voi poiketa muutamalla sentillä alkuperäisestä lainasummasta.

```
def main():
    print("Ohjelma laskee asuntolainan kuukausierat.")
    rivi = input("Anna lainasumma: ")
    lainasumma = float(rivi)
    rivi = input("Anna laina-aika vuosina: ")
    laina_aika = int(rivi)
    if laina_aika < 1:
        print("Liian lyhyt laina-aika")
```

```
    else:
        kk_lkm = 12 * laina_aika
        rivi = input("Anna korkoprosentti: ")
        korko = float(rivi)
        lyhennys = lainasumma / kk_lkm
        paaoma = lainasumma
        i = 0
        print("    lyhennys    korko   yhteensa")
        while i < kk_lkm:
            i = i + 1
            korkoera = korko / 1200.0 * paaoma
            paaoma = paaoma - lyhennys
            kuukausiera = korkoera + lyhennys
            print("{:2d}. {:8.2f} {:8.2f} {:8.2f}".format(
                   i, lyhennys, korkoera, kuukausiera))

main()
```

Esimerkki ohjelman suorituksesta:

```
Ohjelma laskee asuntolainan kuukausierat.
Anna lainasumma: 20000
Anna laina-aika vuosina: 2
Anna korkoprosentti: 5.5
    lyhennys    korko   yhteensa
 1.   833.33    91.67   925.00
 2.   833.33    87.85   921.18
 3.   833.33    84.03   917.36
 4.   833.33    80.21   913.54
 5.   833.33    76.39   909.72
 6.   833.33    72.57   905.90
 7.   833.33    68.75   902.08
 8.   833.33    64.93   898.26
 9.   833.33    61.11   894.44
10.   833.33    57.29   890.62
11.   833.33    53.47   886.81
12.   833.33    49.65   882.99
13.   833.33    45.83   879.17
14.   833.33    42.01   875.35
15.   833.33    38.19   871.53
16.   833.33    34.37   867.71
17.   833.33    30.56   863.89
18.   833.33    26.74   860.07
19.   833.33    22.92   856.25
20.   833.33    19.10   852.43
21.   833.33    15.28   848.61
22.   833.33    11.46   844.79
23.   833.33     7.64   840.97
24.   833.33     3.82   837.15
```

Toinen esimerkki:

```
Ohjelma laskee asuntolainan kuukausierat.
Anna lainasumma: 100000
Anna laina-aika vuosina: 0
Liian lyhyt laina-aika
```

Sama ohjelma voidaan kirjoittaa `for`-käskyn avulla seuraavasti:

```
def main():
    print("Ohjelma laskee asuntolainan kuukausierat.")
    rivi = input("Anna lainasumma: ")
    lainasumma = float(rivi)
    rivi = input("Anna laina-aika vuosina: ")
    laina_aika = int(rivi)
    if laina_aika < 1:
        print("Liian lyhyt laina-aika")
    else:
        kk_lkm = 12 * laina_aika
        rivi = input("Anna korkoprosentti: ")
        korko = float(rivi)
        lyhennys = lainasumma / kk_lkm
        paaoma = lainasumma
        print("    lyhennys    korko   yhteensa")
        for i in range(1, kk_lkm + 1):
            korkoera = korko / 1200.0 * paaoma
            paaoma = paaoma - lyhennys
            kuukausiera = korkoera + lyhennys
            print("{:2d}. {:8.2f} {:8.2f} {:8.2f}".format(
                   i, lyhennys, korkoera, kuukausiera))

main()
```

Tässä ohjelmassa silmukkalaskuria `i` ei siis alusteta eikä sen arvoa kasvateta joka kierroksella, koska `for`-käsky huolehtii näistä toimenpiteistä yhdessä `range`-funktion kanssa.

Luku 4

Funktiot

Todellisessa elämässä tarvittavat ohjelmat ovat usein tähän asti esitettyjä ohjelmia selvästi pitempiä, usein tuhansien ja jopa kymmenien tuhansien rivien pituisia. Ohjelman rakenteen selkeyttämiseksi ohjelma jaetaan tällöin pienempiin osiin, funktioihin. Funktio on ohjelman osa, jolle on annettu oma nimi. Tyypillisesti funktio suorittaa jonkin tehtävän.

Kun halutaan, että funktio suoritetaan, kirjoitetaan ohjelmaan käsky, joka *kutsuu* funktiota. Funktion kutsu voi olla joko pääohjelmassa tai sitten toisen funktion sisällä.

4.1 Yksinkertaisia esimerkkejä

Tarkastellaan seuraavaa esimerkkiä: halutaan kirjoittaa ohjelma, joka tulostaa kuvaruudulle alla olevan mallin mukaisen kuvion.

```
  *
 ***
*****
  *
 ***
*****
  *
 ***
*****
```

Havaitaan, että kuvio muodostuu kolmesta kolmiosta. Ohjelma voidaan kirjoittaa nyt niin, että kirjoitetaan funktio `tulosta_kolmio`, joka tulostaa yhden kolmion ja suoritetaan tämä funktio kolme kertaa. Funktio määritellään samaan tapaan kuin pääohjelmakin. Määrittely aloitetaan sanalla `def` ja funktion nimellä, jonka jälkeen tulevat sulut ja kaksoispiste. Seuraavalta riviltä alkavat sitten funktioon kuuluvat käskyt. Jälleen sisennysten avulla osoitetaan, mitkä käskyt kuuluvat tämän funktion sisälle.

Pääohjelman tehtäväksi jää kutsua funktiota `tulosta_kolmio` kolme kertaa. Funktiota kutsutaan kirjoittamalla sen nimi ja nimen perään sulut. Funktion kutsu saa

aikaan sen, että pääohjelman suoritus tavallaan keskeytetään ja siirrytään suorittamaan funktioon kuuluvia käskyjä. Kun funktio suoritus päättyy, siirrytään takaisin pääohjelmaan ja jatketaan siitä kohdasta, jossa pääohjelman suoritus keskeytyi.

Ohjelma on kokonaisuudessaan seuraava:

```
def tulosta_kolmio():
    print("  *  ")
    print(" *** ")
    print("*****")


def main():
    tulosta_kolmio()
    tulosta_kolmio()
    tulosta_kolmio()


main()
```

Ohjelma tulostaa edellä esitetyn kuvion. Koska pääohjelmassa suoritetaan sama käsky (funktion `tulosta_kolmio` kutsu) kolme kertaa peräkkäin, voidaan kutsu kirjoittaa myös toistokäskyn sisään. Jos vielä kutsukertojen määrä määritellään vakioksi, on sitä helppo muuttaa myöhemmin. Muutettu pääohjelma on seuraavan näköinen:

```
def main():
    KERRAT = 3
    for i in range(KERRAT):
        tulosta_kolmio()
```

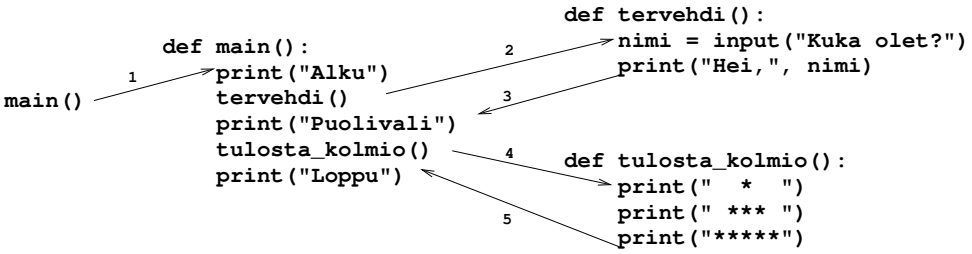
Ohjelman muut osat jäävät ennalleen. Toistokäskyn sisällä oleva käsky (funktion `tulosta_kolmio` kutsu) ei riipu mitenkään toistokäskyssä käytetystä muuttujasta `i`, vaan toistettava käsky suoritetaan joka kerta täysin samalla tavalla. Muuttujaa `i` voidaan kuitenkin käyttää tällä tavalla pitämään huolta siitä, että toistokäskyä suoritetaan juuri haluttu määrä kierroksia.

Tarkastellaan seuraavaa esimerkkiohjelmaa, jonka tarkoituksena on lähinnä selventää sitä, miten ohjelman suoritus ja funktiokutsut etenevät.

```
def tervehdi():
    nimi = input("Kuka olet? ")
    print("Hei,", nimi)


def tulosta_kolmio():
    print("  *  ")
    print(" *** ")
    print("*****")
```

```
def main():
    print("Alku")
    tervehdi()
    print("Puolivali")
    tulosta_kolmio()
    print("Loppu")


main()
```

Ohjelmassa ensimmäiseksi suoritettava käsky on sen lopussa oleva pääohjelman kutsu `main()`, joka saa aikaan sen, että Python-tulkki lähtee suorittamaan pääohjelman määrittelyyn kirjoitettua koodia. Siinä tulostetaan ensin kuvaruudulle yksi rivi. Sen jälkeen ohjelmassa kutsutaan `tervehdi`-funktiota eli lähdetään suorittamaan ko. funktion määrittelyyn kirjoitettuja käskyjä. Kun funktion suoritus päättyy, palataan takaisin pääohjelmaan, jossa on seuraavana vuorossa jälleen yhden rivin tulostus. Sen jälkeen siirrytään suorittamaan funktioon `tulosta_kolmio` määriteltyjä käskyjä. Kun funktion suoritus on lopussa, palataan pääohjelmaan ja suoritetaan siinä viimeisenä oleva tulostuskäsky. Ohjelman osien suoritusjärjestystä on kuvattu vielä alla olevassa kuvassa.



Näin ollen ohjelman tulostus voi olla esimerkiksi seuraava:

```
Alku
Kuka olet? Pekka
Hei, Pekka
Puolivali
  *
 ***
*****
Loppu
```

4.2 Parametrit

Tarkastellaan seuraavaa esimerkkiä: haluamme laatia ohjelman, joka kertoo, miten sijoituksen arvo kasvaa vuosittain, kun ohjelmalle annetaan sijoitettava arvo, sijoitusaika ja oletettu vuosittainen tuottoprosentti (oletetaan, että tuottoprosentti pysyy samana koko sijoitusajan). Haluamme siis, että ohjelma tulostaa luettelon, jossa on sijoituksen arvo kunkin vuoden päätyttyä.

Ohjelma koostuu periaatteessa kahdesta eri osasta: Ensin pyydetään käyttäjältä sijoituksen tiedot. Tämän jälkeen lasketaan sijoituksen arvon kehittyminen ja tulostetaan arvot eri vuosina. Näistä jälkimmäinen sopii hyvin kirjoitettavaksi omaksi funktiokseen, joka voi olla nimeltään esimerkiksi `laske_arvokehitys`. Tämä funktio tarvitsee kuitenkin lähtötietoinaan tiedon sijoituksen alkuperäisestä arvosta, sijoitusajasta ja tuottoprosentista. Nämä tiedot voidaan välittää funktiolle *parametrien* avulla.

Parametri on tapa välittää funktiota kutsuvalta ohjelman osalta tietoa funktiolle. Funktion käyttämät parametrit on kerrottu funktion määrittelyn otsikossa sulkujen sisässä. Esimerkiksi funktion `laske_arvokehitys` määrittelyn otsikko voisi tällöin olla:

```
def laske_arvokehitys(paaoma, aika, tuottoprosentti):
```

Parametreja `paaoma`, `aika` ja `tuottoprosentti` voi käyttää funktion sisällä samalla tavalla kuin mitä tahansa muuttujia. Se, mikä arvo näille parametreille annetaan funktion suorituksen alussa, määrätään funktiota kutsuttaessa funktion nimen perässä olevien sulkujen sisällä. Jos esimerkiksi funktiota kutsutaan

```
laske_arvokehitys(5000.0, 5, 4.5)
```

lähdetään funktiota `laske_arvokehitys` suorittamaan niin, että parametrin `paaoma` arvo on `5000.0`, parametrin `aika` arvo `5` ja parametrin `tuottoprosentti` arvo `4.5`. Kutsussa olevat arvot annetaan siis parametrien arvoksi samassa järjestyksessä kuin ne on funktion määrittelyn otsikossa ja funktion kutsussa lueteltu. Määrittelyn otsikossa ensimmäisenä oleva parametri saa arvokseen kutsussa sulkujen sisällä ensimmäisenä annetun arvon ja niin edelleen.

Funktion kutsussa esiintyvien parametrien arvojen ei tarvitse olla välttämättä vakioita, vaan ne voivat olla mitä tahansa lausekkeita, joiden arvo voidaan laskea, esimerkiksi

```
laske_arvokehitys(500.0 + 600, 2 + 1, 9.0 / 2)
```

tai

```
laske_arvokehitys(muuttuja1, muuttuja2, muuttuja3)
```

missä `muuttuja1`, `muuttuja2` ja `muuttuja3` ovat muuttujia, joille on annettu arvo ennen funktion kutsumista.

Samaa funktiota voidaan kutsua myös esimerkiksi samasta pääohjelmasta useita kertoja eri parametrien arvoilla.

Seuraavaksi on esitetty sijoituksen arvokehityksen laskeva ohjelma.

```
def laske_arvokehitys(paaoma, aika, tuottoprosentti):
    print("Sijoituksen arvo vuoden lopussa:")
    print("vuosi  arvo")
    for i in range(1, aika + 1):
        paaoma = paaoma * (1.0 + tuottoprosentti / 100.0)
        print("{:2d}. {:10.2f} euroa".format(i, paaoma))


def main():
    print("Ohjelma laskee sijoituksen arvon kehittymisen vuosittain")
    rivi = input("Anna sijoitettava summa (euroa): ")
    summa = float(rivi)
    rivi = input("Anna sijoitusaika (vuotta): ")
    vuodet = int(rivi)
    rivi = input("Anna odotettu tuottoprosentti: ")
    tuotto = float(rivi)
    laske_arvokehitys(summa, vuodet, tuotto)


main()
```

Esimerkki ohjelman suorituksesta:

```
Ohjelma laskee sijoituksen arvon kehittymisen vuosittain
Anna sijoitettava summa (euroa): 20000
Anna sijoitusaika (vuotta): 5
Anna odotettu tuottoprosentti: 4.5
Sijoituksen arvo vuoden lopussa:
vuosi  arvo
 1.   20900.00 euroa
 2.   21840.50 euroa
 3.   22823.32 euroa
 4.   23850.37 euroa
 5.   24923.64 euroa
```

Esimerkkitapauksessa siis pääohjelmassa muuttujan `summa` arvoksi tulee `20000.0`, muuttujan `vuodet` arvoksi `5` ja muuttujan `tuotto` arvoksi `4.5`. Kun funktion `laske_arvokehitys` suoritus aloitetaan, sen parametri `paaoma` saa arvokseen pääohjelman muuttujan `summa` arvon, parametri `aika` pääohjelman muuttujan `vuodet` arvon ja parametri `tuottoprosentti` pääohjelman muuttujan `tuotto` arvon. Tämä seuraa siitä, että pääohjelmassa funktion kutsussa näiden muuttujien nimet on kirjoitettu sulkujen sisään tässä järjestyksessä.

Myös pääohjelman se osa, joka pyytää lähtötiedot käyttäjältä, sopisi hyvin omaksi funktiokseen. Tässä vaiheessa opituilla tiedoilla ei ole kuitenkaan vielä keinoja saada funktion käyttäjältä lukemia arvoja pääohjelman ja sitä kautta toisen funktion käyttöön.

4.3 Arvon palauttavat funktiot

Tarkastellaan vähän toisenlaista versiota sijoituksen arvon kehittymisen laskevasta ohjelmasta: tällä kerralla käyttäjä haluaa vertailla useaa eri sijoitusmahdollisuutta, joilla on eri vuosituotto. Käyttäjällä on mielessään sijoitettava pääoma ja sijoitusaika ja hän haluaa vertailla pääoman arvoa sijoitusajan jälkeen eri sijoitustavoissa.

Ohjelmassa käyttäjä antaa oletetun vuosituoton käyttötilille, sijoitustilille, korkorahastosijoitukselle ja osakerahastosijoitukselle. Ohjelma laskee käyttäjän antaman pääoman arvon annetun sijoitusajan jälkeen ja tulostaa pääoman ennustetun arvon eri sijoitustavoille.

Tässä ohjelmassa pääoman arvo sijoitusajan jälkeen pitää laskea neljä eri kertaa. Jokaisella kerralla käytetään samaa kaavaa, mutta käytetty vuosituottoprosentti vaihtelee. On siis järkevää tehdä oma funktio, joka saa parametrina pääoman, sijoitusajan ja tuottoprosentin ja laskee pääoman arvon sijoitusajan jälkeen.

Tällä kerralla ei kuitenkaan haluta, että arvon laskeva funktio tulostaa käyttäjälle sijoituksen arvon, vaan tulostus halutaan tehdä pääohjelmassa. Näin funktiota voidaan käyttää monipuolisemmin, kun sen laskeman arvon tulostus on erotettu itse laskemisesta. Funktion laskema arvo voidaan tulostaa ohjelmassa vasta selvästi myöhemmin kuin itse laskeminen tapahtui tai laskettua arvoa voidaan käyttää ohjelmassa myöhemmin hyväksi.

Määritellään siis funktio `laske_arvo`, joka laskee sijoituksen arvon annetun sijoitusajan jälkeen. Alkuperäinen pääoma, sijoitusaika ja vuosittainen tuottoprosentti annetaan tälle funktiolle parametreina. Nyt tarvitaan kuitenkin jokin keino, jolla pääohjelma saa tietoonsa funktion laskeman arvon, jotta tätä arvoa voitaisiin käyttää pääohjelmassa. Tähän tarkoitukseen käytetään *arvon palauttamista.*

Funktio saadaan palauttamaan jokin arvo kirjoittamalla

```
return lauseke
```

missä `lauseke` voi olla mikä tahansa lauseke, jonka arvo voidaan laskea, myös pelkkä muuttujan nimi. Arvon palauttaminen tarkoittaa sitä, että lasketaan lausekkeen arvo, päätetään funktion suoritus ja palataan takaisin sinne kohtaan, missä funktiota kutsuttiin. Tässä kohdassa funktion palauttama lausekkeen arvo on käytettävissä: se voidaan esimerkiksi ottaa talteen johonkin muuttujaan sijoituskäskyn avulla tai tulostaa suoraan `print`-käskyssä. Tätä selvennetään alla olevassa esimerkissä.

Funktio `laske_arvo` voidaan kirjoittaa esimerkiksi seuraavasti:

```
def laske_arvo(paaoma, aika, vuosituotto):
    loppuarvo = (1.0 + vuosituotto/100.0) ** aika * paaoma
    return loppuarvo
```

Tässä on ensin laskettu sijoituksen arvo annettujen vuosien jälkeen, sijoitettu näin saatu arvo muuttujaan `loppuarvo` ja palautettu muuttujan `loppuarvo` arvo. Palautettu arvo saadaan käyttöön siellä, missä funktiota kutsutaan esimerkiksi sijoittamalla arvo johonkin muuttujaan seuraavasti:

```
sijoituksen_arvo = laske_arvo(10000, 5, 4.0)
```

Tässä on kutsuttu funktiota `laske_arvo` suluissa annetuilla parametrien arvoilla. Funktion suorittamisen jälkeen sen palauttama arvo on sijoitettu muuttujaan `sijoituksen_arvo`. Tämän jälkeen muuttujaa `sijoituksen_arvo` voi käyttää ohjelmassa samaan tapaan kuin mitä tahansa muuta muuttujaa. Sijoituskäsky siis suoritetaan ihan samalla tavalla kuin aikaisemmissa esimerkeissä esitetyt sijoituskäskyt: Ensin lasketaan sijoituskäskyn oikealla puolella olevan lausekkeen arvo, mikä funktiokutsun tapauksessa tarkoittaa kutsutun funktion suorittamista suluissa annetuilla parametrien arvoilla. Tämän jälkeen lausekkeen arvo eli funktion palauttama arvo annetaan arvoksi sijoituskäskyn vasemmalla puolella olevalle muuttujalle.

Funktion palauttamaa arvoa voidaan käyttää myös suoraan ilman että arvo tallennetaan minkään muuttujan arvoksi, esimerkiksi seuraavasti

```
print("Arvo lopuksi", laske_arvo(10000, 5, 4.0))
```

Tässä kutsutaan ensin funktiota `laske_arvo`, suoritetaan funktio ja sen jälkeen tulostetaan funktion palauttama arvo yhdessä muun `print`-käskyssä annetun tekstin kanssa. Tässä tapauksessa funktion palauttamaa arvoa ei ole kuitenkaan tallennettu minkään muuttujan arvoksi, joten arvoa ei voida käyttää ohjelmassa enää tämän jälkeen muuten kuin kutsumalla funktiota uudestaan.

Myös funktio `laske_arvo` voidaan kirjoittaa toisella tavalla kuin edellä on esitetty. Kuten kerrottiin, `return`-käskyssä voi palautettavana arvona olla mikä tahansa lauseke, jonka arvo voidaan laskea. Niinpä ei ole mikään pakko tallentaa funktion laskemaa arvoa johonkin muuttujaan ennen `return`-käskyn suorittamista, vaan laskettava lauseke voidaan kirjoittaa suoraan `return`-käskyyn seuraavan esimerkin mukaisesti:

```
def laske_arvo2(paaoma, aika, vuosituotto):
    return (1.0 + vuosituotto/100.0) ** aika * paaoma
```

Tässä lasketaan ensin `return`-käskyssä olevan lausekkeen arvo ja palautetaan se samalla kun lopetetaan funktion suoritus. Funktiota voidaan kutsua ja sen palauttamaa arvoa käyttää ihan samalla tavalla kuin edellisessä esimerkissä.

Seuraavaksi esimerkkiohjelma, joka pyytää käyttäjältä sijoitettavan pääoman, sijoitusajan vuosina sekä oletetut tuottoprosentit eri sijoitusmuodoille. Ohjelma tulostaa sijoitusten arvon sijoitusajan jälkeen.

```
def laske_arvo(paaoma, aika, vuosituotto):
    loppuarvo = (1.0 + vuosituotto/100.0) ** aika * paaoma
    return loppuarvo


def main():
    rivi = input("Anna sijoitettava paaoma: ")
    rahamaara = float(rivi)
```

```
    rivi = input("Anna sijoitusaika vuosina: ")
    vuodet = int(rivi)
    rivi = input("Anna kayttotilin korko: ")
    korko = float(rivi)
    kayttotilisijoitus = laske_arvo(rahamaara, vuodet, korko)
    rivi = input("Anna sijoitustilin korko: ")
    korko = float(rivi)
    sijoitustilisijoitus = laske_arvo(rahamaara, vuodet, korko)
    rivi = input("Anna korkorahaston vuosituotto: ")
    korko = float(rivi)
    korkorahastosijoitus = laske_arvo(rahamaara, vuodet, korko)
    rivi = input("Anna osakerahaston vuosituotto: ")
    korko = float(rivi)
    osakerahastosijoitus = laske_arvo(rahamaara, vuodet, korko)
    print("Sijoitusten arvo", vuodet, "vuoden paasta:")
    print("Kayttotili:", kayttotilisijoitus, "euroa")
    print("Sijoitustili:", sijoitustilisijoitus, "euroa")
    print("Korkorahasto:", korkorahastosijoitus, "euroa")
    print("Osakerahasto", osakerahastosijoitus, "euroa")


main()
```

Python-ohjelmissa yksi funktio voi myös palauttaa kerralla useamman arvon, jotka voidaan ottaa talteen samanaikaisella sijoituksella. Tarkastellaan esimerkkinä seuraavaa ongelmaa: Internetissä ja urheilulehdissä on hyvin paljon erilaisia harjoitusohjelmia juoksuharrastuksen aloittamiseen ja kehittämiseen. Ongelma on kuitenkin se, että englanninkielisillä sivuilla ja lehdissä on yleensä annettu ohjeelliset juoksunopeudet juostavaa mailia kohti, kun taas suomalainen haluaisi tarkastella nopeutta kilometriä kohti. Kirjoitetaan ohjelma, jonka avulla käyttäjä voi helposti muuttaa esimerkiksi harjoitusohjelmassa annetun nopeusohjeen 8 min 5 s – 9 min 30 s / maili nopeudeksi kilometriä kohti.

Nopeusmuunnosta varten kirjoitetaan funktio `muuta_km_vauhdiksi`, joka ensin muuntaa alkuperäisen nopeuden sekunneiksi mailia kohti ja muuttaa tämän jakolaskun avulla sekunneiksi kilometriä kohti. Sitten funktio muuttaa kokonaislukujen jakolaskun ja jakojäännöksen avulla sekunnit minuuteiksi ja sekunneiksi. Tämän jälkeen funktion halutaan palauttavan arvonaan sekä kilometrivauhdin minuutit että sekunnit. Tämä voidaan tehdä kirjoittamalla `return`-käskyyn molemmat palautettavat arvot pilkulla erotettuna.

```
return km_minuutit, km_sekunnit
```

Funktion palauttamat arvot voidaan ottaa talteen esimerkiksi muuttujiin `ala_min` ja `ala_s` seuraavalla sijoituskäskyllä:

```
ala_min, ala_s = muuta_km_vauhdiksi(maili_minuutit, maili_sekunnit)
```

Koko muunnoksen tekevä ohjelma on esitetty alla.

```
def muuta_km_vauhdiksi(minuutit, sekunnit):
    MAILIKERROIN = 1.609344
    km_vauhti = int ((minuutit * 60 + sekunnit) / MAILIKERROIN)
    km_minuutit = km_vauhti // 60
    km_sekunnit = km_vauhti % 60
    return km_minuutit, km_sekunnit


def main():
    rivi = input("Anna nopeuden alarajan minuutit: ")
    maili_min = int(rivi)
    rivi = input("Anna nopeuden alarajan sekunnit: ")
    maili_sek = int(rivi)
    ala_min, ala_s = muuta_km_vauhdiksi(maili_min, maili_sek)
    rivi = input("Anna nopeuden ylarajan minuutit: ")
    maili_min = int(rivi)
    rivi = input("Anna nopeuden ylarajan sekunnit: ")
    maili_sek = int(rivi)
    yla_min, yla_s = muuta_km_vauhdiksi(maili_min, maili_sek)
    print("Suositeltu nopeus on", ala_min, "min", ala_s, "s / km -",
          yla_min, "min", yla_s, "s / km")


main()
```

Esimerkki ohjelman käyttämisestä:

```
Anna nopeuden alarajan minuutit: 8
Anna nopeuden alarajan sekunnit: 5
Anna nopeuden ylarajan minuutit: 9
Anna nopeuden ylarajan sekunnit: 30
Suositeltu nopeus on 5 min 1 s / km - 5 min 54 s / km
```

4.3.1 Sisäkkäiset funktiokutsut

Kuten aikaisemmin on todettu, funktiota kutsuttaessa parametrina voidaan antaa mikä tahansa lauseke, jonka arvo voidaan laskea. Myös toisen funktion kutsu on tällainen lauseke. Voidaan siis kirjoittaa funktion kutsu, jonka sisällä on parametrin arvona toisen (tai joissakin tapauksissa myös saman) funktion kutsu. Tällöin aina sisempi funktiokutsu suoritetaan ensin, ja sen paluuarvoa käytetään ulomman funktion parametrina.

Hyvin tyypillinen esimerkki sisäkkäisistä funktiokutsuista on tilanne, jossa käyttäjältä luetaan arvo Pythonin valmiin `input`-funktion avulla ja tämä arvo pitää sitten muuttaa kokonais- tai desimaaliluvuksi Pythonin valmiin `int`- tai `float`-funktion avulla. Tähän asti olleissa esimerkeissä tämä on tehty kahdella eri ohjelmarivillä, joista ensimmäisellä luetaan haluttu arvo ja toisella se muutetaan sopivan tyyppiseksi luvuksi, esimerkiksi:

```
rivi = input("Anna nopeuden ylarajan sekunnit: ")
maili_sek = int(rivi)
```

Käskyt voidaan kuitenkin yhdistää yhdelle riville, jos `input`-funktion kutsu annetaan suoraan `int`-funktion parametrina:

```
maili_sek = int(input("Anna nopeuden ylarajan sekunnit: "))
```

Nyt siis suoritetaan ensin sulkujen sisällä oleva `input`-funktion kutsu, joka pyytää ja lukee käyttäjältä sekuntien arvon. Funktio palauttaa käyttäjältä saadun arvon merkkijonona. Tämä merkkijono tulee ulomman `int`-funktion kutsun parametrin arvoksi. Funktio `int` taas muuttaa sille parametrina annetun merkkijonon vastaavaksi kokonaisluvuksi. (Tarkasti ottaen funktio ei muuta alkuperäistä merkkijonoa, vaan se tekee parametrina annettua merkkijonoa vastaavan kokonaisluvun ja palauttaa arvonaan sen.)

4.3.2 Totuusarvon palauttavat funktiot ja niiden paluuarvon käyttö

Aikaisemmissa esimerkeissä funktion paluuarvo oli yksittäinen tai useampi luku. Monesti ohjelmia kirjoitettaessa on kuitenkin kätevää kirjoittaa erilaisiin tarkistuksiin funktioita, jotka tarkistavat, onko jokin ehto tosi vai epätosi ja palauttavat sen mukaan joko arvon `True` tai `False`. Tällaista paluuarvoa voidaan sitten käyttää esimerkiksi `if`- tai `while`-käskyn ehdossa.

Tarkastellaan seuraavaa esimerkkiä: halutaan kirjoittaa ohjelma, joka laskee kahden tason suoran leikkauspisteen. Suoraa kuvaava yhtälö annetaan muodossa $ax + by + c = 0$, missä a, b ja c ovat kokonaislukuja. Leikkauspiste voidaan laskea muodostamalla suoria kuvaavista yhtälöistä yhtälöpari ja ratkaisemalla tämä yhtälöpari.

Leikkauspisteen laskemisessa tulee kuitenkin vastaan erilaisia erikoistapauksia:

- Suorat voivat olla yhtenevät eli niillä on sama kulmakerroin ja sama leikkauspiste y-akselin kanssa. Tällöin suorat leikkaavat toisensa koko ajan eikä niillä ole yhtä leikkauspistettä.
- Suorat voivat olla yhdensuuntaiset, mutta erilliset. Niillä on siis sama kulmakerroin, mutta ne leikkaavat y-akselin eri pisteissä. Tällöin suorilla ei ole lainkaan leikkauspisteitä.
- Molemmat suorat ovat y-akselin suuntaisia. Tällöin ne ovat joko yhtenevät tai sitten yhdensuuntaiset mutta erilliset sen mukaan, leikkaavatko ne x-akselin samassa vai eri kohdassa.

Näissä tapauksissa ei voida käyttää suoraan yhtälöryhmän ratkaisevaa kaavaa, vaan tapaukset on käsiteltävä erikseen.

Erikoistapausten käsittelemistä varten kannattaa kirjoittaa oma funktio, joka tarkistaa, onko kahdella suoralla sama kulmakerroin, ja toinen funktio, joka tarkistaa, onko kahden yhdensuuntaisen suoran leikkauspiste y-akselin kanssa sama. Jos molemmat suorat ovat yhdensuuntaisia y-akselin kanssa, jälkimmäinen funktio tarkistaa, onko niiden leikkauspiste x-akselin kanssa sama.

Tarkastellaan funktiota, joka tarkistaa, onko kahden suoran kulmakertoimet samat. Funktio saa parametrinaan ensimmäisen suoran x:n ja y:n kertoimet (`xkerroin1` ja `ykerroin1`) sekä vastaavat toisen suoran kertoimet (`xkerroin2` ja `ykerroin2`). Suorien yhtälössä esiintyvällä vakiolla ei ole merkitystä kulmakertoimia määrätessä. Ensimmäisen suoran kulmakerroin on nyt `xkerroin1 / ykerroin1`. Kulmakertointen yhtäsuuruuden tutkiminen lausekkeella `xkerroin1 / ykerroin1 == xkerroin2 / ykerroin2` ei kuitenkaan ole järkevää, koska pyöristysvirheiden takia jakolaskujen tuloksissa voi olla pieni ero, vaikka kulmakertoimet oikeasti olisivatkin samoja.

Yksi vaihtoehto olisi sieventää kulmakertoimet murtolukuina, mutta sievennykseltä säästytään, jos tutkitaan tuloja `xkerroin1 * ykerroin2` ja `xkerroin2 * ykerroin1`. Jos ne ovat yhtäsuuret, myös suorien kulmakertoimet ovat yhtäsuuret, vaikka kulmakertoimia kuvaavat murtoluvut eivät olisikaan sievennetyssä muodossa. Funktio siis tutkii näiden tulojen yhtäsuuruutta ja palauttaa arvon `True`, jos tulot ovat yhtäsuuria ja muuten arvon `False`.

Kirjoitetaan funktio, joka saa parametrina kahden suoran yhtälöiden x:n ja y:n kertoimet ja tarkistaa, onko suoriten kulmakerroin sama. Funktio palauttaa arvon `True`, jos kulmakerroin on sama ja arvon `False`, jos suorilla on eri kulmakerroin:

```
def sama_kulmakerroin(xkerroin1, ykerroin1, xkerroin2, ykerroin2):
    if (xkerroin1 * ykerroin2 == xkerroin2 * ykerroin1):
        return True
    else:
        return False
```

Katsotaan seuraavaksi, miten funktion paluuarvoa voi käyttää pääohjelmassa. Funktion paluuarvo on siis `True` tai `False` ja sitä voi käyttää if-käskyn ehdossa esimerkiksi seuraavasti:

```
if sama_kulmakerroin(eka_a, eka_b, toka_a, toka_b) == True:
```

Ehdossa oleva yhtäsuuruusvertailu arvoon `True` on kuitenkin tarpeeton, sillä funktion paluuarvo on jo itsessään tosi tai epätosi, jolloin sitä voidaan käyttää suoraan if-käskyn ehtona seuraavasti:

```
if sama_kulmakerroin(eka_a, eka_b, toka_a, toka_b):
```

Tällöin if-käskyyn kuuluvat käskyt suoritetaan, jos funktion `sama_kulmakerroin` paluuarvo on `True`, ja jätetään suorittamatta, jos funktion paluuarvo on `False`.

Jos taas halutaan tehdä if-käsky, johon kuuluvat käskyt suoritetaan silloin kun funktio palauttaa arvon `False`, voidaan tämä tehdä `not`-operaattorin avulla:

```
if not sama_kulmakerroin(eka_a, eka_b, toka_a, toka_b):
```

Tässä `if`-käskyn ehto on tosi aina, kun funktion `sama_kulmakerroin` paluuarvo on `False`.

Seuraavaksi koko suorien leikkauspisteen laskeva ohjelma. Aikaisemmin kuvatun funktion `sama_kulmakerroin` lisäksi ohjelmaan on määritelty funktiot `sama_akselileikkaus`, joka tutkii, leikkaavatko yhdensuuntaiset suorat y-akselin (y-akselin suuntaiset suorat x-akselin) samassa pisteessä, `laske_leikkaus`, joka laskee suorien leikkauspisteen, sekä `pyyda_suora`, joka pyytää yhden suoran kertoimet. Merkki \ pääohjelman kahden rivin lopussa kertoo Python-tulkille, että rivi jatkuu seuraavalla rivillä.

Yksinkertaisuuden vuoksi ohjelma ei tarkista, että sille on todella annettu suoran yhtälö. Ohjelma ei siis toimi oikein silloin, jos sille annetaan "suoran" yhtälö, jossa sekä x:n että y:n kertoimet ovat nollia.

```
def sama_kulmakerroin(xkerroin1, ykerroin1, xkerroin2, ykerroin2):
    if xkerroin1 * ykerroin2 == xkerroin2 * ykerroin1:
        return True
    else:
        return False


def sama_akselileikkaus(a1, b1, c1, a2, b2, c2):
    if b1 == 0 and b2 == 0:  # suorat y-akselin suuntaiset
        if a1 * c2 == a2 * c1:
            return True
        else:
            return False
    elif b1 * c2 == b2 * c1:
        return True
    else:
        return False


def laske_leikkaus(a1, b1, c1, a2, b2, c2):
    yleikkaus = (a1 * c2 - a2 * c1) / (b1 * a2 - b2 * a1)
    if a1 == 0:
        xleikkaus = (-b2 * yleikkaus - c2) / a2
    else:
        xleikkaus = (-b1 * yleikkaus - c1) / a1
    return xleikkaus, yleikkaus


def pyyda_suora():
    print("Anna suoran yhtalo")
    a = int(input("Anna x:n kerroin: "))
    b = int(input("Anna y:n kerroin: "))
    c = int(input("Anna vakio: "))
    return a, b, c
```

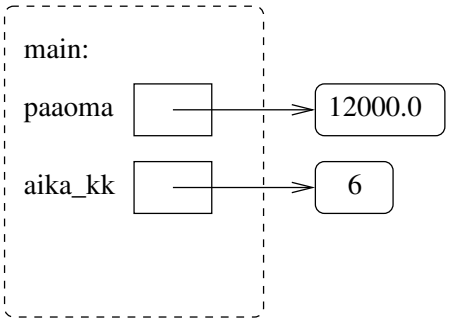
```
def main():
    print("Etsitaan kahden suoran ax + by + c leikkauspiste")
    eka_a, eka_b, eka_c = pyyda_suora()
    toka_a, toka_b, toka_c = pyyda_suora()
    if sama_kulmakerroin(eka_a, eka_b, toka_a, toka_b) and \
           sama_akselileikkaus(eka_a, eka_b, eka_c, toka_a, toka_b, toka_c):
        print("Suorat ovat yhtenevat koko pituudeltaan.")
    elif sama_kulmakerroin(eka_a, eka_b, toka_a, toka_b):
        print("Suorat ovat yhdensuuntaiset, ei leikkauspistetta.")
    else:
        x_koord, y_koord = laske_leikkaus(eka_a, eka_b, eka_c,\
                                          toka_a, toka_b, toka_c)
        print("Leikkauspiste on {:.2f}, {:.2f}".format(x_koord, y_koord))


main()
```

4.4 Kertausta: parametrit, muuttujat ja paluuarvot

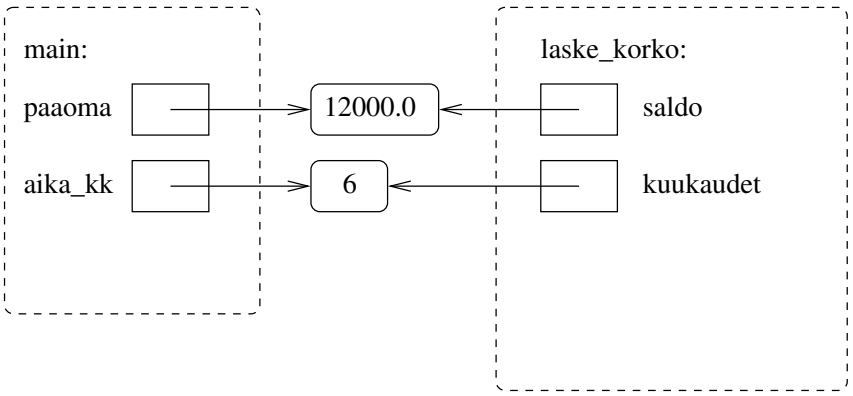
Tässä kappaleessa kerrataan vielä parametrien, muuttujien ja paluuarvon käsitettä käyttämällä esimerkkinä ohjelmaa, joka laskee sijoitustilille kertyvän koron. Oletetaan, että pankki maksaa sijoitustilille vuosittaista korkoa 0,5 %, jos talletettava summa on 10000 euroa, 1,0 %, jos talletussumma on vähintään 10000 mutta alle 15000 euroa ja 1,5 %, jos talletussumma on tätä suurempi. Ohjelma kysyy käyttäjältä talletettavan rahasumman ja talletusajan kuukausina. Ohjelma laskee ja tulostaa talletukselle kertyvän koron.

```
def laske_korko(saldo, kuukaudet):
    if saldo < 10000.0:
        korkoprosentti = 0.5
    elif saldo < 15000.0:
        korkoprosentti = 1.0
    else:
        korkoprosentti = 1.5
    kokonaiskorko = korkoprosentti / 100.0 * kuukaudet / 12 * saldo
    return kokonaiskorko


def main():
    print("Ohjelma laskee sijoitustilin tuoton.")
    paaoma = float(input("Anna sijoitettava paaoma (eur): "))
    aika_kk = int(input("Montako kuukautta rahat pidetaan tililla? "))
    korko = laske_korko(paaoma, aika_kk)
    print("Saat rahoille korkoa {:.2f} euroa.".format(korko))


main()
```

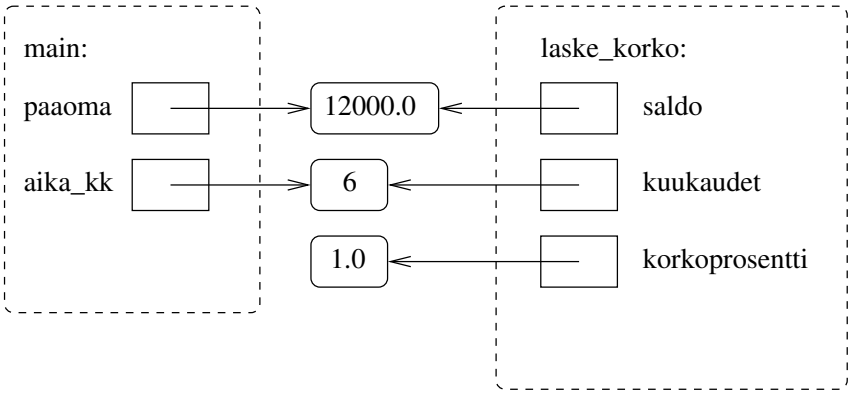
Ohjelman suoritus alkaa siis pääohjelmasta eli `main`-funktiosta. Kun Python-tulkki aloittaa funktion suorittamisen, varataan tietokoneen keskusmuistista oma muistialue juuri tämän funktion käyttämien muuttujien tallentamiseen. Tätä muistialuetta kutsutaan *kehykseksi*. Jos käyttäjä antaa sijoitettavaksi pääomaksi 12000 euroa ja talletusajaksi 6 kuukautta, on tilanne keskusmuistissa ennen funktion `laske_korko` kutsua alla olevan kuvan näköinen. (Itse arvot 12000.0 ja 6 on piirretty kehyksen ulkopuolelle, koska ne voidaan tallentaa keskusmuistissa muualle. Kehyksen alueella on kuitenkin muistipaikat funktion käyttämille muuttujille ja näissä muistipaikoissa tieto siitä, missä itse arvo sijaitsee.)



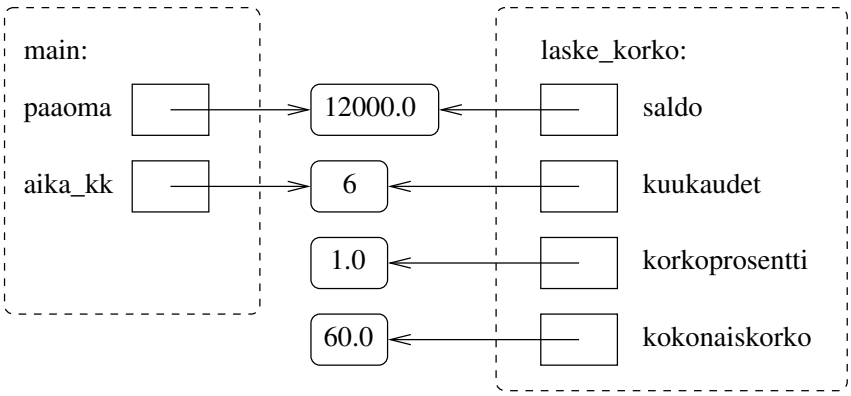
Kun ohjelmassa kutsutaan funktiota `laske_korko`, luodaan sen käyttämiä muuttujia varten oma kehys. Ennen funktion suorituksen alkua kopioidaan funktiokutsussa annetut parametrien arvot vastaavien `laske_korko`-funktion parametrien arvoiksi. Tilanne on nyt alla olevan kuvan näköinen.



Funktion `laske_korko` if-käskyssä annetaan arvo muuttujalle `korkoprosentti`. If-käskyn suorituksen jälkeen muistin tilanne näyttää seuraavalta:

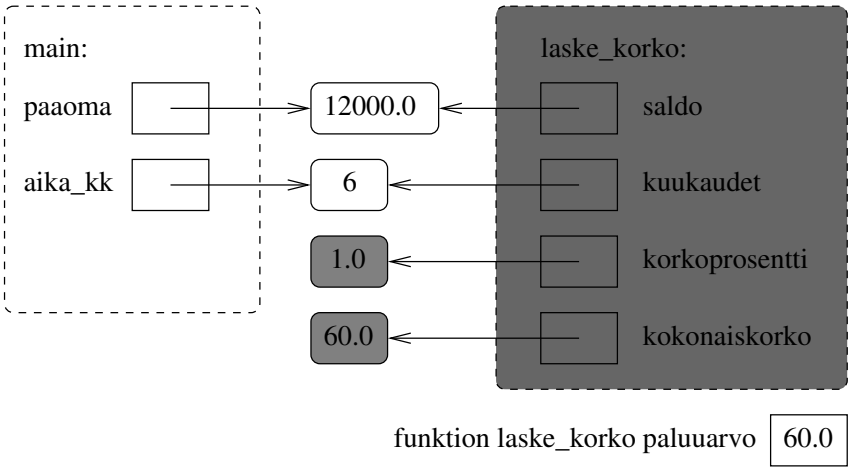


Seuraavaksi funktiossa `laske_korko` lasketaan ja annetaan arvo muuttujalle `kokonaiskorko`. Tilanne on alla olevan kuvan mukainen.

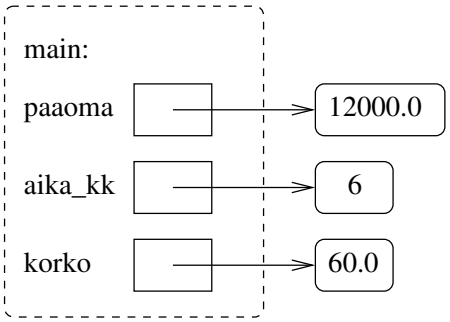


Pääohjelmalla (funktiolla `main`) ja funktiolla `laske_korko` on siis molemmilla omat muuttujansa. Funktion sisällä ei voi käyttää suoraan sellaisia muuttujia, jotka ovat toisen funktion kehyksessä. Esimerkiksi funktiossa `laske_korko` ei voi käyttää suoraan muuttujaa `paaoma`. (Tarkasti ottaen funktiossa `laske_korko` saa käyttää `paaoma`-nimistä muuttujaa, mutta se ei ole silloin sama muuttuja kuin pääohjelmassa, vaan toinen muuttuja, jolla vain sattuu olemaan sama nimi ja jonka arvo voi muuttua itsenäisesti riippumatta pääohjelman samannimisen muuttujan arvosta.)

Kun funktio `laske_korko` on suoritettu loppuun, sen kehys tuhotaan tietokoneen keskusmuistista. Funktion käyttämät muuttujat häviävät. Kuitenkin `return`-käskyssä palautettu arvo otetaan talteen ja välitetään ohjelmassa sinne, mistä funktiota kutsuttiin.



Pääohjelmassa funktion `laske_korko` palauttama arvo sijoitetaan muuttujan `korko` arvoksi. Huomaa, että funktion `laske_korko` käyttämiä muuttujia ei ole enää tallessa missään. Jos pääohjelmassa kutsuttaisiin funktiota uudelleen, aloitettaisin sen seuraava suoritus puhtaalta pöydältä eikä edellisen suorituskerran tietoja olisi tallella. Ennen `print`-käskyn suoritusta tilanne keskusmuistissa näyttää seuraavalta



Huomautus: Ohjelmassa olisi ollut järkevää tallentaa eri korkoprosentit ja rajat niiden käytölle vakiomuuttujiin. Sitä ei ole kuitenkaan tehty tässä esimerkissä, koska tässä on haluttu pitää muuttujien määrä pienenä, jotta esimerkkikuvat olisivat tarpeeksi selviä.

4.5 Tiivistelmä funktioiden määrittelystä ja käytöstä

Tähän kappaleeseen on vielä koottu lyhyesti esimerkkejä luvussa esitetyistä keskeisistä asioita ilman selityksiä.

Funktion määrittely

```
def funktion_nimi(parametri1, parametri2, parametri3):
    funktioon kuuluvat käskyt
```

Parametrien määrä voi vaihdella.

Funktion kutsu

```
funktion_nimi(arvo1, arvo2, arvo3)
```

Suoritetaan funktio `funktion_nimi` niin, että `parametri1` saa alkuarvokseen `arvo1`n, `parametri2` saa alkuarvokseen `arvo2`n ja `parametri3` saa alkuarvokseen `arvo3`n. Arvot voivat olla mitä tahansa lausekkeita, joiden arvo pystytään laskemaan.

Arvon palauttaminen (funktion sisällä)

```
return lauseke1
```

`lauseke1` voi olla mikä tahansa lauseke, jonka arvo pystytään laskemaan. Arvon palauttaminen on eri asia kuin arvon tulostaminen!

Funktion paluuarvon käyttö (siellä, missä funktiota kutsutaan)
Paluuarvon voi ottaa talteen muuttujaan sijoituskäskyllä

```
muuttuja1 = funktion_nimi(arvo1, arvo2, arvo3)
```

tai sen voi tulostaa suoraan

```
print("Lopputulos on", funktion_nimi(arvo1, arvo2, arvo3))
```

tai sitä voi käyttää jossain lausekkeessa

```
summa = 5.0 + funktion_nimi(arvo1, arvo2, arvo3)
```

Kaikissa tapauksessa esitetty rivi on samalla myös funktion kutsu.

Useamman arvon palauttaminen (funktion sisällä)

```
return lauseke1, lauseke2
```

ja esimerkki arvojen ottamisesta talteen

```
muuttuja1, muuttuja2 = funktion_nimi(arvo1, arvo2, arvo3)
```

Esimerkki totuusarvon (`True` tai `False`) palauttavan funktion kutsumisesta ja paluuarvon käytöstä if-käskyn ehdossa

```
if funktion_nimi(arvo1, arvo2, arvo3):
    käskyt, jotka suoritetaan, jos funktio palauttaa arvon True
else:
    käskyt, jotka suoritetaan, jos funktio palauttaa arvon False
```

Luku 5

Listat, merkkijonot ja sanakirja

5.1 Lista

Tarkastellaan seuraavaa ongelmaa: Meteorologi haluaa ohjelman, johon hän voi syöttää kuukauden jokaisen päivän maksimilämpötilan. Kun lämpötilat on syötetty, ohjelman pitää tulostaa lämpötilojen keskiarvo ja syötetyt lämpötilat alkuperäisessä järjestyksessä. Oletetaan, että jokaisessa kuukaudessa on 30 päivää.

Jos ohjelman pitäisi tulostaa vain lämpötilojen keskiarvo, olisi helppo kerätä lämpötilojen summa yhteen muuttujaan ja laskea lopuksi keskiarvo tämän muuttujan avulla. Nyt kuitenkin halutaan tulostaa myös syötetyt lämpötilat. Tämän vuoksi jokaisen päivän lämpötila pitää tallentaa erikseen. Tähän asti opituilla välineillä tarvitsisimme kolmekymmentä eri muuttujaa lämpötilojen tallentamista varten. Meidän pitäisi myös kirjoittaa kolmekymmentä käskyä lämpötilojen lukemiseen ja toiset kolmekymmentä käskyä lämpötilojen tulostamiseen. Näin ohjelmasta tulisi tarpeettoman pitkä. Lisäksi sen muuttaminen toimimaan jollain muulla lämpötilamäärällä olisi hankalaa. Jos esimerkiksi haluaisimme ohjelman käsittelevän samalla tavalla koko vuoden lämpötilat, pitäisi ohjelmaan lisätä 335 uutta muuttujaa sekä saman verran uusia luku- ja tulostuskäskyjä.

Tällainen ongelma voidaan ratkaista selvästi helpommin käyttämällä *listaa*. Lista on tietorakenne, johon voi tallentaa useita arvoja. Näihin arvoihin pääsee käsiksi indeksin avulla. Jos on esimerkiksi lista `lampotilat`, niin listan viidenteen alkioon pääsee käsiksi kirjoittamalla `lampotilat[4]`. (Hakasuluissa oleva indeksi on 4 eikä 5 siksi, että listan ensimmäisen alkion indeksi on aina 0.)

Uusi lista luodaan esimerkiksi kirjoittamalla

```
lampotilat = []
```

Sijoituskäskyn oikealla puolella luodaan tyhjä lista ja sijoituskäsky yhdistää muuttujan `lampotilat` luotuun listaan.

Tämän jälkeen listan loppuun voi lisätä uuden arvon kirjoittamalla

```
lampotilat.append(arvo)
```

Uusi arvo lisätään aina vanhan listan loppuun niin, että listan pituus kasvaa yhdellä. Esimerkiksi ohjelma

```
def main():
    lampotilat = []
    lampotilat.append(15.0)
    lampotilat.append(22.5)
    lampotilat.append(-12.9)
    print(lampotilat)


main()
```

tulostaa

```
[15.0, 22.5, -12.9]
```

Tässä siis listaan lisättiin lukuja *metodin* `append` avulla. Kuten funktio, metodi on ohjelmakoodin osa, jolle on annettu oma nimi. Metodia kuitenkin kutsutaan eri tavalla kuin funktiota: lista, johon lisäys tehdään, ei olekaan metodin parametrina vaan se on kirjoitettu kutsuun ennen metodin nimeä ja erotettu metodin nimestä pisteellä. Metodien määrittely ja niiden kutsutapa liittyy olio-ohjelmoinnin piirteisiin, joihin tutustutaan tarkemmin luvussa 7. Listojen ja myöhemmin merkkijonojen yhteydessä kuitenkin käytämme joitakin Pythonin valmiita metodeita.

Jos listassa on vähintään `i+1` alkiota, pystyy indeksille `i` sijoittamaan uuden arvon kirjoittamalla

```
listamuuttuja[i] = arvo
```

tällöin indeksillä `i` oleva vanha arvo häviää ja ja se korvautuu sijoituskäskyn oikealla puolella olevalla arvolla.

Jos esimerkiksi edellisen ohjelman `main`-funktion loppuun lisätään rivi

```
lampotilat[1] = 32.0
```

on lista `lampotilat` rivin suorittamisen jälkeen

```
[15.0, 32.0, -12.9]
```

Tällaisella sijoituskäskyllä ei voi kuitenkaan kasvattaa listan kokoa eli hakasuluissa olevan indeksin täytyy olla aina vähintään 0 ja korkeintaan listan koko - 1. Jos edellä esitetyn ohjelman `main`-funktion loppuun lisää käskyn

```
lampotilat[3] = 18.0
```

johtaa se ohjelman kaatumiseen ja seuraavaan virheilmoitukseen:

```
Traceback (most recent call last):
  File "lampotilat1.py", line 10, in <module>
    main()
  File "lampotilat1.py", line 7, in main
    lampotilat[3] = 18.0
IndexError: list assignment index out of range
```

Tässä viimeinen rivi "list assignment index out of range" kertookin aika suoraan, millaisesta virheestä on ollut kysymys eli että sijoituksessa listaan käytetty indeksi on ollut sallittujen rajojen ulkopuolella.

Listan alkioiden arvoja voidaan käyttää indeksoinnin avulla samalla tavalla kuin minkä tahansa muiden muuttujien arvoja, esimerkiksi

```
summa = lampotilat[1] + lampotilat[2]
```

laskee `lampotilat`-listassa indekseillä 1 ja 2 olevien alkioiden arvot yhteen ja sijoittaa näin saadun tuloksen muuttujan `summa` arvoksi. Tässäkin käytettyjen indeksien pitää olla listan indeksialueen sisällä.

Palataan sitten alkuperäiseen esimerkkiin: ohjelmaan, joka kirjaa kuukauden eri päivien lämpötilat ja sen jälkeen tulostaa nämä lämpötilat uudelleen sekä niiden keskiarvon.

Ohjelmassa pitää siis ensinnäkin luoda lista lämpötiloja varten. Tämä voidaan tehdä käskyllä

```
lampotilat = []
```

Sitten tarvitaan toistokäsky, joka pyytää käyttäjältä eri päivien lämpötilat ja lisää ne `lampotilat`-listaan. Toistokäsky tarvitsee tiedon siitä, montako lämpötilaa käyttäjältä luetaan, jotta se osaisi lopettaa lämpötilojen pyytämisen oikean määrän luettuaan. (Vaihtoehtoisesti ohjelma voidaan rakentaa niin, että käyttäjä ilmoittaa jollain sovitulla arvolla, että kaikki halutut lämpötilat on luettu). Otetaan tätä lämpötilojen määrää varten käyttöön vakio `LKM`, jolle siis annetaan arvo 30. Lisäksi `while`-käskyssä tarvitaan laskuri, joka pitää kirjaa siitä, kuinka monta lämpötilaa on jo luettu. Seuraavassa ohjelmassa käytetään siihen takoitukseen muuttujaa `i`. Lämpötilat lukeva ja listaan lisäävä `while`-käsky voi siis olla sitä edeltävine alustuksineen seuraava:

```
LKM = 30
i = 0
while i < LKM:
    lampo = float(input("Seuraava lampotila: "))
    lampotilat.append(lampo)
    i += 1
```

Tarkastellaan sitten sitä osaa ohjelmasta, joka laskee lämpötilojen keskiarvon. Kirjoitetaan toistokäsky, joka laskee listassa `lampotilat` olevien lämpötilojen summan.

(Summan laskemisen voi käytännössä yhdistää samaan toistokäskyyn joko lämpötilojen lukemisen tai tulostamisen kanssa, mutta tässä se on esitetty alkuksi selvyyden vuoksi erikseen.) Tarvitsemme muuttujan, johon summaa kerätään. Tämä muuttuja alustetaan aluksi nollaksi. Sen jälkeen käydään toistokäskyn avulla koko lista `lampotilat` läpi ja jokaisella kierroksella lisätään yksi alkio aikaisemmin laskettuun summaan:

```
    summa = 0.0
    i = 0
    while i < LKM:
        summa += lampotilat[i]
        i += 1
```

Tässä siis `i`:n arvo vaihtelee eri kierroksilla niin, että ensimmäisellä kierroksella muuttujaan `summa` lisätään alkio `lampotilat[0]`, toisella kierroksella `lampotilat[1]` ja niin edelleen, kunnes viimeisellä kierroksella summaan lisätään `lampotilat[LKM - 1]`. Listan viimeisen alkion indeksi on `LKM - 1`, vaikka listassa on `LKM` alkiota, koska ensimmäisen alkion indeksi on 0.

Listan läpikäynti voidaan kirjoittaa myös `for`-käskyn avulla. Tämä yleensä onkin kätevämpi tapa silloin, kun listaan ei tarvitse enää lisätä uusia alkioita, sillä `for`-käskyssä ohjelmoijan ei tarvitse ilmaista listan kokoa eikä pitää huolta indeksimuuttujan kasvattamisesta. Käskyn yleinen muoto on tällöin

```
for elementti in lista:
    tee jotain listan alkiolle elementti
```

Tässä muuttuja `elementti` saa vuorotellen arvokseen kunkin listan alkion niin, että ensimmäisellä kierroksella muuttujan `elementti` arvo on listan ensimmäinen alkio, toisella kierroksella toinen alkio jne. Listan `lampotilat` alkioiden summa voidaan laskea seuraavalla `for`-käskyllä:

```
    summa = 0.0
    for arvo in lampotilat:
        summa += arvo
```

Seuraavaksi koko ohjelma, joka pyytää lämpötilat, tulostaa ne ja laskee ja tulostaa niiden keskiarvon. Lämpötilojen tulostus ja niiden summan laskeminen on yhdistetty samaan toistokäskyyn.

```
def main():
    LKM = 30
    lampotilat = []
    i = 0
    print("Anna", LKM, "lampotilaa.")
    while i < LKM:
        lampo = float(input("Seuraava lampotila: "))
        lampotilat.append(lampo)
        i += 1
```

```
    summa = 0.0
    print("Annetut lampotilat")
    for arvo in lampotilat:
        print(arvo)
        summa += arvo
    keskiarvo = summa / LKM
    print("Lampotilojen keskiarvo {:.2f}".format(keskiarvo))


main()
```

Seuraavaksi esimerkki ohjelman toiminnasta. Vakion `LKM` arvoksi on vaihdettu 5, jotta esimerkkiajosta ei tulisi turhan pitkä.

```
Anna 5 lampotilaa.
Seuraava lampotila: 25.0
Seuraava lampotila: 12.0
Seuraava lampotila: -5.0
Seuraava lampotila: -10.0
Seuraava lampotila: 2.0
Annetut lampotilat
25.0
12.0
-5.0
-10.0
2.0
Lampotilojen keskiarvo 4.80
```

Kun uusi lista luodaan, sen ei tarvitse välttämättä olla tyhjä, vaan listaa luodessa voidaan samalla antaa listaan aluksi kuuluvat alkiot, esimerkiksi

```
numerolista = [2, 4, 6, 8]
```

luo listan, jossa on neljä kokonaislukua (2, 4, 6 ja 8 tässä järjestyksessä) ja liittää muuttujan `numerolista` luotuun listaan. Listan alkioihin päästään käsiksi indeksoinnin avulla, ja listaan voidaan myös lisätä uusia alkioita `append`-metodilla ihan samalla tavalla kuin tyhjänä luotuun listaan.

Sen sijaan, että aikaisemmassa lämpötilaesimerkissä kasvatetaan listan kokoa aina uutta lämpötilaa lisättäessä, voidaan luoda heti aluksi 30-alkioinen lista ja lisätä sijoituskäskyllä alkiot siihen jo olemassaoleville paikoille. Ei ole tosin mahdollista luoda listaa, jossa olisi 30 tyhjää paikkaa, mutta sen sijaan on mahdollista luoda 30-paikkainen lista, jonka jokaisessa alkiossa on aluksi arvo 0.0 seuraavasti:

```
lampotilat = [0.0] * 30
```

tai vakiota `LKM` käyttämällä

```
LKM = 30
lampotilat = [0.0] * LKM
```

Tällöin lämpötiloja luettaessa niitä ei enää lisätä listaan `append`-käskyllä, vaan lämpötilojen lisääminen voidaan tehdä `while`-käskyä käytettäessä indeksoimalla seuraavasti:

```
i = 0
while i < LKM:
    lampo = float(input("Seuraava lampotila: "))
    lampotilat[i] = lampo
    i += 1
```

5.1.1 Lista funktion parametrina ja funktion palauttamana arvona

Edellä esitetty ohjelma toimii täysin oikein, mutta sitä voi vielä selkeyttää jakamalla pääohjelmassa esiintyviä toimintoja sopiviin funktioihin.

Lämpötilalistan luominen ja arvojen lukeminen siihen on yksi kokonaisuus, joka sopii hyvin omaksi funktiokseen `kysy_lampotilat`. Tieto perustetusta listasta ja siihen tallennetuista lämpötiloista pitää kuitenkin saada jotenkin pääohjelman käyttöön. Tähän on kaksi keinoa. Toinen on luoda lista pääohjelmassa ja antaa lista sitten parametrina siihen lämpötilat tallentavalle funktiolle. Parametrina annettuun listaan tehdyt muutokset ja lisäykset näkyvät myös pääohjelmassa. Tästä kerrotaan tarkemmin kappaleessa 5.5.

Toinen tapa on luoda lista funktiossa `kysy_lampotilat` ja palauttaa tämä lista funktion lopussa. Aivan samalla tavalla kuin funktion palauttama lukuarvo voidaan ottaa käyttöön pääohjelmassa, myös funktion palauttama lista voidaan ottaa käyttöön pääohjelmassa. Seuraavassa esimerkissä on käytetty tätä tapaa:

```
def kysy_lampotilat():
    LKM = 30
    lampotilat = [0.0] * LKM
    i = 0
    print("Anna", LKM, "lampotilaa")
    while i < LKM:
        lampo = float(input("Seuraava lampotila: "))
        lampotilat[i] = lampo
        i += 1
    return lampotilat
```

Pääohjelmassa funktiota voidaan kutsua esimerkiksi seuraavasti:

```
lampolista = kysy_lampotilat()
```

Tällöin muuttuja `lampolista` saa arvokseen funktion `kysy_lampotilat` palauttaman listan, joka sisältää käyttäjän antamat lämpötilat.

Vastaavasti voidaan kirjoittaa oma funktio listassa olevien lämpötilojen tulostamiseen:

```
def tulosta_lampotilat(lammot):
    print("Annetut lampotilat")
    for arvo in lammot:
        print(arvo)
```

Funktio saa parametrina listan, joka sisältää aikaisemmin luetut lämpötilat. Funktiossa oleva `for`-käsky käy tämän listan läpi ja tulostaa jokaisen siinä olevan lämpötilan. Funktiota voidaan kutsua pääohjelmasta esimerkiksi seuraavasti:

```
tulosta_lampotilat(lampolista)
```

Jos pääohjelman muuttujaan `lampolista` on aikaisemmin sijoitettu luetut lämpötilat sisältävä lista, niin funktiota `tulosta_lampotilat` suoritettaessa parametri `lammot` tarkoittaa tätä samaa listaa.

Kirjoitetaan myös funktio keskiarvon laskemista varten. Keskiarvon laskemista ei ole tässä sisälletetty lämpötilat tulostavaan funktioon kahdesta syystä: Ensiksi, keskiarvon laskeminen on selvästi lämpötilojen tulostamisesta erillinen toimenpide. Kun kirjoitamme funktion jonkin asian suorittamista varten, on selvempää, että funktio tekee juuri tämän asian eikä sen oheen ole liitetty funktion alkuperäiseen tarkoitukseen liittymättömiä asioita. Toiseksi, lämpötilojen tulostaminen sisältää selkeästi ohjelman käyttöliittymään liittyviä asioita, kun taas keskiarvon laskemiseen ei liity kommunikointia ohjelman käyttäjän kanssa. On järkevää erottaa toisistaan ohjelman sellaiset osat, jotka kommunikoivat käyttäjän kanssa ja sellaiset, jotka suorittavat puhdasta laskentaa. Tällöin ohjelman muuttaminen myöhemmin on helpompaa, jos halutaan muuttaa ohjelman käyttöliittymä toisenlaiseksi (esimerkiksi vaihtaa nyt ohjelmassa käytössä oleva tekstipohjainen käyttöliittymä graafiseen käyttöliittymään).

Keskiarvon laskemisessa tarvitaan listassa olevien lämpötilojen määrä. Funktiossa `kysy_lampotilat` on tätä varten määritelty vakio `LKM`, mutta toisessa funktiossa määritelty vakio ei näy toisen funktion sisällä, eikä vakiota voida näin ollen suoraan käyttää funktiossa `laske_keskiarvo`. Yksi vaihtoehto on siirtää vakiolle `LKM` arvon antava sijoituskäsky `LKM = 30` funktioiden ulkopuolelle. Tällöin vakio on käytettävissä kaikissa samaan tiedostoon kirjoitetuissa funktioissa. Tässä tapauksessa siirto on kuitenkin tarpeeton, koska listassa olevien lämpötilojen määrän saa selville helposti myös ilman vakiota. Pythonissa on listoja varten valmis funktio `len`, joka palauttaa funktiolle parametrina annetun listan pituuden. Listan `lampotilalista` pituuden saa siis selville ilmauksella `len(lampotilalista)`.

Aina jakolaskuja suoritettaessa on hyvä tarkistaa se, että jakaja ei ole nolla, koska nollalla jakaminen aiheuttaisi ohjelman kaatumisen. Funktiossa `laske_keskiarvo` on myös tämä mahdollisuus otettu huomioon. Jos lämpötilojen määrä on 0, funktio palauttaa arvon 0.0. Pääohjelmassa on kutsuttu keskiarvon laskevaa funktiota (`lampolista` on jälleen annetttu funktiolle parametrina) ja tulostettu funktion palauttama arvo. Seuraavaksi koko ohjelma:

```
def kysy_lampotilat():
    LKM = 30
    lampotilat = [0.0] * LKM
    i = 0
    print("Anna", LKM, "lampotilaa")
    while i < LKM:
        lampo = float(input("Seuraava lampotila: "))
        lampotilat[i] = lampo
        i += 1
    return lampotilat

def tulosta_lampotilat(lammot):
    print("Annetut lampotilat")
    for arvo in lammot:
        print(arvo)

def laske_keskiarvo(lampotilalista):
    summa = 0.0
    for lampotila in lampotilalista:
        summa += lampotila
    lukumaara = len(lampotilalista)
    if lukumaara > 0:
        keskiarvo = summa / lukumaara
    else:
        keskiarvo = 0.0
    return keskiarvo

def main():
    lampolista = kysy_lampotilat()
    tulosta_lampotilat(lampolista)
    keskiarvo = laske_keskiarvo(lampolista)
    print("Lampotilojen keskiarvo on {:.2f}".format(keskiarvo))

main()
```

Annetussa ohjelmassa on käytetty useita eri muuttujia ja parametreja, joiden arvona on lämpötilat sisältävä lista. Kunkin funktion sisällä listaan käydään käsiksi oman muuttujan tai parametrin kautta. Asian selventämiseksi esimerkissä on käytetty jokaisessa funktiossa tuolle muuttujalle tai parametrille eri nimeä. Yhdessä funktiossa määritelty nimi ei näy suoraan toisessa funktiossa, mutta tiedon käsiteltävästä listasta voi välittää funktiolta toiselle parametrien ja paluuarvojen välityksellä.

Käytännön ohjelmoinnissa eri funktioissa käytetään samaa asiaa tarkoittavasta parametrista kuitenkin usein samaa muuttujan ja parametrin nimeä, vaikka Python-ohjelman kannalta kysymys on useasta eri muuttujasta tai parametrista. Käytännössä esimerkiksi yllä oleva ohjelma kirjoitettaisiin usein niin, että käytössä olisi lämpötilat sisältävälle listalle vain yksi nimi, `lampotilat`, jota käytettäisiin sekä parametrina funktioissa `kysy_lampotilat`, `tulosta_lampotilat` ja `laske_keskiarvo` että muuttujana `main`-funktiossa. Kysymys ei olisi kuitenkaan yhdestä muuttujasta, vaan kolmesta eri parametrista ja yhdestä muuttujasta, joilla kaikilla vain on sama

nimi. Tieto käsiteltävästä listasta välittyisi edelleen parametrien ja funktioiden paluuarvojen välityksellä. Tässä monisteessa on kuitenkin pyritty aluksi käyttämään eri parametreille eri nimiä, jotta aloittelijan olisi helpompi nähdä, milloin on kysymys samasta ja milloin eri muuttujasta.

Tarkastellaan seuraavaksi esimerkkiä hieman monimutkaisemmasta listan läpikäynnistä. Haluamme selvittää, kuinka moni listassa olevista lämpötiloista on vähintään yhtä suuri kuin käyttäjän antama raja. (Tällaista toimintoa voidaan käyttää esimerkiksi hellepäivien määrän selvittelyyn.) Kirjoitetaan oma funktio `montako_yli_rajan`, joka käy sille parametrina annetun listan läpi ja tarkistaa jokaisen listassa olevan lämpötilan kohdalla, onko kyseinen lämpötila suurempi tai yhtäsuuri kuin funktiolle toisena parametrina annettu raja. Tarkistus onnistuu siten, että kirjoitamme toistokäskyn sisään `if`-käskyn, joka tekee tarvittavan tarkastuksen jokaisella kierroksella. Lisäksi tarvitaan laskuri, joka pitää kirjaa siitä, kuinka monta tarpeeksi suurta lämpötilaa on jo kohdattu. Tämä laskuri alustetaan aluksi nollaksi ja sitä kasvatetaan aina, kun listasta löydetään annettua rajaa suurempi tai yhtäsuuri luku. Kun koko lista on käyty läpi, funktio palauttaa laskurin arvon.

```
def montako_yli_rajan(lampotilojen_lista, raja):
    ylittavien_maara = 0
    for asteet in lampotilojen_lista:
        if asteet >= raja:
            ylittavien_maara += 1
    return ylittavien_maara
```

Arvon palauttava `return`-käsky on kirjoitettava toistokäskyn ulkopuolelle. Jos käsky olisi for-käskyn sisällä, arvo palautettaisiin (ja funktion suoritus lopetettaisiin) jo siinä vaiheessa, kun listan ensimmäinen alkio on tutkittu ja listan loppuosaa ei ole vielä käyty lainkaan läpi.

Funktiota voidaan käyttää esimerkiksi lisäämällä pääohjelmaan seuraavat käskyt:

```
lamporaja = float(input("Anna raja, jonka ylittavat lampotilat lasketaan: "))
paiva_lkm = montako_yli_rajan(lampolista, lamporaja)
print("Raja ylittyi", paiva_lkm, "paivana.")
```

Toinen tyypillinen esimerkki listan läpikäynnistä on tilanne, jossa halutaan etsiä listasta parhaiten jonkin ehdon täyttävä arvo, esimerkiksi listan korkein lämpötila. Tällöin listaa läpikäydessä käytetään apumuuttujaa, johon tallennetaan korkein tähän asti löydetty lämpötila. Ennen läpikäyntiä apumuuttujan arvoksi sijoitetaan joko listan ensimmäinen alkio (jos tiedetään, että lista ei ole tyhjä) tai sitten niin pieni arvo, että se ei voi olla minkään päivän lämpötila. Sitten käydään lista läpi alkio kerrallaan. Jos tarkasteltavan alkion arvo on suurempi kuin apumuuttujan arvo, päivitetään apumuuttujan arvo. Kun lista on käyty kokonaan läpi, apumuuttuja sisältää koko listan korkeimman lämpötilan. Alla esimerkkinä funktio `etsi_maksimi`, joka käy sille parametrina annetun listan läpi ja etsii ja palauttaa korkeimman siitä löytyneen lämpötilan. Jos lista on tyhjä, funktio palauttaa erikoisarvon `None`.

```
def etsi_maksimi(lampotilalista):
    if len(lampotilalista) == 0:
        return None
    else:
        maksimi = lampotilalista[0]
        for lampotila in lampotilalista:
            if lampotila > maksimi:
                maksimi = lampotila
        return maksimi
```

Funktiota voidaan kutsua ja sen paluuarvo tulosta lisäämällä pääohjelmaan esimerkiksi seuraavat rivit:

```
maksimilampotila = etsi_maksimi(lampolista)
print("Korkein lampotila on ", maksimilampotila)
```

5.1.2 Haku listasta

Hyvin usein ohjelmia kirjoitettaessa pitää tutkia, löytyykö jokin haluttu arvo listasta ja jos löytyy, niin miltä indeksiltä. Tarkastellaan esimerkkinä tapausta, jossa listaan on tallennettu pelaajien eräästä pelistä saamat pisteet (kokonaislukuja). Halutaan selvittää, onko jollain pelaajalla määrätty pistemäärä ja jos on, niin mille listan indeksille tämä pistemäärä on tallennettu. Tässä kappaleessa katsotaan, miten haku voidaan toteuttaa.

Peräkkäishaku

Jos luvut on tallennettu listaan mielivaltaisessa järjestyksessä, ei ole sen tehokkaampaa vaihtoehtoa kuin käydä listan alkiot järjestyksessä läpi ja verrata jokaista listan alkiota etsittävään arvoon. Läpikäyntiä jatketaan niin kauan, kunnes etsitty arvo löytyy tai lista on käsitelty loppuun asti. Tällaista menettelyä kutsutaan *peräkkäishauksi*.

Alla on esimerkkiohjelma, jonka funktio `hae_pistemaara` etsii ensimmäisenä parametrina annetusta listasta toisena parametrina annetun arvon ja palauttaa sen indeksin listassa. Jos etsittyä arvoa ei löydy lainkaan, funktio palauttaa arvon -1 kertomaan sen, että arvoa ei löytynyt.

Esimerkkiohjelmassa on lisäksi funktio, jolla luetaan pisteet listaan, ja pääohjelma.

```
def lue_pisteet():
    lkm = int(input("Monenko pelaajan pisteet haluat antaa? "))
    pistelista = [0] * lkm
    i = 0
    while i < len(pistelista):
        pistelista[i] = int(input("Anna seuraavan pelaajan pisteet: "))
        i += 1
    return pistelista
```

```
def hae_pisteet(lista, arvo):
    for i in range(0, len(lista)):
        if lista[i] == arvo:
            return i
    return -1

def main():
    pisteet = lue_pisteet()
    haettava_arvo = int(input("Mita arvoa etsitaan? "))
    paikka = hae_pisteet(pisteet, haettava_arvo)
    if paikka == -1:
        print("Haettavaa arvoa ei loytynyt pistelistasta")
    else:
        print("Haettava arvo on listassa indeksilla", paikka)

main()
```

Jos haluttu arvo esiintyy listassa useamman kerran, esimerkin funktio etsii niistä vain ensimmäisen ja palauttaa sen indeksin. Jos halutaan hakea kaikki arvon esiintymät, voidaan sopivat indeksit kerätä esimerkiksi listaan ja palauttaa tämä lista. Näin on tehty seuraavassa esimerkissä.

```
def lue_pisteet():
    lkm = int(input("Monenko pelaajan pisteet haluat antaa? "))
    pistelista = [0] * lkm
    i = 0
    while i < len(pistelista):
        pistelista[i] = int(input("Anna seuraavan pelaajan pisteet: "))
        i += 1
    return pistelista

def hae_pisteet_kaikki(lista, arvo):
    tuloslista = []
    for i in range(0, len(lista)):
        if lista[i] == arvo:
            tuloslista.append(i)
    return tuloslista

def main():
    pisteet = lue_pisteet()
    haettava_arvo = int(input("Mita arvoa etsitaan? "))
    tuloslista = hae_pisteet_kaikki(pisteet, haettava_arvo)
    if len(tuloslista) == 0:
        print("Haettavaa arvoa ei loytynyt pistelistasta")
    else:
        print("Haettava arvo on listassa seuraavilla indekseilla")
        print(tuloslista)

main()
```

Pythonissa on myös valmiita rakenteita, joiden avulla voidaan tutkia, onko haluttu alkio annetussa listassa ja millä listan indeksillä alkio on. Operaattorin `in` avulla voidaan tutkia, onko haluttu alkio listassa, esimerkiksi ensimmäisen peräkkäishakuesimerkin pääohjelmassa olisi voitu kirjoittaa myös

```
if haettava_arvo in pisteet:
    print("Haettava arvo on pistelistassa")
else:
    print("Haettava arvo ei ole pistelistassa")
```

tai vaihtoehtoisesti

```
if haettava_arvo not in pisteet:
    print("Haettava arvo ei ole pistelistassa")
else:
    print("Haettava arvo on pistelistassa")
```

Haettavan alkion indeksin voi selvittää metodin `index` avulla. Kutsussa metodille annetaan käsiteltävä listamuuttuja jo ennen metodin nimeä. Listamuuttuja ja metodi erotetaan toisistaan pisteellä. Mahdolliset parametrit annetaan metodin nimen jälkeen sulkujen sisällä aivan samalla tavalla kuin funktioita kutsuttaessa.

Esimerkin aikaisemmin esitetyssä yhden arvon hakevan ohjelman pääohjelmassa voitaisiin kutsua `index`-metodia esimerkiksi seuraavasti sen sijaan, että käytetään omaa `hae_pisteet`-metodia:

```
paikka = pisteet.index(haettava_arvo)
```

Metodin `index` kutsuminen johtaa kuitenkin virhetilanteeseen ja ohjelman kaatumiseen siinä tapauksessa, että parametrina annettua arvoa ei löydy lainkaan listasta. Sen vuoksi on ensin syytä tarkistaa, onko haettava arvo listassa lainkaan, ja vasta tämän jälkeen kannattaa kutsua `index`-metodia, esimerkiksi seuraavasti:

```
if haettava_arvo not in pisteet:
    print("Haettava arvo ei ole pistelistassa")
else:
    paikka = pisteet.index(haettava_arvo)
    print("Haettava arvo on listassa indeksilla", paikka)
```

Vaikka Pythonin `in`-operaattoria ja `index`-metodia käytettäessä haku saadaan toteutettua yhdellä rivillä, ei näin tehty haku ole olennaisesti sen tehokkaampi kuin edellä kirjoitetun oman `hae_pisteet`-funktion käyttäminen. Pythonin `in`-operaattori ja `index`-metodi joutuvat käymään annetun listan läpi alkio kerrallaan ihan samalla tavalla kuin edellä esitetty oma funktio. Läpikäynti on vain piilotettu Pythonin omiin rakenteisiin eikä näy suoraan ohjelman lukijalle.

Binäärihaku

Jos ennakolta tiedetään, että listassa olevat alkiot ovat suuruusjärjestyksessä (joko pienimmästä suurimpaan tai päinvastoin), voidaan haku suorittaa paljon peräkkäishakua tehokkaammin.

Tarkastellaan tilannetta, jossa pisteet on tallennettu listaan nousevassa suuruusjärjestyksessä. Otetaan listan keskimmäinen alkio ja verrataan haettavaa arvoa siihen. Jos keskimmäinen alkio on suurempi, haettavan arvon on pakko olla listan alkuosassa (jos se on listassa). Vastaavasti jos keskimmäinen alkio on pienempi kuin haettava arvo, haettavan arvon on pakko olla listan loppuosassa.

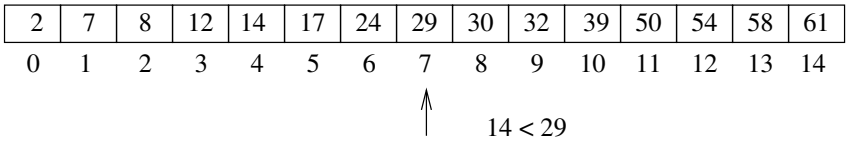
Oletetaan, että keskimmäinen arvo on suurempi kuin haettava arvo. Siinä tapauksessa voidaan hakua jatkaa listan alkuosasta vastaavalla menetelmällä. Otetaan nyt alkuosan keskimmäinen arvo ja verrataan haettavaa arvoa siihen. Jos haettava arvo on pienempi, hakua voidaan jatkaa vastaavalla menetelmällä listan alkuosan alkuosasta (siis alkuperäisen listan ensimmäisestä neljänneksestä), ja jos haettava arvo on suurempi, hakua voidaan jatkaa listan alkuosan loppuosasta (siis alkuperäisen listan toisesta neljänneksestä). Näin jatketaan, kunnes tarkasteltavan hakualueen keskimmäinen alkio on sama kuin haettava arvo (jolloin haluttu arvo on löytynyt) tai hakualue on tyhjä (joilloin voidaan todeta, että haluttua arvoa ei ole listassa lainkaan).

Tällaista hakutapaa kutsutaan *binäärihauksi*. Kirjoitetaan binäärihaun periaate vähän täsmällisemmin:

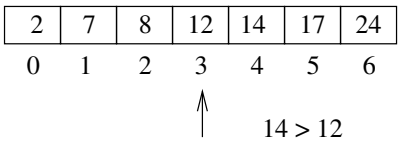
1. Aloita niin, että hakualueena on koko alkuperäinen lista.
2. Jos hakualue on tyhjä eli hakualueen alaindeksi on suurempi kuin hakualueen yläindeksi, lopeta haun suoritus. Haettavaa arvoa ei ole listassa lainkaan. Muussa tapauksessa laske hakualueen keskimmäisen alkion indeksi. (Keskimmäinen tarkoittaa tässä sitä alkiota, joka on hakualueen keskimmäisessä paikassa.)
3. Vertaa haettavaa arvoa hakualueen keskimmäiseen alkioon:
   (a) Jos haettava arvo ja keskimmäinen alkio ovat yhtäsuuret, haettava arvo on löytynyt. Lopeta haun suoritus ja palauta keskimmäisen alkion indeksi.
   (b) Jos haettava arvo on pienempi kuin hakualueen keskimmäinen alkio, jatka kohdasta 2, mutta niin, että hakualueena on nykyisen hakualueen alkupuolisko.
   (c) Jos haettava arvo on suurempi kuin hakualueen keskimmäinen alkio, jatka kohdasta 2, mutta niin, että hakualueena on nykyisen hakualueen loppupuolisko

Seuraavassa kuvassa on vielä esitetty, miten binäärihaku etenee, kun kuvassa olevasta listasta haetaan arvoa 14. Listan alapuolelle on merkitty alkioiden indeksit. Eri vaiheissa listasta on esitetty vain se osa, joka vielä kuuluu hakualueseen. Hakualueen keskimmäinen alkio on merkitty nuolella.

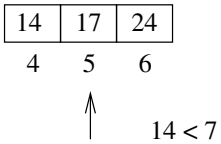
1. kierros



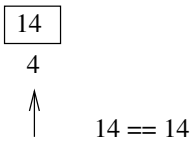
2. kierros



3. kierros



4. kierros



Seuraava funktio hakee binäärihaulla ensimmäisenä parametrina annetusta listasta toisena parametrina annetun arvon. Funktion sisällä pidetään muuttujien `ala` ja `yla` avulla tietoa siitä, mikä on hakualue milläkin hetkellä. Aluksi `ala`-muuttujan arvoksi asetetaan nolla ja `yla`-muuttujan arvoksi listan viimeisen alkion indeksi. Joka kierroksella lasketaan aina hakualueen keskimmäinen indeksi, jonka arvo tallennetaan muuttujaan `keski`. Tämän jälkeen verrataan haettavaa arvoa hakualueen keskimmäisellä indeksillä olevaan alkioon. Vertailun tuloksen mukaan joko poistutaan funktiosta tai muutetaan hakualueen ylä- tai alaindeksin arvoa.

```
def binaarihaku(lista, arvo):
    ala = 0
    yla = len(lista) - 1
    while ala <= yla:
        keski = (ala + yla) // 2
        if lista[keski] == arvo:  # arvo loytyi, lopetetaan
            return keski
        elif lista[keski] < arvo:
            ala = keski + 1  # jatketaan hakua loppuosasta
        else:
            yla = keski - 1  # jatketaan hakua alkuosasta
    return -1   # hakualue on tyhja, mutta arvoa ei loytynyt
```

Annettu funktio toimii oikein vain siinä tapauksessa, että parametrina annetussa listassa luvut ovat suuruusjärjestyksessä. Funktio ei mitenkään tarkista, pitääkö tämä paikkansa. Jos listan alkiot ovat satunnaisessa järjestyksessä, funktion paluuarvo voi olla täysin väärä eikä funktio anna mitään ilmoitusta siitä, että se ei toiminut oikein.

Binäärihaku on hyvin tehokas verrattuna peräkkäishakuun. Tarkastellaan esimerkiksi hakua listasta, jossa on miljoona alkiota. Peräkkäishakua käytettäessä joudutaan pahimmassa tapauksessa suoritamaan funktiossa `hae_pisteet` olevaa `for`-käskyä miljoona kierrosta (kaikki listan alkiot pitää käydä läpi, jos haettava alkio ei ole listassa tai on siinä viimeisellä indeksillä). Binäärihakua käytettäessä riittää, että `while`-käskyä suoritetaan pahimmassakin tapauksessa noin 20 kierrosta. Vielä paremmin ero tulee näkyviin, kun mietitään tilannetta, jossa listan koko kasvaa kahteen miljoonaan. Peräkkäishakua käytettäessä pahimmassa tapauksessa tarvittavien `for`-käskyn kierrosten määrä kasvaa miljoonalla, kun taas binäärihaussa `while`-käskyn kierrosten määrä kasvaa vain yhdellä.

Alla on vielä esimerkkinä koko ohjelma, jossa on ensin pyydetty käyttäjää antamaan pelipisteet suuruusjärjestyksessä ja sen jälkeen on käytetty `binaarihaku`-funktiota halutun arvon hakemiseen.

```
def lue_pisteet():
    print("Anna pelaajien pisteet suuruusjarjestyksessa pienimmasta alkaen.")
    lkm = int(input("Monenko pelaajan pisteet haluat antaa? "))
    pistelista = [0] * lkm
    i = 0
    while i < len(pistelista):
        pistelista[i] = int(input("Anna seuraavan pelaajan pisteet: "))
        i += 1
    return pistelista

def binaarihaku(lista, arvo):
    ala = 0
    yla = len(lista) - 1
    while ala <= yla:
        keski = (ala + yla) // 2
        if lista[keski] == arvo:  # arvo loytyi, lopetetaan
            return keski
        elif lista[keski] < arvo:
            ala = keski + 1  # jatketaan hakua loppuosasta
        else:
            yla = keski - 1  # jatketaan hakua alkuosasta
    return -1   # hakualue on tyhja, mutta arvoa ei loytynyt


def main():
    pisteet = lue_pisteet()
    haettava_arvo = int(input("Mita arvoa etsitaan? "))
    paikka = binaarihaku(pisteet, haettava_arvo)
    if paikka == -1:
        print("Haettavaa arvoa ei loytynyt pistelistasta")
```

```
    else:
        print("Haettava arvo on listassa indeksilla", paikka)

main()
```

5.1.3 Muita listan käsittelyyn tarkoitettuja funktioita ja metodeita

Pythonissa on valmiina lukuisia listojen käsittelyyn tarkoitettuja funktioita ja metodeita, eikä seuraavassa suinkaan esitellä niistä kaikkia, vaan vain joitakin tärkeimpiä. Seuraavissa esimerkeissä ei ole esitetty kokonaisia ohjelmia, vaan metodien ja funktioiden toimintaa on havainnollistettu esimerkeillä, joissa Python-tulkille on annettu yksittäisiä käskyjä.

Kuten edellä on kerrottu, listan `lista` i:nnen alkion arvon saa kirjoittamalla `lista[i]`, kunhan `i` on listan indeksialueen sisällä. Listasta voi kuitenkin ottaa myös usemman peräkkäisen alkion *alilistoja*. Jos on esimerkiksi määritelty:

```
>>> lista = [2, 4, 6, 8, 10, 12, 14, 16]
```

niin `lista[2:5]` palauttaa listan, joka sisältää alkuperäisen listan indekseillä 2, 3, ja 4 olevat alkiot, siis listan `[6, 8, 10]`. Merkintä `lista[2:5]` siis tarkoittaa sitä, että otetaan mukaan alkiot indeksistä 2 indeksiin 5 asti mutta niin, että indeksillä 5 olevaa alkiota ei enää oteta mukaan.

Lausekkeessa voidaan jättää joko alku- tai loppuindeksi merkitsemättä. Esimerkiksi `lista[:5]` palauttaa listan, joka sisältää alkuperäisen listan alkiot listan alusta indeksille 5 asti (mutta ei enää indeksillä 5 olevaa alkiota). Vastaavasti taas `lista[5:]` palauttaa listan, joka sisältää alkuperäisen listan alkiot indeksiltä 5 lähtien (se mukaanlukien) listan loppuun asti.

Negatiivisilla indekseillä tarkoitetaan alkioita listan lopusta alkaen. Esimerkiksi `lista[-1]` palauttaa listan viimeisen alkion ja `lista[:-1]` palauttaa listan, joka sisältää kaikki alkuperäisen listan alkiot viimeistä alkiota lukuunottamatta.

Kuten jo aikasemmin on kerrottu, funktio `len` palauttaa sille parametrina annetun listan pituuden, esimerkiksi

```
>>> len(lista)
8
```

Metodin `append` avulla voidaan lisätä listan loppuun metodille parametrina annettu alkio, esimerkiksi

```
>>> lista.append(18)
>>>> lista
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Metodin `insert` avulla voidaan lisätä listaan ensimmäisenä parametrina annetulle indeksille toisena parametrina annettu alkio, esimerkiksi

```
>>> lista.insert(2, 7)
>>> lista
[2, 4, 7, 6, 8, 10, 12, 14, 16, 18]
```

Ero sijoitukseen `lista[2] = 7` on siinä, että `insert`-metodi lisää uuden alkion listassa jo olevien alkioiden väliin ja siirtää listan lopussa olevia alkioita yhden indeksin eteenpäin, kun taas sijoituskäsky korvaa indeksillä 2 olevan alkion uudella alkiolla. Metodin `insert` suorituksen yhteydessä listan pituus siis kasvaa, kun taas sijoituskäsky ei muuta listan pituutta.

Metodi `remove` poistaa listasta ensimmäisen alkion, jolla on parametrina annettu arvo, esimerkiksi

```
>>> lista.remove(10)
>>> lista
[2, 4, 7, 6, 8, 12, 14, 16, 18]
```

Operaattorin `del` avulla voidaan taas poistaa listasta halutulla indeksillä oleva alkio, esimerkiksi

```
>>> del lista[3]
>>> lista
[2, 4, 7, 8, 12, 14, 16, 18]
```

poisti edellisen esimerkin listasta indeksillä 3 olevan alkion.

Kuten aikaisemmin esitettiin, metodi `index` palauttaa sille parametrina annetun alkion ensimmäisen esiintymän listassa, esimerkiksi

```
>>> lista.index(6)
3
```

Operaattorin `in` avulla voidaan taas selvittää, onko jokin alkio listassa:

```
>>> 8 in lista
True
>>> 9 in lista
False
```

Metodi `sort` järjestää listan, esimerkiksi

```
>>> lista.sort()
>>> lista
[2, 4, 6, 7, 8, 12, 14, 16, 18]
```

Vaikka järjestäminen voidaankin suorittaa yhden Pythonissa valmiina olevan metodin kutsulla, niin käytännössä tämän valmiin metodin suorittaminen vie yleensä enemmän aikaa kuin listan läpikäynti alkio kerrallaan.

Funktio `sorted` tekee listasta kopion ja järjestää tämän kopion. Erona metodiin `sort` on se, että funktiota `sorted` käytettäessä alkuperäisessä listassa alkioiden järjestys jää ennalleen ja vain uusi kopio järjestetään. Koska `sorted` on funktio eikä metodi, sitä kutsutaan eri tavalla kuin metodia `sort`:

```
>>> lista1 = [19, 3, 7, 2, 11]
>>> lista2 = sorted(lista1)
>>> lista2
[2, 3, 7, 11, 19]
>>> lista1
[19, 3, 7, 2, 11]
```

Metodi `reverse` kääntää listan järjestyksen päinvastaiseksi. Metodi muokkaa alkuperäistä listaa. Se ei siis luo uutta listaa, jossa olisi uusi järjestys, esimerkiksi

```
>>> lista.reverse()
>>> lista
[18, 16, 14, 12, 8, 7, 6, 4, 2]
```

Välillä tulee vastaan tilanteita, joissa pitäisi kopioida yhden listan alkioiden arvot toisen listan alkioiden arvoksi. Tarkastellaan esimerkkiä: Muuttuja `lista1` viittaa listaan `[1, 2, 3]`. Halutaan ottaa käyttöön uusi muuttuja `lista2`, ja saada se viittaamaan listaan, joka sisältää samat luvut. Pelkkä sijoituskäskyn käyttäminen voi johtaa yllättävään lopputulokseen, kuten seuraava Python-tulkissa suoritettu esimerkki näyttää:

```
>>> lista1 = [1, 2, 3]
>>> lista2 = lista1
>>> lista1[1] = 5
>>> print(lista1)
[1, 5, 3]
>>> print(lista2)
[1, 5, 3]
```

Huomataan, että kun `lista1`n toista alkoita muutettiin, niin samalla myös `lista2`n alkio muuttui. Tämä johtuu siitä, että molemmat muuttujat, `lista1` ja `lista2` viittaavat samaan listaan. Vaikka ohjelmassa käytetäänkin kahta eri muuttujaa, niin ohjelman käytössä on vain yksi lista, jonka alkioita voidaan muuttaa kumman tahansa muuttujan kautta.

Jos tarkoituksena on tehdä alkuperäisestä listasta kopio (minkä jälkeen ohjelman käytössä on kaksi toisistaan erillistä listaa), voi sen lukuja (ja myös merkkijonoja tai totuusarvoja) sisältävien listojen tapauksessa tehdä esimerkiksi seuraavasti:

```
>>> lista1 = [1, 2, 3]
>>> lista2 = [] + lista1
>>> lista1[1] = 5
>>> print(lista1)
[1, 5, 3]
>>> print(lista2)
[1, 2, 3]
```

Tässä `lista2`n viittama lista on muodostettu luomalla ensin uusi tyhjä lista ja lisäämällä siihen sitten `lista1`n sisältämät alkiot. Ohjelman käytössä on nyt kaksi eri listaa (vaikka ne aluksi sisältävätkin yhtäsuuret alkiot), joten muutos ensimmäiseen listaan ei vaikuta mitenkään toisen listan sisältöön. Toki olisi ollut mahdollista myös kirjoittaa toistokäsky, joka olisi kopioinut `lista1`n alkiot uuteen listaan yksi kerrallaan.

Tilannetta, jossa useampi muuttuja viittaa samaan listaan, on käsitelty tarkemmin kappaleessa 5.5.

5.1.4 Moniulotteiset listat

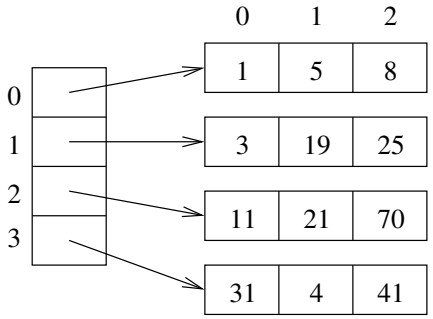
Tarkastellaan ongelmaa, jossa halutaan esittää matriiseja Python-ohjelmassa. Matriisi koostuu riveistä ja sarakkeista, esimerkiksi

$$\begin{pmatrix} 1 & 5 & 8 \\ 3 & 19 & 25 \\ 11 & 21 & 70 \\ 31 & 4 & 41 \end{pmatrix}$$

Tässä matriisissa on siis neljä riviä ja kolme saraketta. Matriiseja tarvitaan hyvin yleisesti esimerkiksi erilaisissa tekniikan alan laskutoimituksissa.

Miten sitten matriisia voidaan käsitellä Python-ohjelmassa? On toki mahdollista esittää kaikki matriisin alkiot yhtenä listana eli esimerkkimatriisi listana `[1, 5, 8, 3, 19, 25, 11, 21, 70, 31, 4, 41]`. Tässä listassa ei kuitenkaan näy mitenkään matriisin rakenne eli sen jako riveihin ja sarakkeisiin. Matriisien laskutoimituksissa on tärkeä merkitys sillä, millä rivillä ja missä sarakkeessa kukin alkio on. Vaikka tämän pystyykin laskemaan alkion indeksistä, olisi kätevämpää, jos indeksit kertoisivat suoraan alkion rivin ja sarakkeen matriisissa.

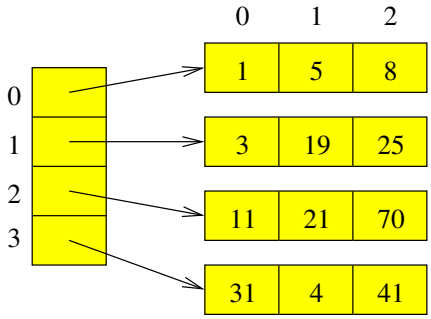
Parempi vaihtoehto on käyttää kaksiulotteista listaa. Siinä listan jokaista riviä vastaa yksi listan alkio. Tämä alkio ei kuitenkaan ole yksittäinen luku, vaan toinen lista, jonka alkioina on riville kuuluvat luvut. Esimerkkimatriisi esitettäisiin listana `[[1, 5, 8], [3, 19, 25], [11, 21, 70], [31, 4, 41]]`. Rakenteen voi ajatella näyttävän suunnilleen alla olevan kuvan mukaiselta (kuvaan on merkitty myös listojen indeksejä niiden seuraamisen helpottamiseksi):



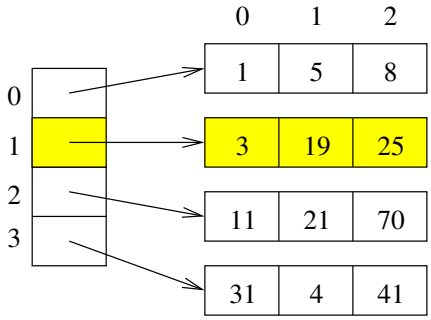
Jos ohjelmassa on suoritettu käsky

```
matriisi = [[1, 5, 8], [3, 19, 25], [11, 21, 70], [31, 4, 41]]
```

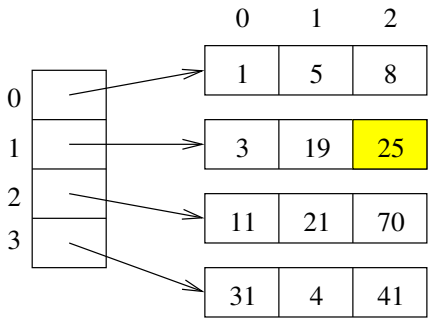
viittaa nyt `matriisi` koko esitettyyn kaksiulotteiseen listaan:



Sen sijaan `matriisi[1]` tarkoittaa listan toista alkiota, siis toisena alkiona olevaa pikkulistaa.



Merkinnässä `matriisi[1][2]` ensimmäisten hakasulkujen sisällä oleva indeksi tarkoittaa indeksiä varsinaisessa (ulommassa) listassa, toisten hakasulkujen sisällä oleva indeksi taas tarkoittaa indeksiä ulomman listan alkiona olevassa pikkulistassa:



Alla on esitetty esimerkkejä eri merkintöjen tarkoituksesta:

```
>>> print(matriisi)
[[1, 5, 8], [3, 19, 25], [11, 21, 70], [31, 4, 41]]
>>> print(matriisi[3])
[31, 4, 41]
>>> print(matriisi[2])
[11, 21, 70]
>>> print(matriisi[2][1])
21
```

Seuraava esimerkkiohjelma lukee käyttäjältä kaksi matriisia ja laskee niiden summan. Ohjelmassa on omat funktionsa yhden matriisin lukemiseen, kahden matriisin summamatriisin laskemiseen (summa lasketaan laskemalla aina yhteenlaskettavien matriisien samassa paikassa olevat alkiot yhteen) ja matriisin tulostamiseen riveittäin. Huomaa, että pääohjelmassa pidetään huolta siitä, että matriisien koko on järkevä ja että yhteenlaskettavat matriisit ovat samankokoisia. Jos tästä ei huolehdittaisi pääohjelmassa, pitäisi esimerkiksi funktioon `laske_summa` lisätä tarkistus yhteenlaskettavien matriisien koosta.

Matriisien käsittelyssä käytetään usein kahta sisäkkäistä toistokäskyä, joista ulompi käy läpi kaikki matriisin rivit ja sisempi yhden rivin kaikki sarakkeet.

```
# Funktio lukee kayttajalta matriisin alkiot. Matriisin
# rivien ja sarakkeiden maara annetaan parametreina.
# Funktio luo matriisia varten kaksiulotteisen listan,
# tallentaa luetut luvut siihen ja palauttaa alkiot
# sisaltavan listan.

def lue_matriisi(rivilkm, sarakelkm):
    matriisi = []
    print("Anna matriisin alkiot riveittain,")
    print(rivilkm, "rivia ja", sarakelkm, "saraketta.")
    for i in range(rivilkm):
        rivi = [0.0] * sarakelkm
        for j in range(sarakelkm):
            rivi[j] = float(input())
        matriisi.append(rivi)
    return matriisi


# Funktio saa parametrina kaksi matriisia, jotka on tallennettu
# kaksiulotteisiin listoihin. Funktio laskee naiden matriisien
# summamatriisin ja palauttaa sita vastaavan kaksiulotteisen listan.

def laske_summa(mat1, mat2):
    summamat = []
    rivimaara = len(mat1)  # ulomman listan alkioiden maara
    sarakemaara = len(mat1[0]) # pikkulistan alkioiden maara
    for i in range(rivimaara):
        summarivi = [0.0] * sarakemaara
        for j in range(sarakemaara):
            summarivi[j] = mat1[i][j] + mat2[i][j]
        summamat.append(summarivi)
    return summamat


# Funktio saa parametrina matriisia esittavan kaksiulotteisen
# listan. Se tulostaa matriisin alkiot riveittain.
```

```
def tulosta_matriisi(matri):
    rivit = len(matri)
    sarakkeet = len(matri[0])
    for i in range(rivit):
        tulosrivi = ""
        for j in range(sarakkeet):
            tulosrivi += "{:8.2f}".format(matri[i][j])
        print(tulosrivi)


def main():
    print("Ohjelma laskee kahden matriisin summan.")
    riveja = int(input("Rivien lukumaara: "))
    sarakkeita = int(input("Sarakkeiden lukumaara: "))
    if riveja <= 0 or sarakkeita <= 0:
        print("Liian vahan riveja tai sarakkeita.")
    else:
        matriisi1 = lue_matriisi(riveja, sarakkeita)
        matriisi2 = lue_matriisi(riveja, sarakkeita)
        summa = laske_summa(matriisi1, matriisi2)
        print("Matriisin")
        tulosta_matriisi(matriisi1)
        print("ja matriisin")
        tulosta_matriisi(matriisi2)
        print("summa on")
        tulosta_matriisi(summa)


main()
```

5.2 Merkkijono

Ohjelmissa pitää usein käsitellä erilaista teksitietoa, esimerkiksi nimiä, osoitteita sekä erilaisia tunnuksia, kuten opiskelijanumeroita ja autojen rekisterinumeroita. Näitä voidaan käsitellä kätevästi *merkkijonojen* avulla. Merkkijono koostuu yhdestä tai useammasta peräkkäisestä merkistä. Merkkijono voi myös olla tyhjä, jolloin siinä ei ole yhtään merkkiä.

Merkkijono esitetään yksin- tai kaksinkertaisten lainausmerkkien avulla. Seuraavat kaksi sijoituskäskyä

```
>>> mjono = 'appelsiini'
>>> mjono = "appelsiini"
```

tarkoittavat täysin samaa.

Toisin kuin monissa muissa ohjelmointikielissä, Pythonissa ei eroteta toisistaan yksittäisiä merkkejä ja merkkijonoja. Myös yksittäiset merkit esitetään aina yhden merkin mittaisina merkkijonoina.

Pythonissa on tyyppi `str` merkkijonojen esittämiseen. Merkkijonojen käsittely Python-ohjelmissa muistuttaa hyvin paljon listojen käsittelyä. Olennaisin ero on siinä, että listan sisältöä voidaan muuttaa listan luomisen jälkeen, mutta merkkijonon sisältöä ei voida. Esimerkiksi seuraavat Python-rivit ovat täysin mahdollisia:

```
>>> lukulista = [5, 2, 7]
>>> lukulista[1] = 4
>>> lukulista
[5, 4, 7]
```

Mutta sen sijaan seuraavat rivit aiheuttavat virhetilanteen, koska aikaisemmin luodun merkkijonon sisältöä ei voi muuttaa:

```
>>> sana = "sitruuna"
>>> sana[1] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Merkkijonoja voidaan kuitenkin muuten käsitellä monilla samoilla tavoilla kuin listoja. Esimerkiksi merkkijonon `mjono` indeksillä `i` olevan kirjaimen saa selville ilmauksella `mjono[i]`, vaikka kirjainta ei pystykään vaihtamaan sijoituskäskyllä, esimerkiksi

```
>>> mjono = "appelsiini"
>>> print(mjono[3])
e
```

Merkkijonojen merkit voidaan myös käydä läpi `for`-käskyn avulla samalla tavalla kuin listan merkit. Esimerkiksi

```
>>> mjono = "appelsiini"
>>> for merkki in mjono:
...     print(merkki)
...
a
p
p
e
l
s
i
i
n
i
```

Merkkijonosta voidaan myös ottaa alimerkkijonoja samalla tavalla kuin listoista antamalla hakasulkujen sisässä indeksialue, esimerkiksi

```
>>> mjono = "appelsiini"
>>> print(mjono[3:7])
elsi
```

Ennen kaksoispistettä oleva luku kertoo jälleen ensimmäisen alimerkkijonoon otettavan indeksin ja kaksoispisteen jälkeen oleva luku ensimmäisen indeksin, jota ei oteta mukaan alimerkkijonoon.

Merkkijonon pituuden saa selville funktiolla len:

```
>>> mjono = "appelsiini"
>>> print(len(mjono))
10
```

Operaattorilla `in` pystyy tutkimaan sitä, esiintykö kirjain merkkijonossa ja metodilla `index` saa selville parametrina annetun merkin ensimmäisen indeksin merkkijonossa:

```
>>> mjono = "appelsiini"
>>> if "i" in mjono:
...     print("i esiintyy sanassa")
...     print("indeksilla", mjono.index("i"))
...
i esiintyy sanassa
indeksilla 6
```

Operaattorin `in` avulla voi tutkia myös sitä, esiintyykö yhtä kirjainta pitempi merkkijono osana toista merkkijonoa:

```
>>> mjono = "appelsiini"
>>> print("elsii" in mjono)
True
>>> print("aelsi" in mjono)
False
```

Vaikka itse merkkijonoa ei voikaan muuttaa sen jälkeen, kun se on luotu, niin merkkijonoon viittaava muuttuja voidaan sijoituskäskyllä panna viittaamaan kokonaan toiseen merkkijonoon. Esimerkiksi seuraavat ohjelmarivit ovat täysin sallittuja:

```
>>> mjono = "appelsiini"
>>> print(mjono)
appelsiini
>>> mjono = "apelsiini"
>>> print(mjono)
apelsiini
```

Tässä alkuperäinen merkkijono `"appelsiini"` ei kuitenkaan muutu miksikään, vaan luodaan kokonaan uusi merkkijono `"apelsiini"`, ja muuttuja `mjono` pannaan viittaamaan luotuun uuteen merkkijonoon.

Uusi merkkijono voi riippua vanhasta merkkijonosta. Pythonissa on metodeita, joilla voidaan luoda kokonaan uusi, mutta vanhaa merkkijonoa muistuttava merkkijono. Aikaisempaan merkkijonoon viitannut muuttuja voidaan silloin sijoituskäskyllä panna viittaamaan luotuun, kokonaan uuteen merkkijonoon. Esimerkiksi metodi `lower` luo uuden merkkijonon, joka sisältää muuten samat merkit kuin nykyinen merkkijono, mutta kaikki isot kirjaimet on muutettu pieniksi kirjaimiksi.

```
>>> mjono = "AppelSiIni"
>>> mjono = mjono.lower()
>>> print(mjono)
appelsiini
```

Tämä metodi on hyvin kätevä silloin, kun pitää lukea käyttäjän syötettä ja verrata sitä johonkin muuhun tekstiin. Jos ei tiedetä, antaako käyttäjä syötteen isoina vai pieninä kirjaimina, voidaan syöte muuttaa ensin kokonaan pieniksi kirjaimiksi, jolloin sitä tarvitsee verrata vain pienillä kirjaimilla kirjoitettuun tekstiin.

Vastaavasti metodilla `upper` voidaan luoda uusi merkkijono, joka sisältää muuten samat merkit kuin nykyinen merkkijono, mutta kaikki pienet kirjaimet on muutettu isoiksi kirjaimiksi.

```
>>> mjono = "AppelSiIni"
>>> mjono = mjono.upper()
>>> print(mjono)
APPELSIINI
```

Käyttäjän syötettä lukiessa tarvitaan usein myös metodia `strip`. Se luo uuden merkkijonon, jossa on poistettu nykyisen merkkijonon alussa ja lopussa esiintyvät tyhjät merkit. Tyhjillä merkeillä tarkoitetaan välilyönti-, tabulointi, rivinvaihto- ja sivunvaihtomerkkejä. Tarkoituksena on poistaa luetun merkkijonon alusta ja lopusta mahdolliset käyttäjän vahingossa kirjoittamat välilyönnit ja vastaavat. Alla on esimerkki `strip`-metodin käytöstä. Merkkijonot on nyt tulostettu lainausmerkkien kanssa niin, että näkyy tarkemmin, mistä merkkijono alkaa ja mihin se loppuu.

```
>>> mjono = "           appelsiini   "
>>> mjono
'    \tappelsiini   '
>>> mjono = mjono.strip()
>>> mjono
'appelsiini'
```

Esimerkin merkkijonossa aluksi esiintynyt merkki `"\t"` tarkoittaa tabulointimerkkiä.

Useampi merkkijono voidaan yhdistää yhdeksi `+`-operaattorin avulla, ja merkkijonoja voi jopa monistaa `*`-operaattorilla:

```
>>> etunimi = "Mikko"
>>> sukunimi = "Mallikas"
>>> kokonimi = etunimi + " " + sukunimi
>>> print(kokonimi)
Mikko Mallikas
>>> merkki = "&"
>>> rivi = 5 * merkki
>>> print(rivi)
&&&&&
>>> rivit = 3 * (rivi + "\n")
>>> print(rivit)
&&&&&
&&&&&
&&&&&
```

Yksi käytännössä usein tarvittava merkkijonoja käsittelevä metodi on `split`. Sen avulla voi jakaa merkkijonon useampaan osaan niin, että jako tehdään aina halutun merkin kohdalta. Oletusarvoisesti jako tehdään välilyöntimerkin kohdalla. Jaon tuloksena syntyneet osamerkkijonot voidaan tallentaa listaan seuraavan esimerkin mukaisesti:

```
>>> teksti = "Monta eri sanaa samassa merkkijonossa"
>>> sanat = teksti.split()
>>> print(sanat)
['Monta', 'eri', 'sanaa', 'samassa', 'merkkijonossa']
>>> for yksittainen_sana in sanat:
...     print(yksittainen_sana)
...
Monta
eri
sanaa
samassa
merkkijonossa
```

Jaossa käytetty merkki (tässä tapauksessa välilyönti) ei kuulu yhteenkään jaon tuloksena syntyneeseen osamerkkijonoon. Jos jako halutaan tehdä jonkin muun kuin välilyöntimerkin kohdalta, annetaan tämä merkki parametrina `split`-metodille, esimerkiksi

```
>>> teksti2 = "sanat/erotettu/toisistaan/kauttaviivalla"
>>> sanat2 = teksti2.split("/")
>>> print(sanat2)
['sanat', 'erotettu', 'toisistaan', 'kauttaviivalla']
```

Tässä esimerkissä näkyy myös hyvin se, että listan alkioiden ei tarvitse suinkaan olla lukuja, vaan lista voi sisältää myös esimerkiksi merkkijonoja.

Merkkijonoja voidaan myös vertailla operaattoreilla `==`, `!=`, `<=`, `>=`, `<` ja `>`. Esimerkiksi `mjono1 == mjono2` on tosi, jos muuttujan `mjono1` viittaama merkkijono sisäl-

tää täsmälleen samat merkit (samassa järjestyksessä) kuin muuttujan `mjono2` viittaama merkkijono. Vertailussa pienet ja suuret kirjaimet katsotaan eri merkeiksi. Muut operaattorit käyvät läpi vertailtavia merkkijonoja merkki kerrallaan, kunnes kohdataan ensimmäistä kertaa eri merkit. Tällöin näiden merkkien asema käytetyssä merkkikoodausjärjestelmässä (siinä, miten kukin merkki esitetään tietokoneen muistissa binäärilukuna) ratkaisee sen, kumpi merkkijonoista katsotaan toista pienemmäksi. Käytännössä vertailu menee useimmiten aakkosjärjestyksen mukaan, mutta kaikki isot kirjaimet ovat järjestyksessä ennen pieniä kirjaimia ja skandinaavisten aakkosten (å, ä ja ö) osalta vertailu ei toimi aakkosjärjestyksen mukaisesti.

Esimerkkinä merkkijonojen vertailusta esitetään vielä ohjelma, joka tekee lämpötilamuunnoksia fahrenheit-asteista celsius-asteiksi ja päinvastoin. Ohjelma ensin kysyy käyttäjältä, minä asteina hän haluaa lämpötilan antaa. Ohjelma lukee käyttäjältä muunnettavan lämpötilan ja tekee muunnoksen haluttuun suuntaan. Sen jälkeen ohjelma kysyy käyttäjältä, haluaako hän jatkaa antamalla toisen lämpötilan. Tätä jatketaan niin kauan, kunnes käyttäjä kertoo, että hän ei halua jatkaa.

```
def muunna_celsiuksiksi(F_asteet):
    celsius_asteet = (F_asteet - 32) * 5.0 / 9.0
    return celsius_asteet


def muunna_fahrenheiteiksi(C_asteet):
    fahrenheit_asteet = 9.0/5.0 * C_asteet + 32
    return fahrenheit_asteet

def main():
    jatko = "kylla"
    while jatko != "ei":
        rivi = input("Mina asteina annat lampotilan (C/F)? ")
        yksikko = rivi.upper()
        if yksikko == "C":
            asteet = float(input("Anna lampotila celsius-asteina: "))
            fahrenheit = muunna_fahrenheiteiksi(asteet)
            print(asteet, "C on", fahrenheit, "F.")
        elif yksikko == "F":
            asteet = float(input("Anna lampotila fahrenheit-asteina: "))
            celsius = muunna_celsiuksiksi(asteet)
            print(asteet, "F on", celsius, "C.")
        else:
            print("Virheellinen yksikko, pitaisi olla C tai F")
        rivi = input("Haluatko jatkaa (kylla/ei)? ")
        jatko = rivi.lower()

main()
```

Alla esimerkkiajo ohjelman suorituksesta:

```
Mina asteina annat lampotilan (C/F)? c
Anna lampotila celsius-asteina: 25.0
25.0 C on 77.0 F.
Haluatko jatkaa (kylla/ei)? kylla
Mina asteina annat lampotilan (C/F)? F
Anna lampotila fahrenheit-asteina: -40.0
-40.0 F on -40.0 C.
Haluatko jatkaa (kylla/ei)? kylla
Mina asteina annat lampotilan (C/F)? f
Anna lampotila fahrenheit-asteina: 92.0
92.0 F on 33.3333333333 C.
Haluatko jatkaa (kylla/ei)? Ei
```

Useammalle kuin yhdelle riville jakaantuvaa merkkijonoa voidaan merkitä kolmen lainausmerkin avulla, esimerkiksi

```
>>> teksti = """Hei, opiskelija!
... meidan Tosi on -pankistamme saat
... opintolainat edullisesti"""
>>> print(teksti)
Hei, opiskelija!
meidan Tosi on -pankistamme saat
opintolainat edullisesti
>>> mainos = '''Hei, opiskelija!
... tule suoraan omaan pankkiisi,
... ala vilkuile naapureihin'''
>>> print(mainos)
Hei, opiskelija!
tule suoraan omaan pankkiisi,
ala vilkuile naapureihin
```

Useammasta rivistä koostuva teksti voidaan tehdä ohjelmassa lisäämällä rivinvaihtojen kohtaan `\n`-merkkejä. Esimerkiksi seuraava ohjelma pyytää käyttäjältä henkilön nimi- ja osoitetiedot sekä tekee niistä merkkijonon, joka sisältää myös rivinvaihtoja.

```
def tee_osoite():
    print("Anna seuraavan henkilon tiedot.")
    etunimi = input("Etunimi: ")
    sukunimi = input("Sukunimi: ")
    katuosoite = input("Katuosoite: ")
    postinumero = input("Postinumero: ")
    postitoimipaikka = input("Postitoimipaikka: ")
    osoite = etunimi + " " + sukunimi + "\n" + katuosoite + "\n" + \
        postinumero + " " + postitoimipaikka.upper()
    return osoite
```

```
def main():
    print("Talla ohjelmalla voit syottaa ja tulostaa osoitteita.")
    osoitteet = []
    jatko = True
    while jatko:
        uusi_osoite = tee_osoite()
        osoitteet.append(uusi_osoite)
        vastaus = input("Haluatko antaa lisaa osoitteita (k/e)?\n")
        if vastaus.lower() == "e":
            jatko = False
    print()
    print("Antamasi osoitteet:")
    for osoitetieto in osoitteet:
        print(osoitetieto)
        print()

main()
```

Osoite muodostetaan siten, että nimen ja katuosoitteen sekä katuosoitteen ja postinumeron välille lisätään rivinvaihto. Lisäksi postitoimipaikka muutetaan isoilla kirjaimilla kirjoitetuksi. Ohjelma pyytää käyttäjältä uusia osoitteita ja lisää funktiolla `tee_osoite` muodostetun osoitteen listaan `osoitteet` niin kauan, että käyttäjä ei enää halua antaa uusia osoitteita. Käyttäjän vastaus jatkokysymykseen muutetaan pieniksi kirjaimiksi, jolloin sekä `e` että `E` kelpaavat ohjelman suorituksen lopettaviksi vastauksiksi. Kun käyttäjä on antanut haluamansa määrän osoitteita, listassa olevat osoitteet tulostetaan. Ohjelmassa olevat `print()`-käskyt ilman tulostettavaa tekstiä lisäävät tyhjiä rivejä selkiyttämään ohjelman tulostusta.

Esimerkki ohjelman suorituksesta:

```
Talla ohjelmalla voit syottaa ja tulostaa osoitteita.
Anna seuraavan henkilon tiedot.
Etunimi: Tiina
Sukunimi: Teekkari
Katuosoite: Jamerantaival 3 C 324
Postinumero: 02150
Postitoimipaikka: Espoo
Haluatko antaa lisaa osoitteita (k/e)?
k
Anna seuraavan henkilon tiedot.
Etunimi: Teemu
Sukunimi: Teekkari
Katuosoite: Servinkuja 4 B 44
Postinumero: 02150
Postitoimipaikka: espoo
Haluatko antaa lisaa osoitteita (k/e)?
e

Antamasi osoitteet:
```

```
Tiina Teekkari
Jamerantaival 3 C 324
02150 ESPOO

Teemu Teekkari
Servinkuja 4 B 44
02150 ESPOO
```

Käyttäjän syötettä lukiessa halutaan usein, että käyttäjä voi lopettaa syötteen antamisen tyhjällä rivillä sen jälkeen, kun hän on antanut haluamansa määrän rivejä. Kun syötettä luetaan `input`-funktion avulla, tunnistetaan tyhjä rivi siitä, että funktio palauttaa arvonaan tyhjän merkkijonon `""`.

Seuraavassa esimerkissä luetaan käyttäjän antamia lukuja niin kauan, että käyttäjä antaa tyhjän rivin. Sen jälkeen ohjelma laskee ja tulostaa annettujen lukujen keskiarvon. Käyttäjän antama rivi muutetaan luvuksi vasta sen jälkeen, kun on ensin tutkittu, onko luettu merkkijono tyhjä. Jos muunnos tehtäisiin ennen merkkijonon tutkimista, ohjelma kaatuisi siihen, että tyhjää merkkijonoa ei voi muuttaa luvuksi.

```
def main():
    print("Lasken keskiarvon antamistasi desimaaliluvuista.")
    print("Lopeta tyhjalla rivilla.")
    lukujen_maara = 0
    summa = 0.0
    loppu = False
    while not loppu:
        rivi = input()
        if rivi == "":
            loppu = True
        else:
            luku = float(rivi)
            summa = summa + luku
            lukujen_maara = lukujen_maara + 1
    if lukujen_maara > 0:
        keskiarvo = summa / lukujen_maara
        print("Niiden keskiarvo on", keskiarvo)
    else:
        print("Et antanut yhtaan lukua.")

main()
```

Esimerkki ohjelman suorituksesta:

```
Lasken keskiarvon antamistasi desimaaliluvuista.
Lopeta tyhjalla rivilla.
-5.0
15.0
8.0

Niiden keskiarvo on 6.0
```

Toinen esimerkki, jossa käyttäjä antaa heti aluksi tyhjän rivin:

```
Lasken keskiarvon antamistasi desimaaliluvuista.
Lopeta tyhjalla rivilla.

Et antanut yhtaan lukua.
```

5.3 Monikko

Monikko (engl. tuple) muistuttaa listaa, mutta erona on se, että monikon sisältöä ei voi muuttaa sen jälkeen, kun monikko on luotu. Sen vuoksi monikkoon kohdistuvat operaatiot ovat jonkin verran tehokkaampia kuin vastaavat listaan kohdistuvat operaatiot (koska monikon toteutuksessa ei tarvitse varautua sen mahdolliseen muuttamiseen). Monikkoa merkitään kaarisulkujen `()` avulla. Esimerkiksi alla luodaan uusi monikko, joka sisältää kolme kokonaislukua:

```
>>> lukumonikko = (22, 28, 35)
```

Monikoita voidaan monissa asioissa käsitellä samoin kuin listoja, esimerkiksi:

```
>>> print(lukumonikko[2])
35
```

Kuten edellä kerrottiin, monikon sisällön muuttaminen sen luomisen jälkeen ei kuitenkaan onnistu:

```
>>> lukumonikko[1] = 13
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Kappaleessa 4.3 kerrottiin, miten funktio voi palauttaa samalla kerralla useammankin kuin yhden arvon. Täsmällisesti ottaen tässä on kysymys siitä, että funktio palauttaa monikon, jonka alkioiden arvot voidaan sijoittaa yksittäisten muuttujien arvoksi. Vaikka palautettavia arvoja ei olekaan sijoitettu sulkujen sisään, Python-tulkki tulkitsee kuitenkin niiden muodostavan monikon.

Tällä kurssilla monikoita ei käsitellä tämän tarkemmin. Ne on vain esitelty tässä lyhyesti, jotta lukijalla olisi mahdollisuus ymmärtää muiden kirjoittamaa koodia, jossa monikoita on käytetty.

5.4 Sanakirja

Tarkastellaan esimerkkiä, jossa haluamme toteuttaa puhelinluettelon. Luettelo koostuu pareista, jotka sisältävät henkilön nimen ja hänen puhelinnumeronsa. Haluamme, että rakenteesta pystyy helposti etsimään halutun henkilön puhelinnumeron,

siihen pystyy lisäämään uusia puhelinnumeroita ja poistamaan vanhentuneita puhelinnumeroita. Yksinkertaisuuden vuoksi oletamme, että luettelossamme ei ole useita samannimisiä henkilöitä ja että jokaisella henkilöllä on vain yksi puhelinnumero.

Kuvattu rakenne olisi mahdollista toteuttaa listojen avulla. Kunkin henkilön nimi ja puhelinnumero muodostaisi yhden listan, ja itse puhelinluettelo olisi lista, jonka alkiot olisivat nimi–puhelinnumero-listoja. Tämän rakenteen ongelma on kuitenkin se, että halutun henkilön puhelinnumeron haku siitä olisi hidasta. Jos haluaisimme etsiä Matti Meikäläisen puhelinnumeron, ohjelman pitäisi käydä koko puhelinluettelolistaa läpi niin kauan, että se löytäisi nimen Matti Meikäläinen. Pahimmillaan koko puhelinluettelo pitäisi käydä läpi. Jos nimet olisi tallennettu puhelinluetteloon aakkosjärjestyksessä, hakua voitaisiin nopeuttaa selvästi käyttämällä binäärihakua, mutta tällöin taas uusien nimien lisääminen puhelinluetteloon aakkkosjärjestyksen mukaisille paikoilleen olisi hidasta (kaikkia lisättävän nimen jälkeen tulevia alkioita pitäisi siirtää listassa loppuun päin).

Python tarjoaa toisen rakenteen, jossa on mahdollista toteuttaa tehokkaasti alkion haku *avaimen* perusteella (puhelinluetteloesimerkissä henkilön nimi toimii avaimena) sekä uusien alkioiden haku ja vanhojen poisto. Tätä rakennetta kutsutaan *sanakirjaksi* (engl. dictionary). Sanakirjan tehokas toteuttaminen perustuu hajautusrakenteeseen, jota käsitellään tarkemmin Tietorakenteet ja algoritmit -kurssilla.

Tyhjän sanakirjan voi luoda aaltosulkujen `{}` avulla, esimerkiksi

```
>>> puh_luettelo = {}
```

Sanakirjaa luodessa voi myös antaa samalla jo sanakirjaan lisättäviä avain–arvo-pareja. (Puhelinluetteloesimerkissä avain on nimi ja arvo nimeen liittyvä puhelinnumero). Avain ja arvo erotetaan toisistaan kaksoispisteellä ja eri parit toisistaan pilkulla:

```
>>> puhelinluettelo = {"Teekkari Teemu" : "050-12345", "Fyysikko Tiina" : \
... "045-234567", "Kemisti Kalle" : "040-765432"}
```

luo uuden sanakirjan, joka sisältää kolme nimi–puhelinnumero-paria.

Johonkin avaimeen liittyvän arvon saa selville ilmauksella `sanakirja[avain]`, esimerkiksi

```
>>> print(puhelinluettelo["Fyysikko Tiina"])
045-234567
```

Sanakirjan on kuitenkin tällöin sisällettävä hakasulkujen sisällä annettu avain. Muuten aiheutuu virhetilanne, esimerkiksi:

```
>>> print(puhelinluettelo["Virtanen Maija"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Virtanen Maija'
```

Operaattorin `in` avulla voidaan kuitenkin tutkia ensin, onko avain sanakirjassa:

```
>>> nimi = "Virtanen Maija"
>>> if nimi in puhelinluettelo:
...     print(puhelinluettelo[nimi])
... else:
...     print("Nimea ei loydy luettelosta")
...
Nimea ei loydy luettelosta
>>> nimi = "Teekkari Teemu"
>>> if nimi in puhelinluettelo:
...     print(puhelinluettelo[nimi])
... else:
...     print("Nimea ei loydy luettelosta")
...
050-12345
```

Sanakirjaan voidaan lisätä sijoituskäskyn avulla uusia avain–arvo-pareja sekä muuttaa aikaisemmin lisättyihin avaimiin liittyviä arvoja.

```
>>> puhelinluettelo["Rakentaja Niina"] = "0400-123"
>>> puhelinluettelo["Kemisti Kalle"] = "041-56789"
>>> print(puhelinluettelo)
{'Kemisti Kalle': '041-56789', 'Fyysikko Tiina': '045-234567', 'Teekkari Teemu':
 '050-12345', 'Rakentaja Niina': '0400-123'}
```

Kaikki sanakirjaan kuuluvat avaimet voi käydä läpi for-käskyn avulla samaan tapaan kuin listan alkiot:

```
>>> for nimi in puhelinluettelo:
...     print(nimi)
...
Kemisti Kalle
Fyysikko Tiina
Teekkari Teemu
Rakentaja Niina
>>> for nimi in puhelinluettelo:
...     print("{:16s} {:12s}".format(nimi, puhelinluettelo[nimi]))
...
Kemisti Kalle    041-56789
Fyysikko Tiina   045-234567
Teekkari Teemu   050-12345
Rakentaja Niina  0400-123
```

Sanakirjasta voi poistaa jonkin avaimen ja siihen liittyvän arvon `del`-operaattorilla:

```
>>> del puhelinluettelo["Kemisti Kalle"]
>>> print(puhelinluettelo)
```

```
{'Fyysikko Tiina': '045-234567', 'Teekkari Teemu': '050-12345',
'Rakentaja Niina': '0400-123'}
```

5.5 Arvot ja viittaukset

Tässä kappaleessa puhutaan arvojen käsittelystä Pythonissa yleensä, ei siis pelkästään tässä luvussa esitetyistä tietorakenteista. Käytännössä nämä asiat tulevat kuitenkin esiin listojen, sanakirjojen sekä luvussa 7 esiteltävien olioiden yhteydessä.

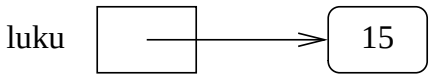
5.5.1 Muuttujat ja niiden arvot

Mitä täsmällisesti ottaen tarkoittaa se, että muuttujalle annetaan jokin arvo? Tietokoneen keskusmuistista on varattu jokaista ohjelman käyttämää muuttujaa varten oma tila tämän muuttujan arvon säilyttämistä varten. Pythonia käytettäessä tähän muistipaikkaan ei kuitenkaan sijoiteta muuttujan varsinaista arvoa (esim. kokonaislukua), vaan vain tieto siitä, mihin muuttujan varsinainen arvo on tietokoneen keskusmuistissa sijoitettu (käytännössä varsinaisen arvon sisältävän muistipaikan tunnus eli osoite).

Jos esimerkiksi Python-ohjelmassa on käsky

```
luku = 15
```

niin käskyä suoritettaessa jonnekin muualle tietokoneen keskusmuistissa tallennetaan arvo `15`, ja itse `luku`-muuttujalle varattuun tilaan sijoitetaan tieto siitä, mihin muistipaikkaan tuo 15 on tallennettu. Tätä tietoa sanotaan *viittaukseksi*.



On syytä huomata, että monissa muissa ohjelmointikielissä ei tehdä näin ainakaan silloin, kun käsitelläänmuuttujia, joiden arvot ovat yksittäisiä kokonais- tai desimaalilukuja. Esimerkiksi C- tai Java-kielellä kirjoitetussa ohjelmassa vastaava käsky aiheuttaisi sen, että muuttujalle `luku` varattuun paikkaan todella tallennettaisiin 15.

Koska Pythonissa muuttujalle varattuun tilaan tallennetaan vain viittaus itse arvoon, ohjelmassa ei tarvitse määritellä, minkä tyyppisiä arvoja kukin muuttuja voi saada. Ohjelman suorituksen aikana sama muuttuja voi jopa saada erityyppisiä arvoja.

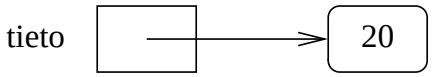
On täysin mahdollista kirjoittaa ohjelmaan ensin käsky

```
tieto = 20
```

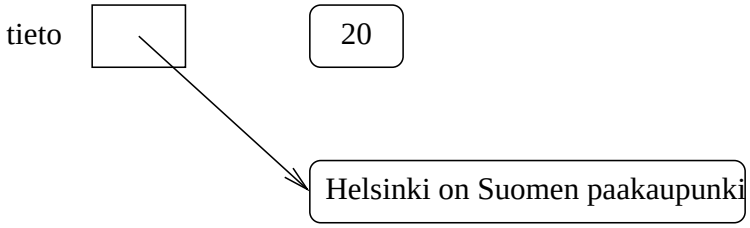
ja sen jälkeen samaan ohjelmaan käsky

```
tieto = "Helsinki on Suomen paakaupunki"
```

Ohjelmaa suoritettaessa ensin keskusmuistiin tallennetaan arvo 20 ja muuttujalle `tieto` varattuun tilaan viittaus tähän arvoon, eli tilanne on seuraavan kuvan mukainen



Kun jälkimmäinen käsky suoritetaan, keskusmuistiin tallennetaan merkkijono `Helsinki on Suomen paakaupunki`, ja muuttujalle `tieto` varatussa tilassa oleva viittaus vaihdetaan osoittamaan merkkijonon käyttämään muistipaikkaan, kuten alla olevassa kuvassa on esitetty.

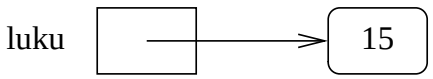


Jos itse arvo sijoittettaisiin muuttujalle varattuun tilaan, olisi annettujen käskyjen toteuttaminen selvästi hankalampaa, koska esimerkin merkkijono tarvitsee tietokoneen muistissa selvästi suuremman tilan kuin kokonaisluku 20.

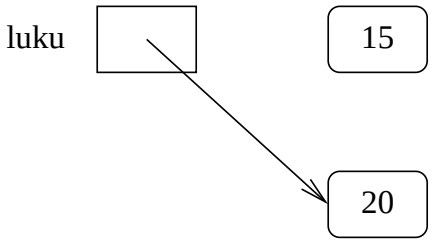
Yleensä erityyppisten arvojen sijoittaminen samalle muuttujalle ohjelman aikana ei ole kuitenkaan suositeltavaa, koska tällainen muuttujan käyttö tekee helposti ohjelmasta sekavan. Sen sijaan on hyvin yleistä se, että sama muuttuja saa ohjelman suorituksen kuluessa samantyyppisiä, mutta eri arvoja. Esimerkiksi ohjelmaa

```
luku = 15
luku = luku + 5
```

suoritettaessa `luku`-muuttuja viittaa ensimmäisen käskyn jälkeen arvoon 15,



ja toisen käskyn suorituksen jälkeen arvoon 20.



5.5.2 Muuttuvat tyypit

Edellä käsiteltiin muuttujia, joiden varsinaiset arvot ovat kokonaislukuja ja merkkijonoja. Näitä muuttujia käsitellessä muuttajan varsinaista arvoa ei koskaan muuteta sen jälkeen, kun arvo on annettu. Jos esimerkiksi muuttujan `luku` varsinainen arvo on 15 ja tämän jälkeen suoritetaan käsky

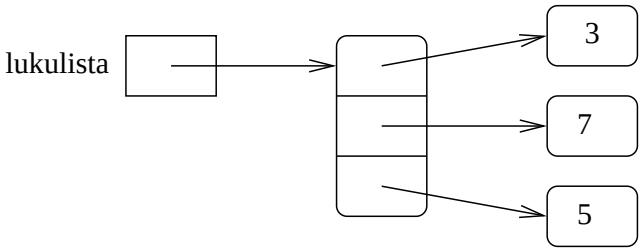
```
luku = luku + 5
```

niin tietokoneen keskusmuistissa luvun 15 päälle ei kirjoiteta arvoa 20, vaan arvo 20 kirjoitetaan toiseen paikkaan keskusmuistiin ja muuttuja `luku` vain vaihdetaan viittaamaan tähän uuteen paikkaan.

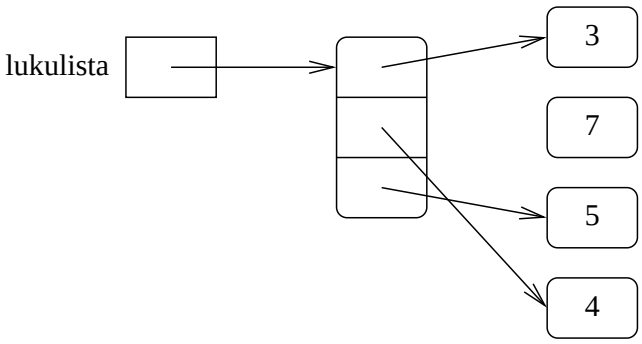
Osa Pythonin tyypeistä on kuitenkin *muuttuvia*. Tällaisilla tyypeillä varsinaista arvoa voi muuttaa. Esimerkiksi listat, sanakirjat ja luvussa 7 esiteltävät oliot ovat muuttuvia tyyppejä. Jos on luotu lista, joka sisältää luvut 3, 7 ja 5, niin luvun 7 paikalle voi vaihtaa sijoituskäskyllä luvun 4. Tarkasti ottaen lista sisältää ensin viittaukset lukuihin 3, 5 ja 7, ja sijoituskäskyn jälkeen listan toinen alkio siirretään viittaamaan lukuun 4. Olennaista tässä kuitenkin on se, että listan sisältöä voidaan muuttaa ilman, että pitää varata tilaa uudelle listalle ja vaihtaa listamuuttuja viittaamaan uuteen listaan. Jos siis suoritetaan käskyt

```
lukulista = [3, 7, 5]
lukulista[1] = 4
```

on tilanne keskusmuistissa ensimmäisen sijoituskäskyn jälkeen seuraavan kuvan näköinen

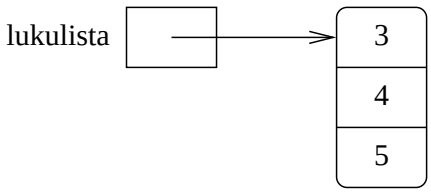


ja toisen sijoituskäskyn jälkeen alla olevan kuvan näköinen



Muuttuja `lukulista` siis viittaa koko ajan samaan listaan, mutta listan sisältö on vaihtunut.

Yksinkertaisuuden vuoksi listan sisältöä esimerkiksi jälkimmäisen sijoituskäskyn jälkeen kuvataan tässä luvussa jatkossa seuraavalla kuvalla, vaikka edellä ollut kuva kertoo tarkemmin todellisen tilanteen.



5.5.3 Parametrin arvon muuttaminen funktion sisällä

Muuttumattomat ja muuttuvat tyypit toimivat eri tavalla funktion parametreina. Jos funktio muuttaa muuttumatonta tyyppiä olevan parametrin arvoa, muutos ei näy mitenkään funktion ulkopuolella.

Tarkastellaan seuraavaa esimerkkiohjelmaa:

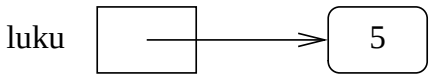
```
def muuta_luku(eka):
    print("Arvo funktiossa aluksi", eka)
    eka = 10
    print("Arvo funktiossa lopuksi", eka)


def main():
    luku = 5
    print("Arvo paaohjelman aluksi", luku)
    muuta_luku(luku)
    print("Arvo paaohjelman lopuksi", luku)


main()
```

Ohjelman tulostus on seuraava:

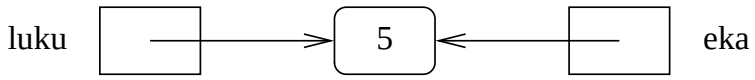
```
Arvo paaohjelman aluksi 5
Arvo funktiossa aluksi 5
Arvo funktiossa lopuksi 10
Arvo paaohjelman lopuksi 5
```

Parametrin arvo on siis muuttunut funktion sisällä, mutta pääohjelmassa muuttujalla `luku` on funktiosta palatessa vanha arvo. Seuraavassa katsotaan tarkemmin, mitä keskusmuistissa tapahtuu ohjelman suorituksen aikana.

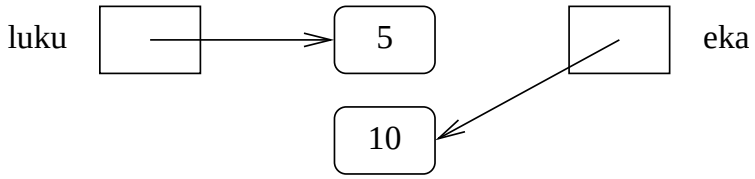
Pääohjelman alussa muuttuja `luku` pannaan viittaamaan arvon 5:



Kun funktion `muuta_luku` suoritus alkaa, parametri `eka` pannaan viittaamaan samaan arvoon.



Funktion `muuta_luku` sisällä olevassa sijoituskäskyssä parametri `eka` vaihdetaan viittaamaan uuteen arvoon. Itse arvoa 5 ei kuitenkaan muuteta, joten muuttuja `luku` viittaa edelleen samaan arvoon 10 kuin aikaisemminkin.



Kun ohjelman suoritus palaa takaisin pääohjelmaan, muuttuja `luku` viittaa edelleen arvoon 5, joten sen arvo ei ole mitenkään muuttunut funktion `muuta_luku` suorituksen aikana.

Sen sijaan funktio voi muuttaa muuttuvaa tyyppiä olevan parametrin varsinaista arvoa niin, että muutos näkyy myös funktion ulkopuolelle. Tarkastellaan esimerkkinä funktiota, joka muuttaa sille parametrina annetun listan sisältöä.

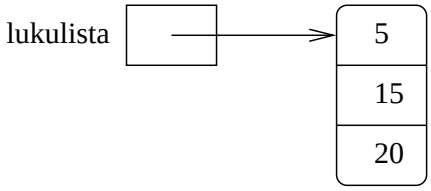
```
def muuta_alkio(lista):
    print("Lista funktiossa aluksi", lista)
    lista[1] = 12
    print("Lista funktiossa lopuksi", lista)


def main():
    lukulista = [5, 15, 20]
    print("Lista paaohjelman aluksi", lukulista)
    muuta_alkio(lukulista)
    print("Lista paaohjelman lopuksi", lukulista)


main()
```

Ohjelman tulostus näyttää seuraavalta:

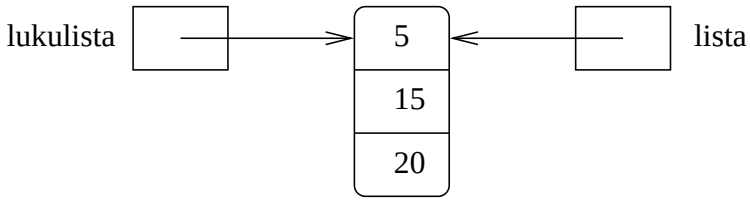
```
Lista paaohjelman aluksi [5, 15, 20]
Lista funktiossa aluksi [5, 15, 20]
Lista funktiossa lopuksi [5, 12, 20]
Lista paaohjelman lopuksi [5, 12, 20]
```

Listan sisältö siis näyttää muuttuneen myös silloin, kun lista tulostetetaan pääohjelmassa.

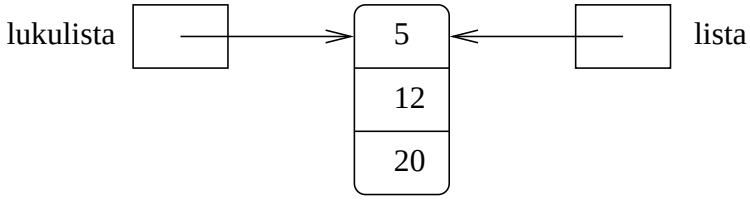
Tässä ohjelmassa luodaan pääohjelmassa ensin lista ja pannaan muuttuja `lukulista` viittaamaan siihen.



Kun funktion `muuta_alkio` suoritus alkaa, funktion parametri `lista` pannaan viittaamaan samaan listaan.



Funktion sisällä olevassa sijoituskäskyssä muutetaan yhtä listan sisällä olevaa alkiota, mutta parametri `lista` viittaa edelleen samaan listaan kuin funktion suorituksen alussa. Vain listan sisältö on muuttunut. Näin ollen myös pääohjelman muuttuja `lukulista` viittaa listaan, jonka yksi alkio on muuttunut. Siksi muutos näkyy myös pääohjelmassa.



Vastaava ilmiö esiintyy myös silloin, jos kaksi eri muuttujaa viittaa saman funktion sisällä samaan listaan. Tarkastellaan esimerkiksi seuraavaa ohjelmaa.

```
def main():
    lukulista = [5, 15, 20]
    print("Lukulista aluksi", lukulista)
    lista = lukulista
    lista[1] = 12
    print("Lukulista lopuksi", lukulista)


main()
```

Ohjelman tulostus näyttää seuraavalta.

```
Lukulista aluksi [5, 15, 20]
Lukulista lopuksi [5, 12, 20]
```

Sijoitus `lista[1] = 12` muutti siis myös muuttujan `lukulista` kautta tavoitettavaa listaa. Tämä johtuu siitä, että sijoituskäsky `lista = lukulista` ei tee kopiota muuttujan `lukulista` viittaamasta listasta ja pane sitä muuttujan `lista` arvoksi, vaan sijoituskäsky vain panee muuttujan `lista` viittaamaan samaan listaan kuin mihin muuttuja `lukulista` viittaa. Sijoituskäskyn jälkeenkin ohjelman käytössä on siis vain yksi lista, jota voi kuitenkin käsitellä kahden eri muuttujan kautta. Tilanteesta voisi piirtää samanlaisen kuvan kuin mitä esitettiin edellisen esimerkin yhteydessä.

Palataan vielä tilanteeseen, jossa pääohjelmassa käyttöön otettu listaan viittaava muuttuja annetaan parametrina funktiolle. Jos funktio muuttaa sitä, mihin listaan parametri viittaa eikä itse listan sisältöä, toimii ohjelma toisella tavalla, kuten neljännessä esimerkissä nähdään. Tarkastellaan ohjelmaa

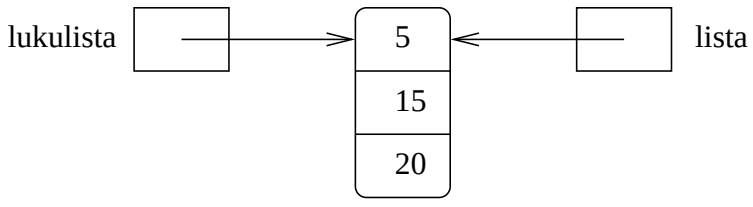
```
def muuta_lista(lista):
    print("Lista funktiossa aluksi", lista)
    lista = [1, 2, 5, 6]
    print("Lista funktiossa lopuksi", lista)

def main():
    lukulista = [5, 15, 20]
    print("Lista paaohjelman aluksi", lukulista)
    muuta_lista(lukulista)
    print("Lista paaohjelman lopuksi", lukulista)

main()
```

Ohjelman tulostus on seuraava:

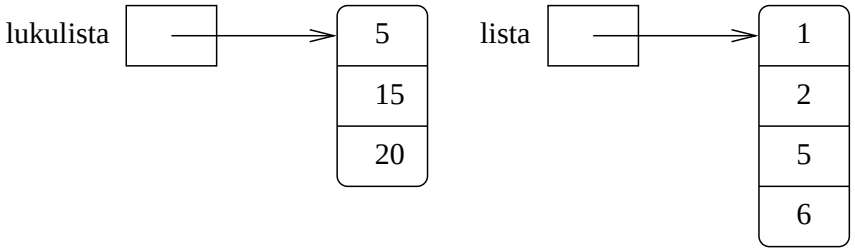
```
Lista paaohjelman aluksi [5, 15, 20]
Lista funktiossa aluksi [5, 15, 20]
Lista funktiossa lopuksi [1, 2, 5, 6]
Lista paaohjelman lopuksi [5, 15, 20]
```

Funktion sisällä tulostetaan muuttunut lista, mutta pääohjelman lopuksi tulostettava lista on sama kuin pääohjelman alussa.

Funktion `muuta_lista` suorituksen alussa parametri `lista` pannaan viittaamaan samaan listaan kuin mihin pääohjelman muuttuja `lukulista` viittaa.



Funktiossa luodaan kuitenkin kokonaan uusi lista, ja sijoituskäskyssä parametri `lista` vaihdetaan viittaamaan tähän uuteen listaan. Vanhan listan sisältö ei muutu mitenkään, ja pääohjelman muuttuja `lukulista` viittaa edelleen samaan listaan kuin se viittasi aikaisemminkin.



Kun funktion suoritus päättyy ja palataan pääohjelmaan, muuttuja `lukulista` viittaa edelleen samaan listaan kuin ennen funktion kutsua. Tätä listaa ei ole muutettu mitenkään ohjelman suorituksen aikana.

Ratkaiseva ero toisen ja neljännen esimerkin välillä on se, että toisessa esimerkissä muutettiin parametrin viittaaman listan sisältöä, jolloin muutos näkyi myös samaan listaan viittavan pääohjelman muuttujan kautta, kun taas kolmannessa esimerkissä muutettiin sitä, mihin listaan parametri viittaa.

5.6 Tiivistelmä luvussa esitettyjen tietorakenteiden käytöstä

Tähän kappaleeseen on vielä koottu lyhyesti esimerkkejä listojen, merkkijonojen ja sanakirjarakenteen käytöstä ilman selityksiä.

Tyhjän listan luominen

```
listamuuttuja = []
```

Uuden alkion lisääminen listan loppuun (listan koko kasvaa)

```
listamuuttuja.append(arvo)
```

Alkioiden lisääminen listaan heti listaa luodessa

```
listamuuttuja = [4, 8, 1, 15]
```

tai

```
listamuuttuja = [0] * 30
```

Listan alkion korvaaminen uudella

```
listamuuttuja[3] = 15
```

Listan läpikäynti

```
for muuttuja in listamuuttuja:
    kaskyjä, jotka kayttavat muuttujaa
```

(tässä tavassa käskyt eivät kuitenkaan voi muuttaa listan alkioiden arvoja)

tai

```
i = 0
while i < len(listamuuttuja):
    kaskyjä, joissa voi esiintyä listamuuttuja[i]
    i += 1
```

(tässä tavassa käskyt voivat myös muuttaa listan alkioiden arvoja).

Uuden alkion lisääminen listan keskelle (listan koko kasvaa)

```
listamuuttuja.insert(2,7)
```

lisää arvon 7 indeksille 2.

Listan koon / pituuden (alkioiden määrän) selvittäminen

```
koko = len(listamuuttuja)
```

Alkion poistaminen listasta

```
listamuuttuja.remove(5)
```

poistaa listasta ensimmäisen alkion, jonka arvo on `5`.

```
del listamuuttuja[5]
```

poistaa listasta indeksillä `5` olevan alkion.

Sen selvittäminen, esiintyykö alkio listassa, ja ensimmäisen esiintymän indeksi

```
if arvo in listamuuttuja:
    indeksi = listamuuttuja.index(arvo)
```

Listan järjestäminen

```
listamuuttuja.sort()
```

järjestää alkuperäisen listan.

```
listamuuttuja2 = sorted(listamuuttuja)
```

jättää alkuperäisen listan ennalleen, mutta tekee siitä kopion ja järjestää tämän kopion. Muuttuja `listamuuttuja2` viittaa nyt tähän kopioon.

Listan alkioiden järjestyksen muuttaminen päinvastaiseksi

```
listamuuttuja.reverse()
```

Listan kopiointi silloin, jos listan alkiot ovat lukuja, merkkijonoja tai totuusarvoja

```
kopiolista = [] + listamuuttuja
```

Kaksiulotteisen listan luominen ja viittaaminen sen alkioihin

```
matriisi = [[1, 5, 8], [3, 19, 25], [11, 21, 70], [31, 4, 41]]
print(matriisi[3][2])
matriisi[3][0] = 2
```

Merkkijonon luominen ja sen yksittäiseen alkioon tai osamerkkijonoon viittaaminen

```
mjono = "appelsiini"
print(mjono[3])
print(mjono[2:6])
```

Operaattoria `in` ja metodia `index` voi käyttää merkkijonoille samalla tavalla kuin listoille.

Merkkijonon isojen kirjainten muuttaminen pieniksi

```
mjono = mjono.lower()
```

Merkkijonon pienten kirjainten muuttaminen isoiksi

```
mjono = mjono.upper()
```

Tyhjien merkkien poistaminen merkkijonon alusta ja lopusta

```
mjono = mjono.strip()
```

vain alusta

```
mjono = mjono.lstrip()
```

vain lopusta

```
mjono = mjono.rstrip()
```

Merkkijonojen yhdistäminen

```
mjono = mjono1 + mjono2
```

ja monistaminen

```
pitkamjono = 5 * mjono
```

Merkkijonon jakaminen osiin aina merkin `","` kohdalta

```
osat = mjono.split(",")
```

Muita merkkejä voidaan käyttää vastaavalla tavalla. Jaossa käytetty merkki ei tule osiin mukaan.

Merkkijonojen sisältöjen vertailu

```
if mjono1 == mjono2:
    print("Merkkijonojen sisalto on tasmalleen sama.")
if mjono1 < mjono2:
    print("Ensimmaisen merkkijonon sisalto on jarjestyksessa ennen toista")
```

Sanakirjan luominen

```
hakemisto = {}
```

tai

```
hakemisto = {avain1 : arvo1, avain2 : arvo2, avain3 : arvo3}
```

Avaimeen liittyvän arvon hakeminen sanakirjasta

```
if avain in hakemisto:
    loydetty_arvo = hakemisto[avain]
else:
    print("Haettua avainta ei ole sanakirjassa.")
```

Uuden avain–arvo-parin lisääminen tai avaimeen liittyvän arvon muuttaminen

```
hakemisto[avain] = uusi_arvo
```

Sanakirjan avainten (ja halutessa niihin liittyvien arvojen) läpikäynti

```
for avain in hakemisto:
    tee jotain avaimelle tai hakemisto[avain]:lle
```

Avaimen ja siihen liityvän arvon poistaminen sanakirjasta

```
del hakemisto[avain]
```

Luku 6

Poikkeukset ja tiedostojen käsittely

6.1 Poikkeukset

Ei ole harvinaista, että ohjelmaa suoritettaessa törmätään virhetilanteisiin. Osa virheistä johtuu ohjelmointivirheistä, mutta osa on sellaisia, joihin ohjelmoija ei voi mitenkään vaikuttaa. Esimerkiksi ohjelma pyytää käyttäjältä kokonaisluvun, mutta käyttäjä antaakin kirjaimia. Tai ohjelman pitäisi tallentaa tietoja tiedostoon, mutta kovalevytila on jo täynnä.

Suurta osaa näistä virhetilanteista pystyisi periaateessa käsittelemään if-else-rakenteiden avulla. Niitä käytettäessä ohjelman rakenteesta tulee kuitenkin helposti monimutkainen. Lisäksi ohjelman päätehtävään liittyvät käskyt saattavat hukkua erilaisia virhetilanteita käsittelevien if-else-rakenteiden sisään.

Python tarjoaa virhetilanteiden käsittelyyn oman mekanismin, *poikkeukset*. Jos ohjelman suorituksessa sattuu virhetilanne eli syntyy ns. poikkeus, johon on ennalta varauduttu, se voidaan käsitellä `try`–`except`-rakenteen avulla.

Rakenteen yleinen muoto on seuraava:

```
try:
    # Jono käskyjä, joista jokin tai jotkin voivat aiheuttaa tyypin
    # poikkeuksen_tyyppi poikkeuksen.
except poikkeuksen_tyyppi:
    # Käskyjä, jotka jotenkin selvittävät virhetilanteen, jos on
    # aiheutunut poikkeuksen_tyyppi-tyyppinen poikkeus.
```

Tällainen koodi suoritetaan siten, että try-osassa olevia käskyjä suoritetaan yksi kerrallaan normaaliin tapaan. Jos jonkin käskyn suoritus aiheuttaa poikkeuksen, jonka tyyppi on `poikkeuksen_tyyppi`, hypätään välittömästi suorittamaan except-osassa olevia käskyjä. Kun except-osan käskyt on suoritettu, ei enää palata try-osaan, vaan jatketaan ohjelman suoritusta except-osan jälkeen tulevasta ohjelman osasta.

Jos try-osan suorituksessa ei aiheudu poikkeuksia, except-osan käskyjä ei suoriteta lainkaan, vaan try-osan suorituksen jälkeen siirrytään ohjelmassa except-osan jälkeen tuleviin käskyihin.

Rakenteessa voi olla useita except-osia erityyppisiä poikkeuksia varten. Suoritettava except-osa valitaan aina aiheutuneen poikkeuksen tyypin mukaan.

Except-osaan kirjoitettava koodi vaihtelee tilanteen mukaan. Joskus (jos esimerkiksi ohjelman pitäisi kirjoittaa tiedostoon, mutta kovalevytila on täynnä), ei ole muuta vaihtoehtoa kuin antaa käyttäjälle selväsanainen virheilmoitus siitä, mitä on tapahtunut. Tämäkin on kuitenkin parempi vaihtoehto kuin se, että ohjelma vain kaatuu, eikä käyttäjä saa selville, mikä on mennyt pieleen. Joskus taas käskyt voidaan suunnitella siten, että ne todella korjaavat virhetilanteen, esimerkiksi pyytävät käyttäjältä uutta syötettä käyttäjän antaman kelvottoman syötteen tilalle.

Ensimmäisessä esimerkissä käyttäjälle annetaan vain virheilmoitus väärästä syötteestä. Ohjelman on tarkoitus muuttaa käyttäjän nauloina antama massa kilogrammoiksi. Massa nauloina pitää antaa kokonaislukuna. Normaaliin tapaan käyttäjän antama syöte yritetään muuttaa kokonaisluvuksi `int()`-tyypinmuunnoksella. Jos se ei onnistu, siirrytään except-osaan, jossa tulostetaan käyttäjälle virheilmoitus. Virhe, joka aiheutuu siitä, että vääränlaista syötettä yritetään muuttaa luvuksi on tyypiltään `ValueError`. Sen vuoksi except-osa kirjoitetaan käsittelemään nimenomaan tämäntyyppinen virhe.

```
def main():
    NAULAKERROIN = 0.4536
    print("Muutan nauloina annetun massa kilogrammoiksi.")
    try:
        syote = input("Anna massa nauloina: ")
        naulat = int(syote)
        kilot = NAULAKERROIN * naulat
        print("Massa on {:.3f} kg".format(kilot))
    except ValueError:
        print("Virhe - et antanut nauloja kokonaislukuna.")


main()
```

Esimerkki ohjelman suorituksesta silloin, kun käyttäjä antaa sopivan syötteen:

```
Muutan nauloina annetun massa kilogrammoiksi.
Anna massa nauloina: 45
Massa on 20.412 kg
```

Kaksi esimerkkiä suorituksesta silloin, kun käyttäjän syöte ei ole sopiva:

```
Muutan nauloina annetun massa kilogrammoiksi.
Anna massa nauloina: 45.8
Virhe - et antanut nauloja kokonaislukuna.
```

```
Muutan nauloina annetun massa kilogrammoiksi.
Anna massa nauloina: jotain tekstia
Virhe - et antanut nauloja kokonaislukuna.
```

Toisessa esimerkissä ei tyydytä pelkkään virheilmoitukseen. Siinä käyttäjältä pyydetään massaa uudelleen niin kauan, että tämä saadaan kokonaislukuna. Tämä saadaan aikaiseksi sijoittamalla try–except-rakenne toistokäskyn sisään. Toistokäskyä suoritetaan niin kauan, kunnes on saatu luetuksi kelvollinen kokonaisluku. Tätä tutkitaan muuttujan `luku_onnistui` avulla. Sen arvo on aluksi `False`, mutta se muutetaan `True`:ksi try-osan lopussa (joka suoritetaan vain siinä tapauksessa, että try-osan aikaisemmat käskyt on suoritettu ilman poikkeuksen aiheutumista).

```
def main():
    NAULAKERROIN = 0.4536
    print("Muutan nauloina annetun massa kilogrammoiksi.")
    luku_onnistui = False
    while not luku_onnistui:
        try:
            syote = input("Anna massa nauloina: ")
            naulat = int(syote)
            kilot = NAULAKERROIN * naulat
            print("Massa on {:.3f} kg".format(kilot))
            luku_onnistui = True
        except ValueError:
            print("Virhe - et antanut nauloja kokonaislukuna.")
            print("Yrita uudelleen!")


main()
```

Esimerkki ohjelman suorituksesta silloin, kun käyttäjä antaa heti sopivan syötteen:

```
Muutan nauloina annetun massa kilogrammoiksi.
Anna massa nauloina: 150
Massa on 68.040 kg
```

Toinen ajoesimerkki, jossa käyttäjä antaa ensin kaksi kertaa väärän syötteen:

```
Muutan nauloina annetun massa kilogrammoiksi.
Anna massa nauloina: 120.5
Virhe - et antanut nauloja kokonaislukuna.
Yrita uudelleen!
Anna massa nauloina: sotkua
Virhe - et antanut nauloja kokonaislukuna.
Yrita uudelleen!
Anna massa nauloina: 120
Massa on 54.432 kg
```

6.2 Tekstitiedostojen käsittely

Tähän asti esitetyt ohjelmat ovat aina lukeneet syötteensä suoraan siitä, mitä käyttäjä on kirjoittanut näppäimistöllä. (Täsmällisesti sanottuna ohjelmat ovat lukeneet syötteensä standardisyöttövirrasta, joka on yleensä yhdistetty näppäimistöön, mutta jonka voi käyttöjärjestelmän puolella määritellä yhdistettäväksi johonkin muuhunkin.) Usein on kuitenkin tarpeen tallentaa ohjelman käsittelemää tietoa ohjelman eri suorituskertojen välillä. Ajatellaan aikaisemmin esitettyä puhelinluettelo-esimerkkiä. Tietokoneella olevasta puhelinluettelo-ohjelmasta ei ole paljonkaan iloa, jos käyttäjän pitää aina ohjelmaa käynnistäessään ensin syöttää sille ystäviensä nimet ja puhelinnumerot ennen kuin puhelinnumeroita voi ruveta kyselemään. Paljon kätevämpää on tällöin katsoa puhelinnumero suoraan jostain muualta.

Jotta puhelinluettelo-ohjelma toimisi järkevästi, täytyy olla jokin tapa, jolla säilytetään käyttäjän aikaisemmalla kerralla syöttämiä puhelinnumeroita niin, että ne ovat ohjelman käytössä sen myöhemmillä suorituskerroilla. Tähän voidaan käyttää tiedostoa. Ohjelma tallentaa sillä tiedossa olevat puhelinnumerot tiedostoon. Kun ohjelma seuraavan kerran käynnistetään, tiedostossa olevat puhelinnumerot luetaan ensin keskusmuistiin ja vasta tämän jälkeen aloitetaan ohjelman muiden toimintojen suoritus.

Tiedostoja on kahdenlaisia, tekstitiedostoja ja binääritiedostoja. Tekstitiedostoon tiedot on tallennettu merkkeinä, binääritiedostoihin taas arvoja kuvaavina tavuina. Esimerkiksi luku 190 tallennetaan tekstitiedostoon niin, että tallennetaan peräkkäin merkit '1', '9' ja '0'. Binääritiedostoon taas sama luku tallennetaan niin, että tallennetaan kokonaislukuarvon 190 binääriesitystä vastaavat tavut. Tällä kurssilla tarkastellaan vain tekstitiedostojen käsittelyä. Tekstitiedostojen etuna on se, että niitä voidaan kirjoittaa muutenkin kuin Python-ohjelmilla, esimerkiksi Emacsilla, Notepadilla tai millä tahansa ohjelmalla, joka kirjoittaa puhdasta tekstiä.

Seuraavissa kappaleissa tarkastellaan ensin tiedon lukemista tekstitiedostosta ja sitten sitä, miten ohjelma voi kirjoittaa tietoa tekstitiedostoon.

6.2.1 Lukeminen tekstitiedostosta

Ennen kuin ohjelma aloittaa lukemisen tekstitiedostosta, on ohjelmalle kerrottava, mitä käyttöjärjestelmän tiedostoa ohjelmassa tiedostosta käytettävä muuttuja vastaa. Tämä tehdään `open`-funktion avulla. Funktiota käytetään seuraavasti:

```
tiedostomuuttuja = open(tiedoston_nimi, kasittelytapa)
```

Tässä `tiedostomuuttuja` on ohjelmassa sen muuttujan nimi, jonka avulla tiedostoa käsitellään, `tiedoston_nimi` tiedostosta käyttöjärjestelmän puolella käytettävä nimi ja `kasittelytapa` kertoo, aikooko ohjelma lukea tietoa tästä tiedostosta vai kirjoittaa siihen. Funktion `open` kutsumista sanotaan tiedoston *avaamiseksi*. Jos tiedosto avataan lukemista varten, käsittelytavaksi annetaan `"r"` (englannin sanasta read). Avattavan tiedoston on oltava samassa hakemistossa kuin missä ohjelmaa ajetaan, tai muussa tapauksessa tiedoston nimen pitää sisältää myös hakemistopolku.

Esimerkiksi seuraava käsky avaa tiedoston, jonka nimi on `tekstia.txt` lukemista varten niin, että ohjelmassa tiedostoa voidaan käsitellä muuttujan `lahtotiedosto` avulla:

```
lahtotiedosto = open("tekstia.txt", "r")
```

Kun tiedosto on avattu, siitä voidaan lukea rivejä eri tavoin. Yksi tapa on lukea tiedostosta rivi kerrallaan metodin `readline` avulla seuraavasti:

```
eka_rivi = lahtotiedosto.readline()
```

Tässä luettu rivi siis tallennettiin muuttujaan `eka_rivi`. Metodi `readline` toimii siten, että kun sitä kutsutaan ensimmäisen kerran, se palauttaa tiedoston ensimmäisen rivin, toinen kutsu palauttaa tiedoston toisen rivin jne. Python-tulkki siis pitää kirjaa siitä, missä kohdassa tiedostossa ollaan menossa, jolloin `readline`-metodi osaa aina lukea seuraavan vielä lukemattoman rivin. Jos metodia kutsutaan siinä vaiheessa, kun tiedosto on jo luettu loppuun, metodi palauttaa tyhjän merkkijonon `""`. Siitä tiedetään, että tiedosto on jo luettu loppuun. (Jos tiedostossa on välissä rivejä, jolla ei ole tekstiä, metodi ei kuitenkaan niitä lukiessa palauta tyhjää merkkijonoa, vaan merkkijonon, joka sisältää ainoastaan rivinvaihtomerkin `"\n"`.)

Kun tiedoston lukeminen lopetetaan, on tiedosto syytä sulkea `close`-metodilla. Tämä vapauttaa tiedoston käsittelyyn käyttöjärjestelmän puolella varatut resurssit.

Oletetaan, että tiedosto `tekstia.txt` on seuraavan näköinen:

```
ensimmainen rivi
toinen rivi
viimeinen rivi
```

Voimme nyt kirjoittaa ohjelman, joka lukee tiedoston rivit ja tulostaa jokaisen rivin kuvaruudulle seuraavasti:

```
def main():
    try:
        lahtotiedosto = open("tekstia.txt", "r")
        rivi = lahtotiedosto.readline()
        while rivi != "":
            print(rivi)
            rivi = lahtotiedosto.readline()
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston lukemisessa. Ohjelma paattyy.")


main()
```

Ohjelmessa on käytetty try–except-rakennetta käsittelemään mahdolliset `OSError`-tyyppiset poikkeukset. Tällainen poikkeus voi aiheutua esimerkiksi silloin, jos tiedostoa `tekstia.txt` ei ole lainkaan tai sitä ei pystytä lukemaan esimerkiksi lukuoikeuksien puuttumisen tai laitteistovian takia. (Katso lisätietoja käytetystä poikkeuksesta kappaleesta 6.2.4.)

Jos ohjelma suoritetaan onnistuneesti, näyttää sen tulostus seuraavalta:

```
ensimmainen rivi

toinen rivi

viimeinen rivi
```

Ohjelma on siis tulostanut ylimääräisen rivinvaihdon jokaisen rivin jälkeen. Tämä johtuu siitä, että metodi `readline` palauttaa myös rivin lopussa olevan rivinvaihtomerkin lukemansa merkkijonon lopussa. Kun `print`-käsky puolestaan lisää rivinvaihdon tulostuksensa loppuun, tulee joka rivin jälkeen kaksi rivinvaihtoa: toinen, joka on tiedostosta luetun rivin lopussa, ja toinen, jonka `print`-käsky on lisännyt.

Ongelma voidaan ratkaista poistamalla tiedostosta luetun rivin lopussa oleva rivinvaihtomerkki. Tämä voidaan tehdä esimerkiksi `rstrip`-metodilla. (Metodi poistaa myös muut luetun rivin lopussa olevat ns. tyhjät merkit, esimerkiksi välilyönnit, tabuloinnit jne. Jos tätä ei toivota, on rivinvaihtomerkki poistettava jollain muulla tavoin – esimerkiksi ottamalla luetusta rivistä alimerkkijono, joka ei sisällä rivin viimeistä merkkiä. Tämäkin tapa voi kuitenkin aiheuttaa vaikeuksia käyttöjärjestelmässä, jossa rivinvaihtoa merkitään useammalla kuin yhdellä merkillä.) Seuraavaa ohjelmaa on muutettu myös niin, että luettavan tiedoston nimeä ei ole määrätty ohjelmassa, vaan se kysytään käyttäjältä.

```
def main():
    nimi = input("Anna luettavan tiedoston nimi: ")
    try:
        lahtotiedosto = open(nimi, "r")
        rivi = lahtotiedosto.readline()
        while rivi != "":
            rivi = rivi.rstrip()
            print(rivi)
            rivi = lahtotiedosto.readline()
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston", nimi, "lukemisessa. Ohjelma paattyy.")


main()
```

Ohjelman suoritus näyttää nyt seuraavalta. Ylimääräiset rivinvaihdot ovat hävinneet.

```
Anna luettavan tiedoston nimi: tekstia.txt
ensimmainen rivi
toinen rivi
viimeinen rivi
```

Toisessa esimerkissä yritetään lukea tiedostoa, jota ei ole olemassa.

```
Anna luettavan tiedoston nimi: olematon.txt
Virhe tiedoston olematon.txt lukemisessa. Ohjelma paattyy.
```

Edellä tiedoston rivejä on luettu while-käskyn avulla rivi kerrallaan. Jos rivit käydään läpi for-käskyn avulla, voidaan lukeminen kirjoittaa selvästi yksinkertaisemmin. Jos toistokäsky kirjoitetaan muotoon

```
for rivi in lahtotiedosto:
    tee jotain riville rivi
```

huolehtii for-käsky siitä, että tiedoston jokainen rivi luetaan vuorotellen ja sijoitetaan muuttujan `rivi` arvoksi. Ohjelmaan ei silloin tarvitse kirjoittaa lainkaan `readline`-käskyjä, vaan Python-tulkki huolehtiin rivien lukemisesta for-käskyä suorittaessaan.

Tiedoston lukeva ja sen rivit tulostava ohjelma voidaan siis kirjoittaa seuraavasti:

```
def main():
    nimi = input("Anna luettavan tiedoston nimi: ")
    try:
        lahtotiedosto = open(nimi, "r")
        for rivi in lahtotiedosto:
            rivi = rivi.rstrip()
            print(rivi)
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston", nimi, "lukemisessa. Ohjelma paattyy.")


main()
```

Jos tiedosto luetaan for-käskyn avulla, ei samassa ohjelmassa kannata yleensä käyttää lainkaan `readline`-käskyä, koska sitä käytettäessä for-käsky ei käy lainkaan läpi niitä rivejä, jotka luetaan `readline`-käskyllä.

Oletetaan, että käyttäjä on kirjoittanut suuren juhlan vieraslistan tiedostoon. Kunkin vieraan nimi on kirjoitettu omalle riville. Käyttäjä haluaa sitten tarkistaa, onko jokin nimi vieraslistassa. Seuraava ohjelma pyytää käyttäjältä tiedoston ja yhden nimen sekä tarkistaa, onko käyttäjän antama nimi tiedostossa. Vaikka tiedostosta luettuja rivejä ei nyt tulostetakaan, on `rstrip`-metodin käyttäminen tässäkin tapauksessa tarpeellista, sillä muuten tiedostosta luettujen nimien lopussa olisi rivinvaihtomerkki eikä mikään niistä silloin olisi merkkijonojen vertailussa yhtäsuuri käyttäjän antaman nimen kanssa.

```
def main():
    nimi = input("Anna tiedoston nimi: ")
    etsittava_nimi = input("Anna tiedostosta etsittava nimi: ")
    loytyi = False
    try:
        lahtotiedosto = open(nimi, "r")
        for rivi in lahtotiedosto:
            rivi = rivi.rstrip()
            if rivi == etsittava_nimi:
                loytyi = True
        lahtotiedosto.close()
        if loytyi:
            print("Nimi", etsittava_nimi, "loytyi tiedostosta.")
        else:
            print("Nimea", etsittava_nimi, "ei loytynyt.")
    except OSError:
        print("Virhe tiedoston", nimi, "lukemisessa. Ohjelma paattyy.")


main()
```

Kolmas tapa lukea tiedostosta on käyttää metodia `readlines()`. Tämä metodi lukee kaikki tiedoston jäljellä olevat rivit listaan (kukin rivi on yksi listan alkio). Sen jälkeen listassa olevia rivejä voidaan käsitellä halutulla tavalla. Esimerkiksi seuraava ohjelma lukee käyttäjän antamat rivit listaan ja tulostaa sitten listan kaikki alkiot niin, että se poistaa tyhjät merkit kunkin rivin lopusta:

```
def main():
    nimi = input("Anna luettavan tiedoston nimi: ")
    try:
        lahtotiedosto = open(nimi, "r")
        rivilista = lahtotiedosto.readlines()
        lahtotiedosto.close()
        for rivi in rivilista:
            rivi = rivi.rstrip()
            print(rivi)
    except OSError:
        print("Virhe tiedoston", nimi, "lukemisessa. Ohjelma paattyy.")


main()
```

Jos tiedostosta luetaan lukuja, on tiedostosta luettu rivi muutettava kokonais- tai desimaaliluvuksi ihan samalla tavalla kuin käyttäjän syötettä näppäimistöltäkin luettaessa. Tämä johtuu siitä, että rivit luetaan tiedostosta aina merkkijonoina niiden sisällöstä riippumatta. Oletetaan, että tiedostossa on annettu eri päivien lämpötiloja, kukin omalla rivillään. Seuraava ohjelma lukee lämpötilat ja laskee niiden keskiarvon.

```
def main():
    nimi = input("Mista tiedostosta lampotilat luetaan: ")
    summa = 0.0
    lkm = 0
    try:
        lampotiedosto = open(nimi, "r")
        for rivi in lampotiedosto:
            rivi = rivi.rstrip()
            lampotila = float(rivi)
            summa += lampotila
            lkm += 1
        lampotiedosto.close()
        if lkm == 0:
            print("Tiedostossa ei ollut yhtaan lampotilaa.")
        else:
            keskiarvo = summa / lkm
            print("Lampotilojen keskiarvo on", keskiarvo)
    except OSError:
        print("Virhe tiedoston", nimi, "lukemisessa. Ohjelma paattyy.")
    except ValueError:
        print("Virheellinen rivi tiedostossa", nimi, "- ohjelma paattyy.")


main()
```

Ohjelmassa on käytetty `ValueError`-tyyppistä poikkeusta selvittämään niitä virhetilanteita, joissa tiedostosta luettua riviä ei pystytä muuttamaan luvuksi.

Jos tiedoston samalla rivillä on useita eri tietoja, esimerkiksi päivämäärä ja lämpötila, pitää luettu rivi jakaa ensin osiin esimerkiksi `split`-metodilla. Tämän jälkeen käsittelyä voidaan jatkaa halutuista osista. Seuraavassa esimerkkiohjelmassa lämpötilat luetaan tiedostosta, jossa jokaisella rivillä on ensin päivämäärä (tekstinä) ja sen jälkeen lämpötila. Oletetaan, että päivämäärä ja lämpötila on erotettu toisistaan pilkulla eikä rivillä ole muita pilkkuja.

Kukin luettu rivi jaetaan ensin `split`-metodilla kahteen osaan, joista ensimmäisessä pitäisi olla päivämäärä (josta ohjelma ei ole kiinnostunut) ja toisessa lämpötila. Ohjelma tarkistaa ensin, että osia on todellakin kaksi. Jos ei ole, ohjelma ilmoittaa virheellisestä rivistä ja tulostaa sen, mutta jatkaa kuitenkin toimintaansa lukemalla seuraavan rivin. Jos osia on kaksi, jälkimmäinen niistä muutetaan desimaaliluvuksi ja lasketaan mukaan lämpötilojen summaan samaan tapaan kuin edellisessä esimerkissä. Jos tiedoston lukeminen ei onnistu tai jollain rivillä on kaksi osaa, mutta jälkimmäistä niistä ei voi muuttaa desimaaliluvuksi, ohjelma ilmoittaa virheestä ja lopettaa toimintansa.

Taulukkolaskentaohjelmien käsittelemiä tiedostoja voidaan tallentaa csv-tiedostoiksi, joissa tiedoston tiedot on tallennettu tekstitiedostoon riveittäin siten, että eri sarakkeet on erotettu toisistaan pilkulla. Tällaisia tiedostoja pystyy lukemaan ja käsittelemään Python-ohjelmassa juuri seuraavan esimerkin tavoin. (Monet taulukkolaskentaohjelmat antavat tiedostoa tallennettaessa käyttäjän valita sarakkeita erottavaksi merkiksi jonkin muunkin merkin pilkun sijaan.)

```
def main():
    nimi = input("Mista tiedostosta lampotilat luetaan: ")
    summa = 0.0
    lkm = 0
    try:
        lampotiedosto = open(nimi, "r")
        for rivi in lampotiedosto:
            rivi = rivi.rstrip()
            osat = rivi.split(",")
            if len(osat) != 2:
                print("Virheellinen rivi:", rivi)
            else:
                lampotila = float(osat[1])
                summa += lampotila
                lkm += 1
        lampotiedosto.close()
        if lkm == 0:
            print("Tiedostossa ei ollut yhtaan lampotilaa.")
        else:
            keskiarvo = summa / lkm
            print("Lampotilojen keskiarvo on", keskiarvo)
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")
    except ValueError:
        print("Virheellinen lampotila tiedostossa", nimi,
              "- Ohjelma paattyy.")


main()
```

Jos halutaan, että ohjelma ei lopeta toimintaansa virheellisen lämpötilan lukemisen jälkeen, vaan jatkaa lukemalla seuraavan rivin, kirjoitetaan ohjelmaan `ValueError`-poikkeuksen käsittelevä try-except-rakenne `OSError`-poikkeuksen käsittelevän rakenteen ja tiedoston rivejä lukevan toistokäskyn sisään. Ohjelma näyttää silloin seuraavalta:

```
def main():
    nimi = input("Mista tiedostosta lampotilat luetaan: ")
    summa = 0.0
    lkm = 0
    try:
        lampotiedosto = open(nimi, "r")
        for rivi in lampotiedosto:
            rivi = rivi.rstrip()
            osat = rivi.split(",")
            if len(osat) != 2:
                print("Virheellinen rivi:", rivi)
            else:
                try:
                    lampotila = float(osat[1])
                    summa += lampotila
                    lkm += 1
                except ValueError:
                    print("Virheellinen lampotila:", osat[1])
        lampotiedosto.close()
        if lkm == 0:
            print("Tiedostossa ei ollut yhtaan lampotilaa.")
        else:
            keskiarvo = summa / lkm
            print("Lampotilojen keskiarvo on", keskiarvo)
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")


main()
```

6.2.2 Kirjoittaminen tiedostoon

Jos ohjelman halutaan kirjoittavan tiedostoon, avataan tiedosto muuten samalla tavalla kuin tiedostosta lukiessa, mutta käsittelytapana on nyt joko `"w"` tai `"a"`. Näiden kahden käsittelytavan ero on siinä, että `"w"` (englannin sanasta write) kirjoittaa mahdollisen vanhan samannimisen tiedoston päälle (vanhan tiedoston sisältö häviää siis kokonaan, vaikka vanha tiedosto olisi pitempi kuin kirjoitettava uusi teksti), kun taas `"a"` (englannin sanasta append) säilyttää avattavassa tiedostossa mahdollisesti jo olevan tekstin ja kirjoittaa tiedoston loppuun. Jos annetun nimistä tiedostoa ei vielä ole, se luodaan avaamisen yhteydessä.

Esimerkiksi

```
tulostiedosto = open("tekstia.txt", "w")
```

avaa kirjoittamista varten tiedoston, jonka nimi on `tekstia.txt`. Tiedoston mahdollinen vanha sisältö häviää avaamisen yhteydessä. Ohjelmassa voidaan tämän jälkeen käsitellä avattua tiedostoa muuttujan `tulostiedosto` avulla.

Avattuun tiedostoon voidaan tulostaa merkkijonoja metodin `write` avulla. Metodi ei lisää rivinvaihtomerkkiä tulostettavan merkkijonon loppuun, vaan lisäys pitää tehdä itse, esimerkiksi:

```
tulostiedosto.write("Ensimmainen rivi\n")
```

Metodilla `write` voi tulostaa vain merkkijonoja. Jos tiedostoon halutaan tallentaa esimerkiksi lukuja, pitää ne ensin muuttaa merkkijonoiksi. Tämä voidaan tehdä esimerkiksi käyttämällä tulostuksen muotoilua:

```
kanta = 3.5
ekspo = 5
tulos = kanta ** ekspo
tulostiedosto.write("{:.2f} potenssiin {:d} on {:.2f}\n".format(
                     kanta, ekspo, tulos))
```

Toinen vaihtoehto on muuttaa tulostettavat luvut ensin merkkijonoiksi `str`-operaattorilla. Jos samalle riville tulostettava teksti koostuu useammasta osasta, voidaan osat yhdistää pidemmäksi merkkijonoksi `+`-operaattorilla:

```
kanta = 3.5
ekspo = 7
tulos = kanta ** ekspo
tulostiedosto.write(str(kanta) + " potenssiin " + str(ekspo) + " on " + \
                    str(tulos) + "\n")
```

Tällöin desimaaliluvut tulostuvat Pythonin käyttämällä oletustarkkuudella, kun taas tulostuksen muotoilua käytettäessä ohjelmoija määrää käytettävän tarkkuuden.

Kun halutut tiedot on tulostettu tiedostoon, on tiedosto suljettava `close`-metodilla. Tiedostoa ei ole syytä yrittää lukea ennen kuin tiedosto on suljettu. Tehokkuussyistä `write`-käskyjä suoritettaessa tulostamista ei tehdä itse tiedostoon välttämättä heti jokaisen `write`-käskyn jälkeen, vaan kirjoitettavaa dataa kerätään ensin keskusmuistissa olevaan puskuriin, jonka sisältö siirretään itse tiedostoon suurempina osina. Ennen `close`-metodin suorittamista tulostettu tieto ei vielä välttämättä ole itse tiedostossa, vaan vasta puskurissa, eikä se näy, jos tiedostoa yrittää lukea jollain tavalla.

Seuraava esimerkkiohjelma tulostaa edellisen kappaleen esimerkissä käytetyn vieraslistan tiedostoon. Ohjelma pyytää ensin käyttäjältä tiedoston nimen ja sen jälkeen tiedostoon kirjoitettavat nimet. Nimet tulostetaan tiedostoon rivi kerrallaan.

```
def main():
    print("Ohjelma kirjoittaa vieraslistan haluamaasi tiedostoon.")
    nimi = input("Anna kirjoitettavan tiedoston nimi: ")
    try:
        tulostiedosto = open(nimi, "w")
        print("Anna tallennettavat nimet.")
        print("Lopeta tyhjalla rivilla.")
        rivi = input()
        while rivi != "":
            tulostiedosto.write(rivi + "\n")
            rivi = input()
        tulostiedosto.close()
        print("Nimet on kirjoitettu tiedostoon", nimi)
    except OSError:
        print("Virhe tiedoston", nimi, "kirjoittamisessa. Ohjelma paattyy.")


main()
```

Alla esimerkki ohjelman suorituksesta.

```
Ohjelma kirjoittaa vieraslistan haluamaasi tiedostoon.
Anna kirjoitettavan tiedoston nimi: vieraat.txt
Anna tallennettavat nimet.
Lopeta tyhjalla rivilla.
Aku Ankka
Roope Ankka
Minni Hiiri

Nimet on kirjoitettu tiedostoon vieraat.txt
```

Toinen esimerkkiohjelma havainnollistaa lukuarvojen tulostamista tiedostoon. Ohjelma pyytää käyttäjältä ympyröiden säteitä ja kirjoittaa tiedostoon säteen sekä ympyrän pinta-alan aina samalle riville. Ohjelmassa on käsitelty poikkeusten avulla myös se virhetilanne, että käyttäjä antaa kelvottoman luvun. Tällöin ohjelman suoritusta ei kuitenkaan keskeytetä, vaan ohjelma jatkaa lukemalla käyttäjältä seuraavan luvun.

Ohjelmassa on käytetty Pythonin valmiissa `math`-moduulissa määriteltyä vakiota `pi`. Moduuli saadaan käyttöön kirjoittamalla ohjelmatiedoston alkuun `import math`. Tämän jälkeen moduulin funktioita ja vakioita voi käyttää `math.`-etuliitteen avulla, esimerkiksi `math.pi`.

```
import math

def main():
    print("Ohjelma laskee ympyroiden pinta-aloja ja tallentaa ")
    print("ne tiedostoon.")
    nimi = input("Anna kirjoitettavan tiedoston nimi: ")
    try:
        tulostiedosto = open(nimi, "w")
        tulostiedosto.write("sade    pinta-ala\n")
        print("Anna ympyroiden sateet, lopeta negatiivisella.")
        sade = 0.0     # alkuarvo while-kaskyn ehtoa varten
        while sade >= 0.0:
            rivi = input()
            try:
                sade = float(rivi)
                if sade >= 0.0:
                    pinta_ala = math.pi * sade * sade
                    tulostiedosto.write("{:<7.2f} {:<10.2f}\n".format(
                                        sade, pinta_ala))
            except ValueError:
                print("Virhe: sateen pitaa olla desimaaliluku.")
        tulostiedosto.close()
        print("Tulokset on kirjoitettu tiedostoon", nimi)
    except OSError:
        print("Virhe tiedoston", nimi, "kirjoittamisessa. Ohjelma paattyy.")


main()
```

Seuraavana on esimerkki ohjelman suorituksesta.

```
Ohjelma laskee ympyroiden pinta-aloja ja tallentaa
ne tiedostoon.
Anna kirjoitettavan tiedoston nimi: ympyrat.txt
Anna ympyroiden sateet, lopeta negatiivisella.
14.50
suuri
Virhe: sateen pitaa olla desimaaliluku.
pieni
Virhe: sateen pitaa olla desimaaliluku.
3.80
11.00
0.0
7.25
-1
Tulokset on kirjoitettu tiedostoon ympyrat.txt
```

Tiedoston `ympyrat.txt` sisältö on tämän jälkeen seuraava:

```
sade    pinta-ala
14.50   660.52
3.80    45.36
11.00   380.13
0.00    0.00
7.25    165.13
```

6.2.3 Tiedostojen käytöstä

Usein ohjelmassa käsitellään tiedostoon tallennettua tietoa jollain tavalla, esimerkiksi tehdään siihen lisäyksiä tai poistoja. Muutetut tiedot halutaan tallentaa ohjelman seuraavaa suorituskertaa varten.

Jos käsiteltävän tiedoston koko on kohtuullinen (esimerkiksi muutama sata riviä tekstitiedostona), ei muutoksia tehdä heti suoraan tiedostoon, vaan ohjelman toiminta kannattaa suunnitella seuraavasti: Ohjelman alussa kaikki tiedostoon tallennetut tiedot luetaan sopivaan tietorakenteeseen, esimerkiksi listaan tai sanakirjarakenteeseen. Kun ohjelmassa tehdään tietoihin muutoksia, niin nämä muutokset tehdään nimenomaan tähän listaan tai muuhun käytettyyn tietorakenteeseen. Ohjelman lopussa koko lista (tai muu tietorakenne) tallennetaan tiedostoon. Tiedoston vanha sisältö siis tuhoutuu ja uusi lista kirjoitetaan aikaisempien tietojen paikalle. Tämä on huomattavasti yksinkertaisempaa kuin yrittää tehdä muutoksia keskelle tiedostoa. Jos ohjelman suoritusaika on pitkä, kannattaa toki lista tallentaa aina sopivin väliajoin, jotta sen sisältöä ei menetetä esimerkiksi ohjelman tai tietokoneen kaatuessa. Mutta tällöinkin tallennetaan koko lista kerralla, ei vain muuttuneita tietoja.

Jos käsiteltävä tiedosto on paljon laajempi, ei edellä kuvattua menettelytapaa voi noudattaa, vaan on todella kirjoitettava vain muuttuneita tietoja käsittelevä tiedoston osa kerrallaan. Näin suurten tietomäärien käsittelyssä tarvitaan kuitenkin muutenkin menetelmiä, joita ei opeteta tällä kurssilla.

6.2.4 Huomautus poikkeuksista tiedostojen käsittelyssä

Edellä esitetyissä esimerkeissä on selvitetty `OSError`-nimisen poikkeuksen avulla ne tilanteet, joissa ohjelma ei ole löytänyt haluttua tiedostoa tai tiedoston lukeminen tai kirjoittaminen ei ole onnistunut jostain muusta syystä. Jos käytössä on Pythonin versio 3.2 tai tätä vanhempi, on syytä käyttää sen sijaan `IOError`-nimistä poikkeusta.

Monissa internetin Python-esimerkeissä näkee usein käytettävän poikkeusta `FileNotFoundError` kattamaan ne tilanteet, joissa haluttua tiedostoa ei löydy. Tämä poikkeustyyppi on `OSError`-poikkeuksen alalaji (täsmällisesti ottaen `OSError`-luokan aliluokka, mutta käsitettä aliluokka ei käydä läpi tällä kurssilla). `FileNotFoundError`-poikkeus kattaa kuitenkin vain sen virhetilanteen, että haluttua tiedostoa ei löydy. `OSError`-poikkeus kattaa selvästi enemmän tiedostojen lukemiseen ja kirjoittamiseen liittyviä virhetilanteita, esimerkiksi sellaisen, että tiedosto

on kyllä olemassa, mutta ohjelman käyttäjän oikeudet eivät riitä sen lukemiseen tai sen, että tiedostoon kirjoittaminen ei onnistu siksi, että levytila on täynnä.

6.3 Tiivistelmä tärkeimmistä luvussa esitetyistä käskyistä ja rakenteista

Tähän kappaleeseen on vielä koottu lyhyesti esimerkkejä luvussa esitettyjen rakenteiden ja käskyjen käytöstä ilman selityksiä.

Poikkeusten käsittely

```
try:
    # Jono käskyjä, joista jokin tai jotkin voivat aiheuttaa tyypin
    # poikkeuksen_tyyppi poikkeuksen.
except poikkeuksen_tyyppi:
    # Käskyjä, jotka jotenkin selvittävät virhetilanteen, jos on
    # aiheutunut poikkeuksen_tyyppi-tyyppinen poikkeus.
```

Tämän kurssin esimerkeissä ja harjoitustehtävissä tarvitaan `ValueError`- ja `OSError`-tyyppisiä poikkeuksia.

Tiedoston avaaminen

```
tiedostomuuttuja = open(tiedoston_nimi, kasittelytapa)
```

esimerkiksi

```
tiedostomuuttuja = open("tekstia.txt", "r")
```

Tiedoston sulkeminen

```
tiedostomuuttuja.close()
```

Rivin lukeminen tiedostosta

```
rivi = tiedostomuuttuja.readline()
```

Tiedoston kaikkien rivien läpikäynti for-käskyn avulla

```
for rivi in tiedostomuuttuja:
    tee jotain riville rivi
```

Tyhjien merkkien (esim. rivinvaihtomerkin) poistaminen rivin lopusta

```
rivi = rivi.rstrip()
```

Tiedoston kaikkien rivien lukeminen kerralla

```
rivilista = tiedostomuuttuja.readlines()
```

Rivit ovat tämän jälkeen listassa `rivilista`, jonka kukin alkio vastaa tiedoston yhtä riviä.

Rivin lukeminen tiedostosta ja luetun rivin jakaminen osiin merkin "," kohdalta

```
rivi = tiedostomuuttuja.readline()
osat = rivi.split(",")
```

Tiedoston avaaminen kirjoittamista varten

```
tulostiedosto = open("tekstia.txt", "w")
```

(`w`:n tilalle on vaihdettava `a`, jos halutaan kirjoittaa vanhan tiedoston loppuun niin, että tiedostossa aikaisemmin ollut teksti ei häviä.)

Esimerkki rivin kirjoittamisesta tiedostoon

```
tulostiedosto.write("{:.2f} potenssiin {:d} on {:.2f}\n".format(
                    3.578, 5, 3.578 ** 5))
```

Luku 7

Luokat ja oliot

7.1 Mitä oliot ovat?

Tähän asti ohjelmissa on käsitelty yksinkertaisia muuttujia, jotka voivat saada arvokseen esimerkiksi kokonaislukuja tai desimaalilukuja, sekä listoja ja muita tietorakenteita, joissa kaikki alkiot ovat olleet keskenään samantyyppisiä. Monesti käytännön sovelluksissa joudutaan kuitenkin käsittelemään selvästi monimutkaisempia kokonaisuuksia.

Ajatellaan esimerkkinä ohjelmaa, jonka halutaan käsittelevän erään ohjelmointikurssin opiskelijoita. Kurssilla on kaksi osasuoritusta, tentti ja harjoitukset, joista kummastakin annetaan oma arvosana. Jokaisesta opiskelijasta tarvitaan ohjelmasssa ainakin nimi, opiskelijanumero, tenttiarvosana ja harjoitusarvosana. Kysymys on siitä, miten opiskelijoiden tietoja kannattaisi käsitellä kirjoitettavassa ohjelmassa.

Yksi tapa olisi tehdä neljä eri listaa: yksi opiskelijoiden nimiä, yksi opiskelijanumeroita, yksi tenttiarvosanoja ja yksi harjoitusarvosanoja varten. Saman opiskelijan tiedot olisivat eri listoissa aina samalla indeksillä. Jos esimerkiksi opiskelijan nimi on listassa `nimet` indeksillä `5`, on myös hänen harjoitusarvosanansa listassa `harjoitusarvosanat` indeksillä `5`. Tämä tapa on kuitenkin huono, koska siinä yhden opiskelijan tiedot on hajautettu neljään eri listaan. Ohjelmassa ei ole järkevää tapaa käsitellä yhden opiskelijan tietoja yhtenä kokonaisuutena, vaan tiedot on aina yhdistettävä neljästä eri listasta. Tapa on myös hyvin altis virheille. Jos listoista poistetaan yhden opiskelijan tiedot, on muistettava poistaa alkio jokaisesta neljästä listasta. Muussa tapauksessa eri opiskelijoiden eri tiedot menevät täysin sekaisin.

Toinen tapa olisi koota yhden opiskelijan tiedoista yksi lista, jossa on ensimmäisenä alkiona opiskelijan nimi, toisena alkioina opiskelijan opiskelijanumero, kolmantena alkiona tenttiarvosana ja neljäntenä alkiona opiskelijan harjoitusarvosana. Listan alkiot voivat olla keskenään erityyppisiä, joten ei haittaa, vaikka nimi ja opiskelijanumero ovat merkkijonoja ja arvosanat kokonaislukuja. Kurssin kaikista opiskelijoista muodostetaan puolestaan suurempi lista, jonka alkiot ovat yksittäisten opiskelijoiden tietoja sisältäviä listoja.

Toinen tapa on selvästi ensimmäistä tapaa parempi. Siinä yhden opiskelijan tiedot on koottu yhdeksi kokonaisuudeksi. Tässäkin tavassa on kuitenkin puutteita. Esimerkiksi se, missä järjestyksessä opiskelijan eri tiedot on annettu listassa, perustuu vain

sopimukseen, joka ohjelmoijan pitää muistaa. Olisi helpompaa, jos voitaisiin selkeästi nimetä, että nyt käsitellään opiskelijan nimeä sen sijaan, että puhuttaisiin vain listan indeksillä `0` olevasta alkiosta. Lisäksi listatoteutus ei vielä mitenkään määrittele sitä, millaisia operaatioita opiskelijan tiedoille voidaan tehdä. Olisi hyödyllistä, jos voitaisiin selkeästi määritellä esimerkiksi se, miten opiskelijan tenttiarvosanaa voidaan muuttaa. Tässä määrittelyssä voitaisiin ottaa huomioon se, että arvosanan on oltava määrätyllä välillä ja estää antamasta opiskelijalle virheellisiä (esimerkiksi negatiivisia tai liian suuria) arvosanoja.

Olio-ohjelmointi tarjoaa tavan samalla säilyttää toisen ratkaisutavan edut (yhden opiskelijan tietoja käsitellään yhtenä kokonaisuutena) ja ratkaista siinä olevat ongelmat. Olio-ohjelmoinnissa jokaista kurssin oikeaa opiskelijaa vastaa ohjelmassa yksi opiskelijaolio. Opiskelijaolion eri tiedot (nimi, opiskelijanumero, harjoitusarvosana ja tenttiarvosana) on selvästi nimetty ja ohjelmassa on myös selvästi määritelty, millaisia operaatioita opiskelijaolioille voidaan suorittaa ja miten nämä operaatiot tehdään.

Kurssin kaikkia opiskelijoita käsittelevässä ohjelmassa luodaan siis jokaista oikeaa opiskelijaa kohti opiskelijaolio. Lisäksi voidaan tehdä lista, joka sisältää kaikki luodut opiskelijaoliot ja päästä näin käsittelemään kurssin kaikkia opiskelijoita. Tarkastelemme tällaista listaa vasta myöhemmin ja aloitamme ohjelmilla, jotka käsittelevät vain paria opiskelijaa.

Jo tässä vaiheessa on syytä huomata, että tällä kurssilla käsitellään olio-ohjelmoinnin mahdollisuuksista vain hyvin pientä osaa. Olio-ohjelmointi sisältää paljon muutakin (esimerkiksi perintä), mikä ei kuulu tämän kurssin aihealueeseen.

7.2 Luokan määrittely ja olioiden käsittely

Jos haluamme tehdä ohjelman, joka käsittelee opiskelijaolioita, on ohjelmassa jotenkin määriteltävä se, millaisia opiskelijaoliot ovat ja mitä toimenpiteitä niille voi tehdä. Tämä tehdään luokassa, jonka nimi on `Opiskelija`. Kun on kirjoitettu `Opiskelija`-luokka, voidaan luoda `Opiskelija`-olioita ja suorittaa niille erilaisia toimenpiteitä.

Yleisesti siis luokassa määritellään, millaisia ominaisuuksia sen luokan olioilla on ja mitä toimenpiteitä luokan olioille voi tehdä. Luokkaa voidaan tavallaan verrata jonkin koneen piirustuksiin ja olioita piirustusten perusteella rakennettuihin koneisiin. Samoin kuin yksien piirustusten perusteella voidaan rakentaa monta konetta, yhdestä luokasta voidaan luoda monta oliota.

Tarkastelemme opiskelijaesimerkin avulla, miten luokka kirjoitetaan ja miten luokan olioita voidaan luoda ja käyttää. *Kenttien* avulla kerrotaan, mitä ominaisuuksia luokan olioilla on ja mikä on jonkin ominaisuuden arvo tietyllä oliolla. Esimerkiksi tässä määriteltävällä `Opiskelija`-oliolla on kentät `__nimi`, `__opiskelijanumero`, `__tenttiarvosana` ja `__harjoitusarvosana`, joiden arvoksi voidaan tallentaa käsiteltävän `Opiskelija`-olion nimi, opiskelijanumero sekä tentti- ja harjoitusarvosana vastaavasti. Jokaisella luotavalla oliolla on omat kenttien arvot, jotka eivät riipu toisten saman luokan olioiden kenttien arvoista. Olion kenttien arvot säilyvät niin kauan kuin olio on olemassa (ne eivät siis häviä esimerkiksi jostain funktioista poistuttaessa), ellei ohjelmassa erikseen muuteta jonkin kentän arvoa. Edellä luetellut

kenttien nimet alkavat kahdella alaviivamerkillä (_). Syy tähän kerrotaan vähän myöhemmin.

Olion kenttiä voi käyttää luokan sisällä samaan tapaan kuin tavallisia muuttujia. Kentän nimen edessä kuitenkin kerrotaan aina, minkä olion kentästä on kysymys. Tämä tehdään pistenotaation avulla. Jos halutaan käsitellä muuttujan `oliomuuttuja` viittaman olion kenttää `kentta`, kirjoitetaan `oliomuuttuja.kentta`. Usein luokan sisällä olevissa metodeissa käsitellään sitä oliota, jota ollaan juuri luomassa tai jolle on juuri kutsuttu luokan jotain metodia. Tällaiseen olioon viitataan nimellä `self`. Tätä asiaa selvennetään lisää, kun käydään läpi luokan `Opiskelija` sisältöä.

Luokan määrittely aloitetaan otsikolla, johon kuuluu sana `class` ja luokan nimi. Sen jälkeen tulee kaksoispiste. Yleinen tapa on aloittaa luokan nimi isolla kirjaimella.

```
class Opiskelija:
```

Seuraavien rivien sisennysten avulla osoitetaan, mitkä rivit kuuluvat tämän luokan määrittelyyn. Luokan määritettelyssä kerrotaan, millaisia toimenpiteitä luokan olioille voidaan suorittaa. Tämä määritellään metodien avulla. Samaan tapaan kuin funktio, metodi on kappale koodia, jolle on annettu oma nimi. Metodi liittyy kuitenkin aina johonkin luokkaan ja sitä kutsutaan aina jollekin tämän luokan oliolle. Metodia kutsutaan vähän eri tavalla kuin funktioita. Aikaisemmin on ollut jo esimerkkejä siitä, kuinka Pythonin valmiita metodeita on kutsuttu esimerkiksi listoja, merkkijonoja ja tiedostoja käsiteltäessä. Nyt olioiden yhteydessä tutustumme siihen, miten metodeita voi kirjoittaa itse.

Luokan määrittelyssä kerrotaan yleensä aluksi, mitä tehdään, kun luodaan uusi tämän luokan olio. Ohjelmoijan on kerrottava esimerkiksi se, millaisia alkuarvoja annetaan olion kentille. Tämä tehdään metodissa, jonka nimi on `__init__` (Nimen alussa ja lopussa on kaksi alaviivamerkkiä `_`.) Kaikilla metodeilla on ensimmäinen parametri, jonka nimi on `self`. Tämä tarkoittaa sitä oliota, jota ollaan juuri luomassa tai käsittelemässä. Tämän lisäksi metodilla voi olla muita parametreja, joiden avulla metodille annetaan tietoa käytettävistä arvoista. Esimerkiksi metodin `__init__` parametrien avulla kerrotaan usein, mitä alkuarvoja luotavan olion kentille annetaan.

Kirjoitetaan luokan `Opiskelija` metodi `__init__` siten, että luotavalle `Opiskelija`-oliolle annettava nimi ja opiskelijanumero annetaan metodin parametrina. Uuden opiskelijan tentti- ja harjoitusarvosanoiksi asetetaan aluksi `0`.

```
    def __init__(self, annettu_nimi, numero):
        self.__nimi = annettu_nimi
        self.__opiskelijanumero = numero
        self.__tenttiarvosana = 0
        self.__harjoitusarvosana = 0
```

Tässä siis ilmaus `self.kentan_nimi` tarkoittaa sen `Opiskelija`-olion kenttää, jota ollaan juuri luomassa. Kun `Opiskelija`-luokkaan on kirjoitettu tällainen `__init__`-metodi, voidaan uusia `Opiskelija`-olioita luoda seuraavaan tapaan (yleensä luominen tapahtuu luokan ulkopuolella, esimerkiksi pääohjelmassa):

```
kurssilainen1 = Opiskelija("Minna Lahti", "77112F")
```

Tässä `kurssilainen1` on muuttuja, joka pannaan sijoituskäskyllä viittaamaan luotavaan olioon. Sen kautta on myöhemmin mahdollisuus päästä käsittelemään nyt luotavaa oliota. Sijoituskäskyn oikealla puolella oleva osa saa aikaan varsinaisen olion luonnin. Python-tulkki luo uuden olion ja suorittaa sitten `__init__`-metodissa olevat käskyt niin, että parametrin `annettu_nimi` arvo on merkkijono `"Minna Lahti"` ja parametrin `numero` arvo merkkijono `"77112F"`. Metodin `__init__` ensimmäiselle parametrille `self` ei anneta oliota luodessa arvoa, vaan parametrilla tarkoitetaan sitä oliota, jota ollaan juuri luomassa.

Tarkastellaan sitten luokan muita metodeja. Jotta luodun `Opiskelija`-olion tentti- ja harjoitusarvosanoja voisi myöhemmin muuttaa, määritellään metodit tentti- ja harjoitusarvosanojen muuttamista varten. Alla on kirjoitettu metodi, jonka avulla voi muuttaa olion `__tenttiarvosana`-kenttää. Metodi saa parametrina opiskelijalle annettavan uuden tenttiarvosanan. Ennen kentän arvon muuttamista metodi tarkistaa, että parametri on sallitulla välillä 0–5. Jos parametri ei ole tällä välillä, metodi ei tee mitään. Silloin käsiteltävän olion vanha `__tenttiarvosana`-kentän arvo jää voimaan.

```
    def muuta_tenttiarvosana(self, arvosana):
        if 0 <= arvosana <= 5:
            self.__tenttiarvosana = arvosana
```

Metodin ensimmäinen parametri `self` tarkoittaa sitä `Opiskelija`-oliota, jolle metodia ollaan kutsumassa. Jos on aikaisemmin luotu `Opiskelija`-olio ja pantu muuttuja `kurssilainen1` viittaamaan siihen, voidaan metodia kutsua seuraavasti:

```
kurssilainen1.muuta_tenttiarvosana(4)
```

Usein ohjelmassa on luotu useita saman luokan olioita. Voitaisiin esimerkiksi luoda toinen `Opiskelija`-olio ja panna muuttuja `kurssilainen2` viittaamaan siihen kirjoittamalla

```
kurssilainen2 = Opiskelija("Matti Virtanen", "78414C")
```

Silloin tämän jälkimmäisen opiskelijan tenttiarvosanaa voitaisiin muuttaa kirjoittamalla

```
kurssilainen2.muuta_tenttiarvosana(3)
```

Metodin kutsussa ennen pistettä oleva muuttuja määrää sen, mille oliolle metodia kutsutaan. Aikaisemmin esitetty ensimmäinen `muuta_arvosana`-kutsu muutti Maija Lahden tenttiarvosanaa, kun taas yllä esitetty kutsu muuttaa Matti Virtasen tenttiarvosanaa. Jokaisella oliolla on omat kopionsa kenttien arvoista ja muutokset yhden olion kentän arvossa eivät mitenkään vaikuta toisten saman luokan olioiden kenttien arvoihin.

Kannattaa huomata, että metodien kutsuissa ei ole lainkaan sulkujen sisällä annettu käsiteltävää oliota eli parametrin `self` arvoa. Se on annettu kutsussa jo ennen pistettä ja metodin nimeä. Sen sijaan metodin muiden parametrien arvot on annettu ihan samalla tavalla kuin funktioillakin.

Metodi `muuta_harjoitusarvosana` kirjoitetaan vastaavalla tavalla. Ainoastaan muutettavan kentän nimi on eri.

Kirjoitetaan sitten luokkaan metodi, jonka avulla voidaan laskea opiskelijan kokonaisarvosana. Se lasketaan tentti- ja harjoitusarvosanojen keskiarvona. Aluksi kuitenkin tarkistetaan, onko toisen tai molempien osasuoritusten arvosana 0. Siinä tapauksessa kokonaisarvosanaksi tulee 0 keskiarvosta riippumatta. Metodi palauttaa arvonaan lasketun keskiarvon. Metodin paluuarvoa voidaan käyttää hyväksi aivan samalla tavalla kuin funktioiden paluuarvoja.

```
    def laske_kokonaisarvosana(self):
        if self.__tenttiarvosana == 0 or self.__harjoitusarvosana == 0:
            arvosana = 0
        else:
            arvosana = (self.__tenttiarvosana +
                        self.__harjoitusarvosana + 1) // 2
        return arvosana
```

Keskiarvon laskemisessa lisätään osasuoritusarvosanojen summaan ykkönen. Näin saadaan aikaan se, että puolet numerot pyöristyvät ylöspäin. Laskemisessa suoritetaan kokonaislukujen jakolasku, jolloin saatu arvosana on aina kokonaisluku.

Alla on esimerkki metodin kutsumisesta ja sen paluuarvon tulostamisesta. (Oletetaan jälleen, että ohjelmassa on luotu aikaisemmin mainitut oliot.)

```
print("Matin kurssiarvosana on", kurssilainen2.laske_kokonaisarvosana())
```

Mainittujen metodien lisäksi luokkaan määritellään metodeita, joiden avulla voi luokan ulkopuolelta selvittää jonkin kentän arvon. Nämä metodit siis vain palauttavat yhden kentän arvon eivätkä tee mitään muuta. Alla esimerkkinä metodit `kerro_nimi` ja `kerro_tenttiarvosana`. Myös kahdelle muulle kentälle määritellään vastaavat metodit.

```
    def kerro_nimi(self):
        return self.__nimi


    def kerro_tenttiarvosana(self):
        return self.__tenttiarvosana
```

Seuraavaksi koko luokan määrittely yhtenä kokonaisuutena. Määrittelyyn on lisätty myös kommentit.

```
# Luokka Opiskelija kuvaa eraan ohjelmointikurssin
# yhta opiskelijaa.

class Opiskelija:

    # Metodi __init__ antaa alkuarvot olion kentille.
    # Luotavalle opiskelijalle annettava nimi ja opiskelijanumero
    # annetaan metodin parametreina.

    def __init__(self, annettu_nimi, numero):
        self.__nimi = annettu_nimi
        self.__opiskelijanumero = numero
        self.__tenttiarvosana = 0
        self.__harjoitusarvosana = 0


    # Metodi palauttaa opiskelijan nimen.

    def kerro_nimi(self):
        return self.__nimi


    # Metodi palauttaa opiskelijan opiskelijanumeron.

    def kerro_opiskelijanumero(self):
        return self.__opiskelijanumero


    # Metodi palauttaa opiskelijan tenttiarvosanan.

    def kerro_tenttiarvosana(self):
        return self.__tenttiarvosana


    # Metodi palauttaa opiskelijan harjoitusarvosanan.

    def kerro_harjoitusarvosana(self):
        return self.__harjoitusarvosana


    # Metodi muuttaa opiskelijan tenttiarvosanan. Uusi arvosana
    # annetaan metodin parametrina.

    def muuta_tenttiarvosana(self, arvosana):
        if 0 <= arvosana <= 5:
            self.__tenttiarvosana = arvosana


    # Metodi muuttaa opiskelijan harjoitusarvosanan. Uusi arvosana
    # annetaan metodin parametrina.
```

```
    def muuta_harjoitusarvosana(self, arvosana):
        if 0 <= arvosana <= 5:
            self.__harjoitusarvosana = arvosana


    # Metodi laskee ja palauttaa opiskelijan kokonaisarvosanan.

    def laske_kokonaisarvosana(self):
        if self.__tenttiarvosana == 0 or self.__harjoitusarvosana == 0:
            arvosana = 0
        else:
            arvosana = (self.__tenttiarvosana +
                        self.__harjoitusarvosana + 1) // 2
        return arvosana
```

Seuraava pääohjelma luo kaksi `Opiskelija`-oliota niin, että luotavien olioiden tiedot kysytään käyttäjältä. Ohjelma laskee opiskelijoiden kokonaisarvosanat ja tulostaa ne sekä opiskelijoiden muut tiedot. Arvosanojen lukemiseen käyttäjältä on määritelty oma apufunktio. Näin mahdollisten virheellisten syötteiden käsittely voidaan hoitaa helposti. Ohjelmaa kirjoittaessa on oletettu, että pääohjelma on samassa tiedostossa luokan `Opiskelija` kanssa. Jos pääohjelma on kirjoitettu toiseen tiedostoon (kuten usein käytännössä tehdään), pitää siihen tehdä pieniä lisäyksiä, joista kerrotaan tarkemmin vähän myöhemmin.

```
def lue_kokonaisluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = int(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen kokonaisluku!")
            print("Anna uusi!")
    return luku

def main():
    nimi1 = input("Anna 1. opiskelijan nimi: ")
    op_nro1 = input("Anna 1. opiskelijan opiskelijanumero: ")
    kurssilainen1 = Opiskelija(nimi1, op_nro1)
    nimi2 = input("Anna 2. opiskelijan nimi: ")
    op_nro2 = input("Anna 2. opiskelijan opiskelijanumero: ")
    kurssilainen2 = Opiskelija(nimi2, op_nro2)

    print("Anna 1. opiskelijan tenttiarvosana.")
    tentti1 = lue_kokonaisluku()
    kurssilainen1.muuta_tenttiarvosana(tentti1)
    print("Anna 1. opiskelijan harjoitusarvosana.")
    harjoitus1 = lue_kokonaisluku()
```

```
    kurssilainen1.muuta_harjoitusarvosana(harjoitus1)

    print("Anna 2. opiskelijan tenttiarvosana.")
    tentti2 = lue_kokonaisluku()
    kurssilainen2.muuta_tenttiarvosana(tentti2)
    print("Anna 2. opiskelijan harjoitusarvosana.")
    harjoitus2 = lue_kokonaisluku()
    kurssilainen2.muuta_harjoitusarvosana(harjoitus2)

    print("1. opiskelijan tiedot:")
    print(kurssilainen1.kerro_opiskelijanumero())
    print(kurssilainen1.kerro_nimi())
    print("Tenttiarvosana:", kurssilainen1.kerro_tenttiarvosana())
    print("Harjoitusarvosana:",
          kurssilainen1.kerro_harjoitusarvosana())
    print("Kurssiarvosana:", kurssilainen1.laske_kokonaisarvosana())

    print("2. opiskelijan tiedot:")
    print(kurssilainen2.kerro_opiskelijanumero())
    print(kurssilainen2.kerro_nimi())
    print("Tenttiarvosana:", kurssilainen2.kerro_tenttiarvosana())
    print("Harjoitusarvosana:",
          kurssilainen2.kerro_harjoitusarvosana())
    print("Kurssiarvosana:", kurssilainen2.laske_kokonaisarvosana())


main()
```

Seuraavaksi esimerkki ohjelman suorituksesta:

```
Anna 1. opiskelijan nimi: Minni Hiiri
Anna 1. opiskelijan opiskelijanumero: 11223F
Anna 2. opiskelijan nimi: Hessu Hopo
Anna 2. opiskelijan opiskelijanumero: 33441C
Anna 1. opiskelijan tenttiarvosana.
3
Anna 1. opiskelijan harjoitusarvosana.
5
Anna 2. opiskelijan tenttiarvosana.
tekstia
Virheellinen kokonaisluku!
Anna uusi!
1
Anna 2. opiskelijan harjoitusarvosana.
2
1. opiskelijan tiedot:
11223F
Minni Hiiri
Tenttiarvosana: 3
Harjoitusarvosana: 5
```

```
Kurssiarvosana: 4
2. opiskelijan tiedot:
33441C
Hessu Hopo
Tenttiarvosana: 1
Harjoitusarvosana: 2
Kurssiarvosana: 2
```

Usein jokainen luokka kirjoitetaan omaan moduuliinsa (käytännössä omaan tiedostoonsa) ja myös pääohjelma eri moduuliin kuin luokkien määrittelyt. Näin tehdään esimerkiksi siksi, että omassa moduulissa olevaa luokkaa on helppo käyttää hyväksi muissakin ohjelmissa kuin siinä, jota varten se on alunperin tehty. Jos `Opiskelija`-luokka kirjoitetaan omaan moduuliinsa `opiskelija` (tiedoston nimeksi tulee tällöin `opiskelija.py`) ja pääohjelma toiseen moduuliin, on pääohjelmatiedoston alkuun kirjoitettava rivi

```
import opiskelija
```

jotta toisessa moduulissa oleva luokka saataisiin käyttöön. Lisäksi `Opiskelija`-olioita luodessa on annettava luokan sisältävän moduulin nimi seuraavasti:

```
kurssilainen1 = opiskelija.Opiskelija(nimi1, op_nro1)
# muita koodiriveja
kurssilainen2 = opiskelija.Opiskelija(nimi2, op_nro2)
```

Muuten pääohjelma säilyy samanlaisena kuin ennenkin. Esimerkiksi metodien kutsussa ei tarvita moduulin nimeä.

On myös toinen tapa kirjoittaa import-käsky pääohjelmatiedoston alkuun:

```
from opiskelija import *
```

Jos käytetään tätä tapaa, ei `Opiskelija`-olioita luodessa tarvitse antaa luokan sisältävän moduulin nimeä, vaan oliot luodaan samalla tavalla kuin jos luokan koodi olisi samassa moduulissa:

```
kurssilainen1 = Opiskelija(nimi1, op_nro1)
# muita koodiriveja
kurssilainen2 = Opiskelija(nimi2, op_nro2)
```

Merkkijonoesitys oliosta

Hyvin monessa ohjelmassa halutaan jossain vaiheessa tulostaa käsiteltävien olioiden kaikkien kenttien arvot. Tämä voidaan toki tehdä käyttämällä kenttien arvot palauttavia metodeita ja tulostamalla niiden palauttamat arvot, kuten edellisessä esimerkissä oli tehty. Se on kuitenkin aika työläs tapa erityisesti silloin, kun kenttiä on paljon. Tulostaminen voidaan tehdä helpommin, jos luokkaan kirjoitetaan erityinen metodi `__str__`. Metodi kirjoitetaan siten, että se palauttaa merkkijonon, joka

sisältää käsiteltävän olion kuvauksen, esimerkiksi kenttien arvot tai muuta haluttua tietoa oliosta. Toisin sanoen metodi tekee ja palauttaa oliosta merkkijonoesityksen.

Luokkaan `Opiskelija` voidaan kirjoittaa (luokan sisään) seuraava `__str__`-metodi:

```
    # Metodi palauttaa merkkijonoesityksen opiskelijan tiedoista.

    def __str__(self):
        mjono = self.__nimi + ", " + self.__opiskelijanumero + \
                ", tenttiarvosana: " + str(self.__tenttiarvosana) + \
                ", harjoitusarvosana: " + str(self.__harjoitusarvosana)
        return mjono
```

Jos metodi on kirjoitettu, luokan olioiden tiedot voidaan tulostaa (yleensä luokan ulkopuolella, esimerkiksi pääohjelmassa) antamalla print-käskyssä vain sen muuttujan nimi, joka viittaa tulostettavaan olioon, esimerkiksi

```
print(kurssilainen1)
```

Näin edellisen esimerkkipääohjelman lopussa oleva osa, joka tulostaa luotujen `Opiskelija`-olioiden tiedot voidaan korvata seuraavalla, lyhyemmällä ja yksinkertaisemmalla ohjelmakoodilla:

```
    print("1. opiskelijan tiedot:")
    print(kurssilainen1)
    print("Kurssiarvosana:", kurssilainen1.laske_kokonaisarvosana())

    print("2. opiskelijan tiedot:")
    print(kurssilainen2)
    print("Kurssiarvosana:", kurssilainen2.laske_kokonaisarvosana())
```

Kokonaisarvosanojen tulostaminen on tässäkin koodissa erikseen, koska `__str__`-metodin palauttama merkkijono ei sisällä kokonaisarvosanaa.

Jos pääohjelman loppu muutetaan tällaiseksi, niin edellisen esimerkkiajon suorituksen loppu näyttää seuraavalta:

```
1. opiskelijan tiedot:
Minni Hiiri, 11223F, tenttiarvosana: 3, harjoitusarvosana: 5
Kurssiarvosana: 4
2. opiskelijan tiedot:
Hessu Hopo, 33441C, tenttiarvosana: 1, harjoitusarvosana: 2
Kurssiarvosana: 2
```

Tulostus on vähän erilainen kuin edellisessä esimerkkiajossa, koska `__str__`-metodin palauttama merkkijono poikkeaa vähän edellisen esimerkkiajon pääohjelman tulostuksista.

7.3 Toinen esimerkki

Tarkastellaan toisena esimerkkinä luokkaa `Vesisailio`, joka kuvaa yhden vesisäiliön toimintaa. Vesisäiliöön voi lisätä vettä ja siitä voi poistaa vettä. Vesisäiliöllä on kuitenkin koko, eikä säiliöön voi lisätä vettä enempää kuin siihen mahtuu. Säiliöstä ei voi myöskään poistaa vettä enempää kuin mitä siinä on. Esimerkin mielekkyys ei välttämättä tunnu heti ilmeiseltä, mutta esimerkkiluokkaa voi käyttää vähän muuttamalla sellaisissa sovelluksissa, joissa tarkastellaan jonkin tuotteen määrää varastossa. Lisäksi esimerkistä on helppo muokata luokka vaikka pankkitilin kuvaamiseen.

Luokan määrittely aloitetaan jälleen luokan otsikolla:

```
class Vesisailio:
```

Metodissa `__init__` annetaan alkuarvot olion kentille `__koko` ja `__maara`. Näistä ensimmäinen kertoo käsiteltävän vesisälilön koon ja toinen veden määrän säiliössä. Luotavalle säiliölle annettava koko kerrotaan metodin parametrin avulla. Metodi tarkistaa, että annettu koko on vähintään 0. Se ei siis luo säiliöitä, joilla on negatiivinen koko. Uuden säiliön veden määräksi sijoitetaan 0.

```
    def __init__(self, annettu_koko):
        if annettu_koko >= 0.0:
            self.__koko = annettu_koko
        else:
            self.__koko = 0.0
        self.__maara = 0.0
```

Luokkaan määritellään metodit `kerro_koko` ja `kerro_maara`, jotka palauttavat vastaavien kenttien arvot:

```
    def kerro_koko(self):
        return self.__koko


    def kerro_maara(self):
        return self.__maara
```

Metodin `lisaa` avulla voida lisätä veden määrää säiliössä. Metodi tutkii ensin, että sille annettu parametri on vähintään 0 (negatiivista määrää ei voi lisätä). Tämän jälkeen metodi tutkii, paljonko vesisäiliöön vielä mahtuu vettä. Jos parametrina annettu lisättävä määrä on korkeintaan yhtä suuri kuin säiliöön mahtuva määrä, koko haluttu määrä lisätään. Jos lisättävä määrä on suurempi kuin säiliöön mahtuva määrä, lisätään vain niin paljon kuin säiliöön mahtuu. Molemmissa tapauksissa säiliössä olevan veden määrää (kenttä `__maara`) kasvatetaan lisätyllä määrällä ja metodi palauttaa arvonaan oikeasti lisätyn määrän.

```
    def lisaa(self, paljonko):
        if paljonko > 0.0:
            mahtuu = self.__koko - self.__maara
            if paljonko <= mahtuu:
                self.__maara += paljonko
                return paljonko
            else:
                self.__maara = self.__koko
                return mahtuu
        else:
            return 0.0
```

Metodin `poista` avulla voidaan poistaa vettä säiliöstä. Poistettava määrä annetaan metodille parametrina. Jälleen ensin tutkitaan, että poistettava määrä on vähintään 0 (negatiivista määrää ei voi poistaa). Sen jälkeen tutkitaan, onko parametrina annettu poistettava määrä suurempi kuin säiliössä tällä hetkellä oleva vesimäärä. Jos on, säiliöstä poistetaan vain niin paljon vettä kuin siellä on. Muussa tapauksessa säiliöstä poistetaan vettä haluttu määrä. Molemmissa tapauksissa metodi vähentää `__maara`-kentän arvoa ja palauttaa arvonaan säiliöstä oikeasti poistetun vesimäärän.

```
    def poista(self, paljonko):
        if paljonko > 0.0:
            if paljonko > self.__maara:
                poistetaan = self.__maara
                self.__maara = 0.0
                return poistetaan
            else:
                self.__maara -= paljonko
                return paljonko
        else:
            return 0.0
```

Lisäksi määritellään metodi `__str__`, joka palauttaa merkkijonon, joka sisältää käsiteltävän `Vesisailio`-olion kenttien arvot.

```
    def __str__(self):
        mjono = "Sailio: vetta " + str(self.__maara) + \
                " / " + str(self.__koko) + " l"
        return mjono
```

Luokan määrittely loppuu tähän.

Seuraavaksi on kirjoitettu pääohjelma, joka luo kaksi `Vesisailio`-oliota sekä lisää niihin vettä ja poistaa niistä vettä. Vesisäiliöiden koot sekä lisättävät ja poistettavat määrät kysytään käyttäjältä. Ohjelma myös tulostaa lisätyt ja poistetut määrät sekä vesisäiliöiden tiedot ohjelman lopussa. Ohjelma on kirjoitettu eri moduuliin kuin luokka `Vesisailio`. Alla olevassa koodissa on oletettu, että luokka on kirjoitettu moduuliin `vesisailio` (tiedoston nimi `vesisailio.py`). Käyttäjältä kysyttävien lukujen lukemiseen on kirjoitettu apufunktio, joka myös käsittelee mahdolliset virheelliset syötteet:

```
import vesisailio

def lue_desimaaliluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = float(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen desimaaliluku!")
            print("Anna uusi!")
    return luku

def main():
    print("Anna ensimmaisen sailion koko.")
    koko1 = lue_desimaaliluku()
    sailio1 = vesisailio.Vesisailio(koko1)
    print("Anna toisen sailion koko.")
    koko2 = lue_desimaaliluku()
    sailio2 = vesisailio.Vesisailio(koko2)

    print("Lisataan vetta 1. sailioon.")
    print("Anna lisattava maara.")
    lisays = lue_desimaaliluku()
    lisattiin = sailio1.lisaa(lisays)
    print("Sailioon lisattiin {:.2f} l.".format(lisattiin))
    print("Lisataan vetta 2. sailioon.")
    print("Anna lisattava maara.")
    lisays = lue_desimaaliluku()
    lisattiin = sailio2.lisaa(lisays)
    print("Sailioon lisattiin {:.2f} l.".format(lisattiin))

    print("Poistetaan vetta 1. sailiosta.")
    print("Anna poistettava maara.")
    poisto = lue_desimaaliluku()
    poistettiin = sailio1.poista(poisto)
    print("Sailiosta poistettiin {:.2f} l.".format(poistettiin))
    print("Poistetaan vetta 2. sailiosta.")
    print("Anna poistettava maara.")
    poisto = lue_desimaaliluku()
    poistettiin = sailio2.poista(poisto)
    print("Sailiosta poistettiin {:.2f} l.".format(poistettiin))

    print("Ensimmaisen sailion tiedot:")
    print(sailio1)
    print("Toisen sailion tiedot:")
    print(sailio2)

main()
```

Seuraavaksi esimerkki ohjelman suorituksesta. Huomaa esimerkistä, kuinka säiliöihin ei todellakaan lisätä vettä enempää kuin mitä mahtuu eikä poisteta enempää kuin mitä säiliössä on. Todellisuudessa lisätyt ja poistetut määrät on pystytty tulostamaan käyttämällä hyväksi metodien paluuarvoja.

```
Anna ensimmaisen sailion koko.
50.0
Anna toisen sailion koko.
30.0
Lisataan vetta 1. sailioon.
Anna lisattava maara.
28.5
Sailioon lisattiin 28.50 l.
Lisataan vetta 2. sailioon.
Anna lisattava maara.
jotain
Virheellinen desimaaliluku!
Anna uusi!
41.0
Sailioon lisattiin 30.00 l.
Poistetaan vetta 1. sailiosta.
Anna poistettava maara.
29.5
Sailiosta poistettiin 28.50 l.
Poistetaan vetta 2. sailiosta.
Anna poistettava maara.
11.1
Sailiosta poistettiin 11.10 l.
Ensimmaisen sailion tiedot:
Sailio: vetta 0.0 / 50.0 l
Toisen sailion tiedot:
Sailio: vetta 18.9 / 30.0 l
```

7.4 Olio metodin parametrina: luokka `Tasovektori`

Halutaan kirjoittaa ohjelma, jonka avulla voi käsitellä kaksiulotteisia vektoreita. Vektorit ovat siis muotoa $a\bar{i} + b\bar{j}$, missä a kertoo vektorin x-akselin suuntaisen komponentin kertoimen ja b vektorin y-akselin suuntaisen komponentin kertoimen. Luokan koodissa näitä kertoimia on kuvattu kentillä `__x_kerroin` ja `__y_kerroin`.

Yksi vektoreille usein tarvittava toimenpide on pistetulon laskeminen. Pistetulo lasketaan kahden vektorin välillä. Kun siis määritellään pistetuloa laskevaa metodia `pistetulo`, pitää sille välittää tieto kahdesta eri vektorista, joiden välillä pistetulo lasketaan. Toinen vektoreista on luonnollisesti metodin ensimmäinen parametri `self`, joka annetaan metodia kutsuttaessa ennen pistettä ja metodin nimeä. Toinen vektori voidaan välittää metodille parametrina ihan samalla tavalla kuin metodille annetaan parametrina esimerkiksi kokonaisluku tai desimaaliluku.

Pistetulon laskevassa metodissa pitää selvittää molemmista vektoreissa niiden `__x_kerroin` ja `__y_kerroin`-kenttien arvot. Tämä voidaan tehdä suoraan käyttämällä pistenotaatiota seuraavasti:

```
    def pistetulo(self, toinen_vektori):
        tulo = self.__x_kerroin * toinen_vektori.__x_kerroin + \
               self.__y_kerroin * toinen_vektori.__y_kerroin
        return tulo
```

Tässä on siis selvitetty jonkin olion kentän arvo siten, että on annettu ensin sen oliomuuttujan nimi, joka viittaa käsiteltävään olioon, sitten piste ja sen jälkeen halutun kentän nimi. Tässä nimi `self` tarkoittaa sitä oliota, jolle metodia kutsutaan (johon viittaavan muuttujan nimi on metodin kutsussa ennen pistettä) ja nimi `toinen_vektori` metodille parametrina annettua oliota (sitä, jonka nimi on metodin kutsussa sulkujen sisällä).

Toinen tapa on käyttää kentän arvon selvittämiseen olion luokan omaa metodia. Alla olevassa metodissa parametrina annetun olion kenttien arvot on selvitetty käyttämällä saman luokan metodeita `kerro_x_kerroin` ja `kerro_y_kerroin`.

```
    def pistetulo2(self, toinen_vektori):
        tulo = self.__x_kerroin * toinen_vektori.kerro_x_kerroin() + \
               self.__y_kerroin * toinen_vektori.kerro_y_kerroin()
        return tulo
```

Jos siis metodeia kutsutaan esimerkiksi

```
a_vektori.pistetulo(b_vektori)
```

tai

```
a_vektori.pistetulo2(b_vektori)
```

niin molemmissa tapauksissa metodia lähdetään suorittamaan siten, että metodissa nimi `self` tarkoittaa sitä oliota, johon muuttuja `a_vektori` viittaa ja nimi `toinen_vektori` sitä oliota, johon muuttuja `b_vektori` viittaa.

Seuraavaksi koko luokan koodi. Luokkaan on jätetty vain yksi versio pistetulon laskevasta metodista. Sen lisäksi luokkaan on määritelty metodi `laske_pituus`, joka laskee vektorin pituuden Pythagoraan lauseen avulla. Laskemisessa tarvitaan neliöjuuren laskevaa funktiota, minkä takia luokassa on otetty käyttöön Pythonin valmis moduuli `math` ja siihen kuuluva neliöjuuren laskeva funktio `sqrt`. Luokassa on myös metodi `kerro_luvulla`, joka kertoo vektorin molemmat komponentit parametrina annetulla luvulla. Lisäksi luokkaan on määritelty metodi `__str__` tekemään oliosta merkkijonoesitys. Vektoria kuvaava merkkijono on koottu niin, että jos y-akselin suuntaisen komponentin kerroin on negatiivinen, vektoria kuvaavassa merkkijonossa on komponenttien välillä vain yksi miinusmerkki eikä plus- ja miinusmerkkiä peräkkäin.

```
import math

# Luokkaa kuvaa yhta 2-ulotteista vektoria.

class Tasovektori:

    # Metodi __init__ alustaa vektorin kertoimet. Halutut
    # kertoimet annetaan metodin parametrina.

    def __init__(self, eka_kerroin, toka_kerroin):
        self.__x_kerroin = eka_kerroin
        self.__y_kerroin = toka_kerroin


    # Metodi palauttaa vektorin x-akselin suuntaisen komponentin
    # kertoimen.

    def kerro_x_kerroin(self):
        return self.__x_kerroin


    # Metodi palauttaa vektorin y-akselin suuntaisen komponentin
    # kertoimen.

    def kerro_y_kerroin(self):
        return self.__y_kerroin


    # Metodi laskee ja palauttaa vektorin pituuden. Pituus
    # lasketaan Pythagoraan lauseen avulla.

    def laske_pituus(self):
        return math.sqrt(self.__x_kerroin ** 2 + self.__y_kerroin ** 2)


    # Metodi kertoo vektorin molemmat komponentit parametrina annetulla
    # luvulla.

    def kerro_luvulla(self, kertoja):
        self.__x_kerroin *= kertoja
        self.__y_kerroin *= kertoja


    # Metodi laskee vektorin ja parametrina annetun vektori pistetulon.
    # Metodi palauttaa lasketun pistetulon.

    def pistetulo(self, toinen_vektori):
        tulo = self.__x_kerroin * toinen_vektori.__x_kerroin + \
               self.__y_kerroin * toinen_vektori.__y_kerroin
        return tulo
```

```
    # Metodi palauttaa merkkijonon, joka kuvaa vektoria.

    def __str__(self):
        if self.__y_kerroin >= 0:
            mjono = "{:.3f}i + {:.3f}j".format(\
                     self.__x_kerroin, self.__y_kerroin)
        else:
            mjono = "{:.3f}i - {:.3f}j".format(\
                     self.__x_kerroin, - self.__y_kerroin)
        return mjono
```

Alla on esimerkki pääohjelmasta, joka pyytää käyttäjältä kahden vektorin tiedot ja suorittaa näille vektoreille erilaisia toimenpiteitä. Ohjelma lukee käyttäjältä desimaalilukuja. Sitä varten on jälleen määritelty apufunktio, joka sisältää myös mahdollisten virheellisten syötteiden käsittelyn. Pääohjelmassa on oletettu, että luokka `Tasovektori` on tallennettu omaan moduuliinsa, jonka nimi on `tasovektori`.

```
import tasovektori

def lue_desimaaliluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = float(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen desimaaliluku!")
            print("Anna uusi!")
    return luku

def main():
    print("Anna ensimmaisen vektorin i-kerroin.")
    i_kerroin = lue_desimaaliluku()
    print("Anna ensimmaisen vektorin j-kerroin.")
    j_kerroin = lue_desimaaliluku()
    a_vektori = tasovektori.Tasovektori(i_kerroin, j_kerroin)
    print("Antamasi vektori on", a_vektori)

    print("Anna toisen vektorin i-kerroin.")
    i_kerroin = lue_desimaaliluku()
    print("Anna toisen vektorin j-kerroin.")
    j_kerroin = lue_desimaaliluku()
    b_vektori = tasovektori.Tasovektori(i_kerroin, j_kerroin)
    print("Antamasi vektori on", b_vektori)

    pituus1 = a_vektori.laske_pituus()
    pituus2 = b_vektori.laske_pituus()
    print("Vektorin {:s} pituus on {:.3f}".format(str(a_vektori), \
                                                  pituus1))
```

```
    print("Vektorin {:s} pituus on {:.3f}".format(str(b_vektori), \
                                                  pituus2))

    laskettu_tulo = a_vektori.pistetulo(b_vektori)
    print("Vektoreiden pistetulo on {:.3f}.".format(laskettu_tulo))

    print("Milla luvulla ensimmainen vektori kerrotaan?")
    kerroin = lue_desimaaliluku()
    a_vektori.kerro_luvulla(kerroin)
    print("Ensimmainen vektori kertomisen jalkeen", a_vektori)


main()
```

Esimerkki ohjelman suorituksesta:

```
Anna ensimmaisen vektorin i-kerroin.
4.5
Anna ensimmaisen vektorin j-kerroin.
-2.3
Antamasi vektori on 4.500i - 2.300j
Anna toisen vektorin i-kerroin.
3.3
Anna toisen vektorin j-kerroin.
5.0
Antamasi vektori on 3.300i + 5.000j
Vektorin 4.500i - 2.300j pituus on 5.054
Vektorin 3.300i + 5.000j pituus on 5.991
Vektoreiden pistetulo on 3.350.
Milla luvulla ensimmainen vektori kerrotaan?
-5.0
Ensimmainen vektori kertomisen jalkeen -22.500i + 11.500j
```

Luokan metodien sisällä voidaan kutsua saman luokan metodeita myös käsiteltävälle oliolle `self`. Esimerkiksi luokkaan `Tasovektori` voitaisiin kirjoittaa metodi `onko_pitempi`, joka tutkii, onko käsiteltävä vektori pitempi kuin parametrina saatu vektori. Vektoreiden pituuksien laskemisessa voidaan käyttää hyväksi luokassa jo määriteltyä metodia `laske_pituus`:

```
    def onko_pitempi(self, toinen_vektori):
        if self.laske_pituus() > toinen_vektori.laske_pituus():
            return True
        else:
            return False
```

Tässä siis if-käskyn ehdossa ensimmäisenä oleva `laske_pituus`-metodin kutsu laskee käsiteltävän `Tasovektori`-olion pituuden (sen, joka on annettu metodia `onko_pitempi` kutsuttaessa ennen pistettä) ja toinen kutsu parametrina annetun

`Tasovektori`-olion pituuden. Toki pituudet voitaisiin laskea suoraankin vektoreiden kenttien arvoista käyttämättä lainkaan `laske_pituus`-metodia.

Metodia `onko_pitempi` voitaisiin käyttää lisämäällä edellä esitettyyn pääohjelmaan esimerkiksi seuraavat rivit:

```
if a_vektori.onko_pitempi(b_vektori):
    print("Ensimmainen vektori on pitempi.")
else:
    print("Ensimmainen vektori ei ole pitempi kuin toinen.")
```

7.5 Kenttien yksityisyydestä

Tämän luvun esimerkeissä kenttien nimet ovat aina alkaneet kahdella alaviivalla. Tämä on sen vuoksi, että tällaisella nimeämiskäytännöllä Pythonissa estetään se, että luokan ulkopuolelta käytäisiin käsiksi suoraan kenttien arvoihin esimerkiksi seuraavaan tyyliin:

```
print(a_vektori.__x_kerroin)
a_vektori.__y_kerroin = 3.0
```

Tämä ei ole siis mahdollista, vaan kaikessa `Tasovektori`-luokan ulkopuolella olevassa koodissa kenttien arvoja on käsiteltevä luokassa määriteltyjen metodien kautta, esimerkiksi

```
print(a_vektori.kerro_x_kerroin())
```

Ensiksi voi tuntua siltä, että on turhan hankalaa käsitellä kenttiä vain luokan metodien avulla. Eikö olisi paljon helpompaa, jos kenttiä voisi käsitellä luokan ulkopuolellakin suoraan?

Kenttien suoran käytön sulkeminen luokan sisälle saa kuitenkin aikaan sen, että luokan sisäistä esitystä voidaan myöhemmin muuttaa tarvittaessa helposti ilman että luokkaa käyttäviä ohjelmia tarvitsee muuttaa.

Tarkastellaan esimerkkinä tilannetta, jossa on määritelty luokka yhden henkilön kuvaamiseen. Henkilöllä on ainakin kentät `nimi` ja `ika`. (Luokan olioilla voi olla monia muitakin kenttiä, mutta ne eivät ole oleellisia esiteltävän asian kannalta.) Luokan määrittely siis alkaa seuraavasti:

```
class Henkilo1:

    def __init__(self, nimi1):
        self.nimi = nimi1
        self.ika = 0
```

Luokkaa käytetään osana monia eri ohjelmia ohjelmassa käsiteltävien henkilöiden henkilötietojen esittämiseen. Esimerkiksi eräässä ohjelmassa voisi olla seuraavaa koodia (esitetyt käskyt ovat täysin mahdollisia, koska kenttiä `nimi` ja `ika` voi käsitellä vapaasti myös luokan ulkopuolella).

```
oppilas = Henkilo1("Matti")
oppilas.ika = 15
if oppilas.ika < 18:
    print("Oppilas on alaikainen")
else:
    print("Oppilas on taysi-ikainen")
oppilas.ika = 16
print("Oppilaan ika on", oppilas.ika)
```

Jossain vaiheessa eri ohjelmistojen ylläpitäjät tulevat siihen tulokseen, että on hankalaa esittää henkilöiden ikiä vuosina, koska ikää pitää päivittää joka vuosi. Jos ohjelmistossa pidetään yllä tietoa tuhansista henkilöistä, tarkoittaa tämä tuhansien henkilötietojen päivittämistä vuosittain.

Niinpä päätetään muuttaa luokkaa `Henkilo1` siten, että henkilöstä esitetäänkin iän sijaan henkilön syntymävuosi. Luokan uuden määrittelyn alku olisi siis seuraava:

```
class Henkilo1:

    def __init__(self, nimi1):
        self.nimi = nimi1
        self.syntymavuosi = 0
```

Tutkitaan, mitä muutoksia tämä vaatii edellä esitettyyn ohjelman osaan, jossa luodaan `Henkilo1`-olio ja käsitellään tätä. Huomataan, että kaikki kohdat, joissa viitataan henkilön `ika`-kenttään, pitää muuttaa. Muutokset eivät rajoitu pelkästään tähän ohjelmaan. Jos luokkaa on käytetty apuna monissa muissakin ohjelmissa, pitää kaikkiin näihin ohjelmiin tehdä muutoksia.

Tarkastellaan sitten vaihtoehtoista tapaa. Määritellään luokka `Henkilo2`. Tämä luokka on muuten samanlainen kuin alkuperäinen luokka `Henkilo1`, mutta olion kentät on määritelty nyt niin, että niihin ei pääse käsiksi suoraan luokan ulkopuolelta, vaan luokassa on määritelty omat metodit, joiden avulla kenttiä käsitellään. Luokan määrittelyn alku olisi seuraava:

```
class Henkilo2:

    def __init__(self, nimi1):
        self.__nimi = nimi1
        self.__ika = 0


    def kerro_ika(self):
        return self.__ika


    def muuta_ika(self, uusi_ika):
        if 0 <= uusi_ika <= 150:
            self.__ika = uusi_ika
```

Luokkaa käytettäisiin osana erilaisia ohjelmia henkilötietojen kuvaamiseen, esimerkiksi seuraavasti:

```
oppilas = Henkilo2("Matti")
oppilas.muuta_ika(15)
if oppilas.kerro_ika() < 18:
    print("Oppilas on alaikainen")
else:
    print("Oppilas on taysi-ikainen")
print("Oppilaan ika on", oppilas.kerro_ika())
```

Tutkitaan, mitä tapahtuu, jos henkilöstä päätetäänkin tallentaa iän sijasta syntymävuosi. Luokkaa `Henkilo2` ja siinä määriteltyjä metodeita pitää luonnollisesti muuttaa (nykyinen vuosi voitaisiin selvittää käytetyn vakion sijaan esimerkiksi tietokoneen kellon ajasta, mutta koska tätä ei ole kurssilla opetettu, on vuosi määritelty vakion avulla):

```
NYKYINEN_VUOSI = 2014

class Henkilo2:

    def __init__(self, nimi1):
        self.__nimi = nimi1
        self.__syntymavuosi = 0


    def kerro_ika(self):
        return NYKYINEN_VUOSI - self.__syntymavuosi


    def muuta_ika(self, uusi_ika):
        if 0 <= uusi_ika <= 150:
            self.__syntymavuosi = NYKYINEN_VUOSI - uusi_ika
```

Tarkastellaan sitten, mitä pitää muuttaa aikaisemmassa ohjelmassa, joka luo `Henkilo2`-olion ja käsittelee sitä. Havaitaan, että tähän ohjelmaan ei tarvitse tehdä lainkaan muutoksia. Vaikka luokan `Henkilo2` kenttiä on muutettu, niin luokan metodien toiminta ei ole luokan ulkopuolelta katsottuna muuttunut. Metodeita kutsutaan edelleen samalla tavalla kuin aikaisemminkin ja ne paluttavat samanlaiset arvot kuin aikaisemminkin. Tämä on suuri etu, sillä `Henkilo2`-olioita käyttäviä ohjelmia saattaa olla lukuisia. Nyt niitä ei tarvitse muuttaa mitenkään, vaan pelkkä `Henkilo2`-luokan muuttaminen riittää.

Esimerkissä näkyy myös toinenkin etu siitä, että olion kenttiä käsitellään vain luokan metodien kautta: Kentän arvoa muuttavaan metodiin on helppo lisätä tarkistus siitä, että kentälle ei yritetä antaa jotain kelvotonta arvoa, esimerkiksi negatiivista tai liian suurta ikää. Jos kentän arvoa muutetaan suoraan, pitää tämä tarkistus kirjoittaa erikseen jokaiseen ohjelman kohtaan, jossa kentän arvoa muutetaan tai sitten otetaan riski siitä, että jollakin kentällä voi olla ohjelmassa järjettömiä arvoja.

Tarkasti ottaen Pythonissa kenttien nimien yksityisyys ei ole niin tärkeää kuin edellä on esitetty, koska Pythonissa luokkaa jälkikäteen muokatessa pystyy Pythonin valmiin `property`-funktion avulla määräämään, että kentän suoran käsittelyn sijasta käytetäänkin määrättyä metodia. Monesta muusta olio-ohjelmointikielestä, esimerkiksi Javasta, tämä mahdollisuus kuitenkin puuttuu. Edellä opetettu tapa kenttien käsittelyyn on sellainen, jota voi soveltaa muillakin olio-ohjelmointikielellä ohjelmoidessa.

Vaikka kenttien nimien aloittaminen kahdella alaviivalla saakin aikaan sen, että kenttää ei voi käsitellä suoraan luokan ulkopuolelta, voi luokan sisällä olevissa metodeissa käsitellä suoraan kaikkien saman luokan olioiden kenttiä. Esimerkiksi `Tasovektori`-luokan `pistetulo`-metodin ensimmäinen versio oli kirjoitettu seuraavasti:

```
def pistetulo(self, toinen_vektori):
    tulo = self.__x_kerroin * toinen_vektori.__x_kerroin + \
           self.__y_kerroin * toinen_vektori.__y_kerroin
    return tulo
```

Tässä metodi käsittelee suoraan olion `self` kenttien lisäksi parametrina saadun olion `toinen_vektori` kenttiä. Tämä on täysin mahdollista, koska myös parametrina saatu olio on `Tasovektori`-olio. Jos sen sijaan metodin parametrina olisi esimerkiksi `Opiskelija`-olio, ei `Tasovektori`-luokassa oleva metodi voisi käsitellä suoraan sen kenttiä `__nimi`, `__opiskelijanumero`, `__harjoitusarvosana` ja `__tenttiarvosana`.

7.6 Lista olion kenttänä

Olion kenttien ei tarvitse olla pelkästään lukuja ja merkkijonoja, vaan kenttänä voi olla esimerkiksi lista tai jokin muu oliomuuttuja. Tarkastellaan seuraavaksi esimerkkiä, jossa halutaan tehdä ohjelma kaupan kanta-asiakasrekisterin hallintaan. Kirjoitetaan luokka `Bonusasiakas` yhden kanta-asiakkaan kuvaamiseen. (Varsinaista rekisteriohjelmaa ei ole esitetty tässä monisteessa.)

Luokan oliolla on kenttä asiakkaan nimeä varten sekä toinen kenttä `__ostokset`. Jälkimmäinen kenttä on lista, joka sisältää asiakkaan tarkasteltavana ajanjaksona tekemien kertaostosten arvot.

Metodissa `__init__` annetaan parametrin mukainen alkuarvo asiakkaan nimelle ja luodaan tyhjä lista, joka asetetaan `__ostokset`-kentän arvoksi.

Metodi `lisaa_ostos` lisää `__ostokset`-listaan yhden uuden ostoksen. Käytännössä se lisää listan loppuun uuden alkion, jonka arvo annetaan metodin parametrina. Parametrin pitää kuitenkin olla positiivinen. Jos parametrina annettu arvo on negatiivinen tai nolla, ostosta ei lisätä listaan lainkaan. Metodi palauttaa arvon `True` tai `False` sen mukaan, onnistuiko ostoksen lisääminen (parametri sallitulla välillä) vai ei. Tämä on malli siitä, miten tiedon metodin suorituksen onnistumisesta voi välittää ohjelman siihen kohtaan, jossa metodia kutsutaan.

Metodi `laske_keskiarvo` laskee asiakkaan tekemien ostosten keskiarvon ja palauttaa sen. Jos asiakkaalla ei ole lainkaan ostoksia (lista `__ostokset` on tyhjä) metodi palauttaa arvon 0.

Metodi `laske_rajan_ylittaneet` käy läpi listan `__ostokset` ja laskee, kuinka moni siinä olevista ostoksista ylittää parametrina annetun rajan. Kauppa voi käyttää tätä metodia silloin, kun se haluaa etsiä asiakkaita, jotka tekevät paljon suuria kertaostoksia.

Metodi `laske_bonus` laskee ja palauttaa asiakkaalle tarkasteltavan ajanjakson aikana kertyneen bonuksen.

Metodi `nollaa_ostokset` tyhjentää asiakkaan `__ostokset`-listan. Tarkasti ottaen metodi luo uuden, tyhjän listan, ja panee `__ostokset`-kentän viittaamaan siihen. Tätä metodia on tarkoitus käyttää tarkasteltavan aikajakson lopussa sen jälkeen, kun asiakkaan bonukset on laskettu ja ostotiedot halutaan nollata seuraavaa aikajaksoa varten.

Metodi `__str__` on kirjoitettu niin, että se lisää palauttamaansa merkkijonoon asiakkaan nimen ja ostokset, kunkin ostoksen omalle rivilleen. Rivinvaihdot saadaan aikaan lisäämällä merkkijonoon `"\n"`-merkkejä.

Seuraavaksi koko luokan koodi:

```
# Luokka eraan kaupan yhden kanta-asiakkaan kuvaamiseen.

class Bonusasiakas:

    # Metodi luo uuden bonusasiakkaan. Luotavan asiakkaan nimi
    # annetaan metodin parametrina.

    def __init__(self, asiakkaan_nimi):
        self.__nimi = asiakkaan_nimi
        self.__ostokset = []


    # Metodi palauttaa bonusasiakkaan nimen.

    def kerro_nimi(self):
        return self.__nimi


    # Metodi lisaa bonusasiakkalle yhden kertaostoksen. Ostoksen
    # arvo annetaan metodin parametrina. Metodi palauttaa arvon
    # True, jos ostos voidaan lisata, ja muuten arvon False.

    def lisaa_ostos(self, arvo):
        if arvo > 0.0:
            self.__ostokset.append(arvo)
            return True
        else:
            return False
```

```
    # Metodi laskee ja palauttaa bonusasiakkaan kertaostosten
    # keskiarvon.

    def laske_keskiarvo(self):
        lkm = 0
        summa = 0.0
        for ostoksen_arvo in self.__ostokset:
            summa += ostoksen_arvo
            lkm += 1
        if lkm == 0:
            return 0.0
        else:
            return summa / lkm


    # Metodi laskee, kuinka moni bonusasiakkaan kertaostoksista
    # ylittaa arvoltaan parametrina annetun rajan. Metodi palauttaa
    # naiden kertaostosten lukumaaran.

    def laske_rajan_ylittaneet(self, alaraja):
        ylittaneiden_lkm = 0
        for arvo in self.__ostokset:
            if arvo > alaraja:
                ylittaneiden_lkm += 1
        return ylittaneiden_lkm


    # Metodi laskee ja palauttaa asiakkaalle tulevan
    # bonuksen, joka maaraytyy ostosten kokonaisarvon perusteella.

    def laske_bonus(self):
        PIENIBONUS = 1.0
        SUURIBONUS = 2.5
        BONUSRAJA = 400.0
        summa = 0.0
        for ostos in self.__ostokset:
            summa += ostos
        if summa >= BONUSRAJA:
            bonus = SUURIBONUS * summa / 100.0
        else:
            bonus = PIENIBONUS * summa / 100.0
        return bonus


    # Metodi nollaa bonusasiakkaan ostostiedot.

    def nollaa_ostokset(self):
        self.__ostokset = []
```

```
    # Metodi palauttaa merkkijonon, joka sisaltaa asiakkaan
    # nimen ja ostokset, kunkin kertaostoksen arvon omalla rivillaan.

    def __str__(self):
        mjono = "Asiakas: " + self.__nimi + "\nOstot:\n"
        for arvo in self.__ostokset:
            mjono += "{:.2f} eur\n".format(arvo)
        return mjono
```

Seuraavaksi esimerkki pääohjelmasta, joka luo kaksi `Bonusasiakas`-oliota ja suorittaa näille erilaisia toimenpiteitä. Ohjelmassa on jälleen laadittu apufunktio desimaaliluvun lukemiseen käyttäjältä. Ohjelma kannattaisi jakaa useampaan funktioon, mutta se on tässä jätetty tekemättä, jotta olioiden luonti ja niiden metodien kutsuminen olisi mahdollisimman selkeätä myös aloittelijalle.

Esimerkistä nähdään myös se, miten metodin `lisaa_ostos` paluuarvoa voidaan käyttää hyväksi. Koska metodi palauttaa arvon `True` tai `False`, voidaan sen kutsu kirjoittaa suoraan if-käskyn ehdoksi. Tämä saa aikaan sen, että metodi suoritetaan ja toisaalta paluuarvon totuusarvoa voidaan käyttää saman tien ratkaisemassa se, suoritetaanko if-käskyn sisällä oleva käsky.

```
import bonusasiakas

def lue_desimaaliluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = float(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen desimaaliluku!")
            print("Anna uusi!")
    return luku


def main():
    OSTOSKERRAT = 4
    nimi1 = input("Anna 1. asiakkaan nimi: ")
    asiakas1 = bonusasiakas.Bonusasiakas(nimi1)
    nimi2 = input("Anna 2. asiakkaan nimi: ")
    asiakas2 = bonusasiakas.Bonusasiakas(nimi2)
    for i in range(OSTOSKERRAT):
        print("Anna asiakkaan {:s} yhden kertaoston arvo.".\
               format(asiakas1.kerro_nimi()))
        luettu_arvo = lue_desimaaliluku()
        if asiakas1.lisaa_ostos(luettu_arvo):
            print("Ostoksen lisays onnistui.")
        else:
            print("Ostoksen lisays ei onnistunut.")
```

```
    for i in range(OSTOSKERRAT):
        print("Anna asiakkaan {:s} yhden kertaoston arvo.".\
               format(asiakas2.kerro_nimi()))
        luettu_arvo = lue_desimaaliluku()
        if asiakas2.lisaa_ostos(luettu_arvo):
            print("Ostoksen lisays onnistui.")
        else:
            print("Ostoksen lisays ei onnistunut.")

    print("1. asiakkaan ostosten keskiarvo: {:.2f} eur".\
           format(asiakas1.laske_keskiarvo()))
    print("2. asiakkaan ostosten keskiarvo: {:.2f} eur".\
           format(asiakas2.laske_keskiarvo()))

    print("1. asiakkaan bonus: {:.2f} eur".format(\
           asiakas1.laske_bonus()))
    print("2. asiakkaan bonus: {:.2f} eur".format(\
           asiakas2.laske_bonus()))

    print("Anna raja, jonka ylittavat ostokset haetaan.")
    raja = lue_desimaaliluku()
    ylitykset1 = asiakas1.laske_rajan_ylittaneet(raja)
    ylitykset2 = asiakas2.laske_rajan_ylittaneet(raja)
    print("1. asiakkaalla oli {:d} suurempaa ostosta".\
           format(ylitykset1))
    print("2. asiakkaalla oli {:d} suurempaa ostosta".\
           format(ylitykset2))

    print("Asiakkaiden tiedot:")
    print(asiakas1)
    print(asiakas2)

    asiakas1.nollaa_ostokset()
    print("1. asiakas nollauksen jalkeen:")
    print(asiakas1)


main()
```

Alla esimerkki ohjelman suorituksesta:

```
Anna 1. asiakkaan nimi: Anna Lahti
Anna 2. asiakkaan nimi: Roope Rikas
Anna asiakkaan Anna Lahti yhden kertaoston arvo.
45.0
Ostoksen lisays onnistui.
Anna asiakkaan Anna Lahti yhden kertaoston arvo.
-12.8
Ostoksen lisays ei onnistunut.
Anna asiakkaan Anna Lahti yhden kertaoston arvo.
```

```
0.0
Ostoksen lisays ei onnistunut.
Anna asiakkaan Anna Lahti yhden kertaoston arvo.
15.0
Ostoksen lisays onnistui.
Anna asiakkaan Roope Rikas yhden kertaoston arvo.
120.0
Ostoksen lisays onnistui.
Anna asiakkaan Roope Rikas yhden kertaoston arvo.
10.30
Ostoksen lisays onnistui.
Anna asiakkaan Roope Rikas yhden kertaoston arvo.
120.4
Ostoksen lisays onnistui.
Anna asiakkaan Roope Rikas yhden kertaoston arvo.
180.0
Ostoksen lisays onnistui.
1. asiakkaan ostosten keskiarvo: 30.00 eur
2. asiakkaan ostosten keskiarvo: 107.68 eur
1. asiakkaan bonus: 0.60 eur
2. asiakkaan bonus: 10.77 eur
Anna raja, jonka ylittavat ostokset haetaan.
44.0
1. asiakkaalla oli 1 suurempaa ostosta
2. asiakkaalla oli 3 suurempaa ostosta
Asiakkaiden tiedot:
Asiakas: Anna Lahti
Ostot:
45.00 eur
15.00 eur

Asiakas: Roope Rikas
Ostot:
120.00 eur
10.30 eur
120.40 eur
180.00 eur

1. asiakas nollauksen jalkeen:
Asiakas: Anna Lahti
Ostot:
```

7.7 Listan alkiona olioita

Palataan luvun alun esimerkkiin, jossa haluttiin käsitellä ohjelmointikurssin opiskelijatietoja. Opiskelijoita on sen verran monta, että ei ole järkevää määritellä omaa muuttujaa jokaista luotavaa `Opiskelija`-oliota varten. Sen sijaan luodut `Opiskelija`-oliot (tai täsmällisesti ottaen viitteet luotuihin olioihin) voidaan tal-

lentaa listaan. Kun lista on luotu, sen loppuun voidaan lisätä uusi `Opiskelija`-olio esimerkiksi seuraavasti:

```
opiskelijalista.append(Opiskelija("Matti Virtanen", "11111F"))
```

Tässä on siis sekä luotu uusi olio että lisätty se heti listan loppuun. Toinen vaihtoehto on luoda ensin olio, panna jokin muuttuja viittaamaan siihen ja lisätä sitten olio listaan tämän muuttujan avulla:

```
uusi_opiskelija = Opiskelija("Matti Virtanen", "11111F")
opiskelijalista.append(uusi_opiskelija)
```

Listaa voidaan indeksoida samalla tavalla kuin lukuja sisältävää listaa. Jos lista jo sisältää vähintään `i + 1` alkiota, voidaan sen indeksille `i` sijoittaa viittaus juuri luotuun `Opiskelija`-olioon esimerkiksi seuraavasti (listassa indeksillä aikaisemmin ollut arvo häviää):

```
uusi_opiskelija = Opiskelija("Matti Virtanen", "11111F")
opiskelijalista[i] = uusi_opiskelija
```

Listassa indeksillä `i` olevalle `Opiskelija`-oliolle voidaan kutsua `Opiskelija`-luokan metodeita seuraavasti:

```
print("Opiskelijan", opiskelijalista[i].kerro_nimi())
print("kurssiarvosana on", opiskelijalista[i].laske_kokonaisarvosana())
```

On myös mahdollista käydä for-käskyssä läpi kaikki listan oliot ja kutsua heille jokaiselle vuorotellen luokan metodeita seuraavaan tapaan:

```
for kurssilainen in opiskelijalista:
    print("Opiskelijan", kurssilainen.kerro_nimi())
    print("kurssiarvosana on", kurssilainen.laske_kokonaisarvosana())
```

Katsotaan sitten esimerkkiohjelmaa, joka ensin lukee käyttäjän antamat opiskelijoiden nimet ja opiskelijanumerot, pyytää sen jälkeen käyttäjältä samojen opiskelijoiden tentti- ja harjoitusarvosanat ja lopuksi tulostaa tuloslistan, jossa on myös opiskelijoiden kokonaisarvosanat.

Tämä ohjelma on jaettu useampaan funktioon, ja tieto `Opiskelija`-oliot sisältävästä listasta välitetään funktioiden välillä funktioiden parametrien ja paluuarvojen avulla.

Funktio `lue_kokonaisluku` on jälleen apufunktio, joka lukee käyttäjältä yhden kokonaisluvun ja käsittelee mahdolliset virheelliset syötteet.

Funktio `lue_opiskelijatiedot` luo ensin tyhjän listan opiskelijoita varten. Sitten se lukee opiskelijoiden nimet ja opiskelijanumerot käyttäjältä niin, että yhden opiskelijan nimi ja opiskelijanumero on annettu yhdellä rivillä. Tästä rivistä erotetaan `split`-metodin avulla erikseen nimi ja opiskelijanumero. Näiden perusteella luodaan

uusi `Opiskelija`-olio ja lisätään se listan loppuun. Kun kaikkien opiskelijoiden tiedot on luettu, funktio palauttaa arvonaan opiskelijat sisältävän listan.

Funktio `lisaa_arvosanatiedot` käy parametrina annetun `Opiskelija`-olioita sisältävän listan läpi olio kerrallaan. Se pyytää käyttäjältä kunkin opiskelijan tentti- ja harjoitustehtäväarvosanan ja antaa ne käsiteltävälle oliolle `Opiskelija`-luokan metodien avulla.

Funktio `tulosta_tulokset` käy jälleen parametrina annetun `Opiskelija`-olioita sisältävän listan läpi. Se tulostaa jokaisesta opiskelijasta opiskelijanumeron, nimen sekä tentti-, harjoitus- ja kokonaisarvosanan, jotka selvitetään `Opiskelija`-luokan metodien avulla. Listan läpi käyvä for-käsky on nyt kirjoitettu vähän eri muotoon kuin edellisessä funktiossa. Tämä ei ole mitenkään välttämätöntä, vaan for-käsky olisi voitu kirjoittaa myös samalla tavalla kuin funktiossa `lue_arvosanatiedot`. Esimerkissä on vain haluttu näyttää, miten listaa voi käydä läpi myös indeksoinnin avulla.

Pääohjelma kutsuu määriteltyjä funktioita niin, että halutut toimenpiteet (opiskelijoiden henkilötietojen kysyminen, arvosanatietojen kysyminen ja tulosten tulostaminen) tehdään halutussa järjestyksessä.

```
import opiskelija

def lue_kokonaisluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = int(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen kokonaisluku!")
            print("Anna uusi!")
    return luku


# Funktio lukee kayttajien antamien opiskelijoiden nimet ja
# opiskelijanumerot. Se luo vastaavat Opiskelija-oliot ja
# lisaa ne listaan. Funktio palauttaa taman listan.

def lue_opiskelijatiedot():
    opiskelijat = []
    print("Anna opiskelijoiden nimet ja opiskelijanumerot")
    print("samalla rivilla kauttaviivalla erotettuna.")
    print("Lopeta tyhjalla rivilla.")
    rivi = input()
    while rivi != "":
        tiedot = rivi.split("/")
        if len(tiedot) != 2:
            print("Virheellinen rivi!")
```

```
        else:
            uusi = opiskelija.Opiskelija(tiedot[0], tiedot[1])
            opiskelijat.append(uusi)
        rivi = input()
    print("Opiskelijoiden tiedot luettu!")
    return opiskelijat


# Funktio pyytaa kayttajalta parametrina saadussa listassa
# olevien opiskelijoiden tentti- ja harjoitusarvosanat ja
# lisaa ne listan Opiskelija-olioille.

def lisaa_arvosanatiedot(opiskelijalista):
    for kurssilainen in opiskelijalista:
        print("Anna opiskelijan {:s} tenttiarvosana:".format(\
               kurssilainen.kerro_nimi()))
        tentti_as = lue_kokonaisluku()
        print("harjoitusarvosana:")
        harjoitus_as = lue_kokonaisluku()
        kurssilainen.muuta_tenttiarvosana(tentti_as)
        kurssilainen.muuta_harjoitusarvosana(harjoitus_as)
    print("Arvosanat lisatty!")


# Funktio tulostaa parametrina annetussa listassa
# olevien Opiskelija-olioiden kurssitulokset.

def tulosta_tulokset(opiskelijat):
    print("numero nimi            tentti harj   kurssi")
    for i in range(len(opiskelijat)):
        print("{:6s} {:15s} {:<6d} {:<6d} {:<6d}".format(\
               opiskelijat[i].kerro_opiskelijanumero(), \
               opiskelijat[i].kerro_nimi(), \
               opiskelijat[i].kerro_tenttiarvosana(), \
               opiskelijat[i].kerro_harjoitusarvosana(), \
               opiskelijat[i].laske_kokonaisarvosana()))


def main():
    opiskelijatiedot = lue_opiskelijatiedot()
    lisaa_arvosanatiedot(opiskelijatiedot)
    tulosta_tulokset(opiskelijatiedot)
    print("Ohjelma paattyy.")


main()
```

Alla esimerkki ohjelman suorituksesta:

```
Anna opiskelijoiden nimet ja opiskelijanumerot
samalla rivilla kauttaviivalla erotettuna.
Lopeta tyhjalla rivilla.
Virta Nina/66445G
Heikkila Heikki/77553V
Lahti Mika/56789T
Lahti Maija/87642R

Opiskelijoiden tiedot luettu!
Anna opiskelijan Virta Nina tenttiarvosana:
4
harjoitusarvosana:
5
Anna opiskelijan Heikkila Heikki tenttiarvosana:
5
harjoitusarvosana:
5
Anna opiskelijan Lahti Mika tenttiarvosana:
1
harjoitusarvosana:
3
Anna opiskelijan Lahti Maija tenttiarvosana:
5
harjoitusarvosana:
0
Arvosanat lisatty!
numero nimi            tentti harj   kurssi
66445G Virta Nina      4      5      5
77553V Heikkila Heikki 5      5      5
56789T Lahti Mika      1      3      2
87642R Lahti Maija     5      0      0
Ohjelma paattyy.
```

7.8 Tiivistelmä luokan määrittelystä ja olioiden luonnista ja käytöstä

Tähän kappaleeseen on koottu vielä pääkohtia tärkeimmistä luvussa esitetyistä asioista, ei kuitenkaan läheskään kaikista.

Luokan määrittely

Luokan määrittely aloitetaan luokan otsikolla, esimerkiksi

```
class Opiskelija:
```

Luokan sisälle kirjoitettavassa metodissa `__init__` määritellään, mitä tapahtuu, kun luodaan uusi luokan olio

```
    def __init__(self, annettu_nimi, numero):
        self.__nimi = annettu_nimi
        self.__opiskelijanumero = numero
        self.__tenttiarvosana = 0
        self.__harjoitusarvosana = 0
```

`self` tarkoittaa sitä oliota, jota ollaan juuri luomassa.

Luokkaan kirjoitettavat metodit määrittelevät, mitä toimenpiteitä luokan olioille voidaan tehdä, esimerkiksi

```
    def laske_kokonaisarvosana(self):
        if self.__tenttiarvosana == 0 or self.__harjoitusarvosana == 0:
            arvosana = 0
        else:
            arvosana = (self.__tenttiarvosana +
                        self.__harjoitusarvosana + 1) // 2
        return arvosana
```

`self` tarkoittaa tällöin sitä oliota, jolle metodia kutsutaan (olioon viittava muuttuja on kutsussa ennen pistettä ja metodin nimeä).

Yleensä luokkaan kirjoitetaan metodi `__str__`. Se palauttaa merkkijonon, joka sisältää tietoja oliosta

```
    def __str__(self):
        mjono = self.__nimi + ", " + self.__opiskelijanumero + \
                ", tenttiarvosana: " + str(self.__tenttiarvosana) + \
                ", harjoitusarvosana: " + str(self.__harjoitusarvosana)
        return mjono
```

Olioiden luominen ja käyttö

Jos olioita luodaan ohjelmassa, joka on toisessa moduulissa (tiedostossa) kuin luokan määrittely, pitää luokan sisältävä moduuli ensin tuoda ohjelman käyttöön

```
import opiskelija
```

Esimerkki olion luomisesta (luokka eri moduulissa):

```
kurssilainen1 = opiskelija.Opiskelija(nimi1, op_nro1)
```

Olion luonti on esitetty =-merkin oikealla puolella. Sijoituskäskyn avulla muuttuja `kurssilainen1` pannaan viittaamaan luotuun olioon.

Esimerkki metodin kutsumisesta oliolle:

```
print("Kurssiarvosana:", kurssilainen1.laske_kokonaisarvosana())
```

Jos metodilla on parametrin `self` lisäksi muita parametreja, annetaan ne metodin kutsussa samalla tavalla kuin funktioiden kutsussa.

```
kurssilainen1.muuta_tenttiarvosana(4)
```

Jos olion luokassa on määritelty metodi `__str__`, voidaan olion tiedot tulostaa helposti.

```
print(kurssilainen1)
```