LUT Python ohjelmointiopas 2023 - osa 6
Sisällysluettelo

Luku 6: Ohjelmien lukemat ja kirjoittamat tekstitiedostot

Olemme tähän asti käynnistäneet ohjelman ja syöttäneet sille tietoa, jonka jälkeen ohjelma on tulostanut tulosteita ja päättynyt. Ohjelman joustavan käytön kannalta olisi helpompaa, jos ohjelma pystyisi lukemaan tietoja tiedostosta. Käytännössä ohjelmalle voi syöttää muutamia tai ehkä kymmeniä tietoja näppäimistön avulla, mutta tuhansien tai miljoonien tietojen syöttäminen yhdellä kerralla ei ole järkevää eikä käytännössä mahdollista. Koska ohjelman perusominaisuuksiin kuuluu toistorakenne, tällaisten tietomäärien lukeminen tiedostosta on luonteva ja tehokas toimintatapa. Tutustutaan seuraavaksi tarkemmin tekstitiedostojen käyttöön tietokoneohjelmissa, joka mahdollistaa tietojen lukemisen tiedostoista ja tulosten kirjoittamisen tiedostoihin myöhempää käyttöä varten.

Tässä luvussa keskitymme tekstitiedostojen käsittelyyn. Tekstitiedostot ovat käytännössä pitkiä merkkijonoja, joten lähtökohtaisesti niiden käsittely on meille jo tuttua. Koska tiedostot voivat olla hyvinkin isoja, tutustumme uusiin tapoihin muodostaa ja käsitellä merkkijonoja eli tietoalkioiden tunnistamiseen. Keskitymme tässä vaiheessa itse tehtyihin tiedostoihin ja sellaisiin tiedostoihin, joita voi ja kannattaa muokata itse. Käytännössä esim. tietokoneen käyttöön liittyviä (systeemi) tiedostoja tai muiden ohjelmien tekemiä (binaari) tiedostoja ei kannata muokata, ellei tiedä mitä tekee. Käsiteltäviä tekstitiedostoja voi avata ja tutkia tekstieditorilla eli käytännössä samalla editorilla, jolla teet ja muokkaat ohjelmia. Tekstinkäsittely- ja taulukkolaskentaohjelmat poistavat usein tiettyjä merkkejä tai muuttavat niiden esitysmuotoa, joten niiden käyttö datatiedostojen katselussa on usein harhaanjohtavaa ja jos tiedoston tallentaa katselun jälkeen, tahtoo tiedostomuoto muuttua eikä se vastaa alkuperäistä. Siksi ohjelmoidessa kannattaa käyttää vain ohjelmointiin tehtyjä editoreita sekä ohjelmien että käsiteltävien tiedostojen katseluun ja muokkaamiseen.

Perusoperaatiot

Tutustutaan ensin tiedostonkäsittelyn perusoperaatioihin eli tiedostojen kirjoittamiseen ja lukemiseen. Käytännössä tiedostoissa on aina useita rivejä ja usein joudumme käsittelemään rinnakkain useita tiedostoja, joten käydään ne läpi saman tien. Tiedostoja käsitellessä lähtökohta on, että Python-ohjelmat käsittelevät samassa kansiossa lähdekoodin kanssa olevia tiedostoja. Kuten oppaan alussa oli puhetta, kannattaa kaikki ohjelmat tehdä omaan itse tehtyyn kansioon, jotta sen luku- ja kirjoitusoikeuden ovat kunnossa. Käyttöjärjestelmät rajoittavat usein tietyissä kansioissa olevien tiedostojen käyttöoikeuksia, joten näiden ongelmien välttämiseksi kannattaa luoda itse kansio, johon kaikki tähän oppaaseen liittyvät tiedostot laitetaan.

Tiedoston kirjoitus ja luku

Esimerkissä 6.1 näkyy tiedoston kirjoittava ja lukeva aliohjelma sekä niitä kutsuva pääohjelma. Yksi aliohjelma hoitaa aina yhden asian eli kirjoittava aliohjelma avaa tiedoston, kirjoittaa sinne merkkijonoja, sulkee tiedoston ja kertoo käyttäjälle operaation suorittamisesta sekä palaa kutsuvaan ohjelmaan. Vastaavasti lukeva aliohjelma avaa tiedoston, lukee tiedostossa olevat rivit ja tulostaa ne, sulkee tiedoston ja kertoo käyttäjälle operaation suorittamisesta sekä palaa kutsuvaan ohjelmaan. Pääohjelmassa käytettävä tiedostonimi tallennetaan muuttujaan, kutsutaan aliohjelmia, pidetään käyttäjä tietoisena ohjelman etenemisestä ja lopetetaan ohjelma.

Tiedosto-operaatiot kohdistetaan tiedostonimen perusteella halutun ohjelman kanssa samassa kansiossa olevaan tiedostoon. Siksi tiedoston nimi sijoitetaan pääohjelmassa muuttujaan, jotta se voidaan välittää eri aliohjelmiin ensimmäisenä parametrina ja kaikki aliohjelmat käsittelevät varmasti samaa tiedostoa. Jokainen aliohjelma avaa halutun

tiedoston tarvitsemassaan tilassa eli kirjoittava aliohjelma avaan tiedoston ”w” eli write-moodissa kirjoittamista varten, kun taas lukeva aliohjelma käyttää lukutilaa ts. ”r” eli read-moodi. Jokainen aliohjelma sulkee tiedoston aliohjelman lopuksi. Käytännössä tiedostonkäsittely tehdään kokonaisuutena yhdessä ohjelmassa eli aliohjelman alussa tiedosto avataan, sitten sitä käytetään ja aliohjelman lopuksi tiedosto suljetaan. Näin koodista on helppo nähdä, että avattu tiedosto tulee suljettua.

Varsinainen tiedostonkäsittely kirjoittavassa aliohjelmassa tarkoittaa merkkijonojen kirjoittamista tiedostoon. Merkkijonot tulevat aliohjelmaan nyt parametreina ja kirjoitusvaiheessa jokaiseen kirjoitettavaan merkkijonoon lisätään rivinvaihtomerkki. Näin lukuvaiheessa tiedot on helppo lukea rivi kerrallaan ja käsitellä dataan kuulumaton rivinvaihtomerkki sopivalla tavalla. Tässä esimerkissä tiedot vain tulostetaan, jolloin riittää huolehtia tulostukseen tulevien rivinvaihtomerkkien sopivasta määrästä.

Esimerkki 6.1. Tiedoston kirjoitus ja luku

```
def kirjoitaTiedosto(Nimi, Rivi1, Rivi2, Rivi3):
    Tiedosto = open(Nimi, "w")
    Tiedosto.write(Rivi1 + "\n")
    Tiedosto.write(Rivi2 + "\n")
    Tiedosto.write(Rivi3 + "\n")
    Tiedosto.close()
    print("Tiedosto '"+Nimi+"' kirjoitettu.")
    return None

def lueTiedosto(Nimi):
    Tiedosto = open(Nimi, "r")
    Rivi = Tiedosto.readline()
    print(Rivi, end="")
    Rivi = Tiedosto.readline()
    print(Rivi, end="")
    Rivi = Tiedosto.readline()
    print(Rivi, end="")
    Tiedosto.close()
    print("Tiedosto '"+Nimi+"' luettu ja tulostettu.")
    return None

def paaohjelma():
    TiedostoNimi = "L06E1.txt"
    kirjoitaTiedosto(TiedostoNimi, "Ville", "12345", "3.14159")
    lueTiedosto(TiedostoNimi)
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Tiedosto 'L06E1.txt' kirjoitettu.
Ville
12345
3.14159
Tiedosto 'L06E1.txt' luettu ja tulostettu.
Kiitos ohjelman käytöstä.
```

Tiedoston avaus tapahtuu käskyllä `Tiedosto = open(Nimi, "w")`, jossa `open`-käsky avaa `L06E1.txt` nimisen tekstitiedoston kirjoittamista varten. Käskyn ensimmäinen parametri on avattavan tiedoston nimi ja toinen on sen tila ja avauskäskyn paluuarvo otetaan

talteen `Tiedosto`-muuttujaa eli tiedostokahvaan, jolla avattua tiedostoa voidaan käyttää jatkossa. Avattavan tiedoston nimen on oltava merkkijono (tai merkkijonon sisältävä muuttuja) ja samoin avaustila on merkkijono. Avaustila eli moodi voi olla kirjoittaminen eli 'w' (write), lukeminen eli 'r' (read) tai lisäys eli 'a' (append), jotka mahdollistavat vastaavien operaatioiden tekemisen tiedostolle. Huomaa, että w-moodi luo uuden tiedoston ja mahdollisesti aiempi samanniminen tiedosto tietoineen häviää samalla hetkellä. Siksi vaihtoehto kirjoittamiselle on lisäys, jolloin olemassa oleva tiedosto säilyy ja kirjoitusoperaatiot lisäävät tiedoston loppuun uusia merkkejä/rivejä.

Tiedostoon kirjoittaminen tapahtuu käskyllä `Tiedosto.write()`, joka saa parametrina kirjoitettavan merkkijonon. Tässä kannattaa huomioida seuraavat asiat:

1. Tiedostokahva `Tiedosto` ei päästä meitä käsiksi tiedoston sisältöön vaan se tarjoaa "kulkuyhteyden" tiedoston sisälle. Tiedostoa käytetään kahvan kautta ja jäsenfunktiot kuten `write()` tai `close()` merkataan pistenotaatiolla. Käsittelemme jäsenfunktiot kohta tarkemmin, joten tässä vaiheessa käytä muuttujanimeä ja jäsenfunktionimeä sekä lisää piste niiden väliin.
2. Tiedoston avaaminen kirjoitustilassa tarkoittaa, että tiedostoon voi kirjoittaa. Jos tila olisi valittu toisin, esim. lukutila, kirjoitusyritys aiheuttaisi virheilmoituksen.
3. Tekstitiedostoon voi kirjoittaa vain merkkijonoja. Tämä tulee ottaa kirjaimellisesti; jos haluat tallentaa lukuarvoja tiedostoihin, pitää ne ennen kirjoittamista muuttaa merkkijonoiksi, esim. käyttäen funktioita `str()` ja `round()`.
4. `write()` -käsky poikkeaa `print()` –käskystä kahdella tavalla: se ottaa parametrina vain yhden merkkijonon eikä se lisää mitään merkkijonoon. `write()` ei siis lisää merkkijonon loppuun rivinvaihtomerkkiä niin kuin `print()` tekee.

Esimerkissä tiedosto avataan uudelleen kirjoittamista varten lukutilassa eli `'r'` (`read`). Lukutila on virheherkempi kuin kirjoitustila ja se palauttaa tulkille virheen, mikäli kohdetiedostoa ei ole olemassa. Tämä on loogista, koska jos tiedostoa ei ole olemassa, ei sieltä varmaan pysty lukemaankaan mitään. Koska loimme juuri tiedoston, voimme olla varmoja sen olemassaolosta ja ohjelma jatkaa itse lukuvaiheeseen. Tiedoston luetaan yksi rivi kerrallaan käskyllä `Tiedosto.readline()`, joka palauttaa tiedostosta merkkijonon, joka edustaa yhtä tiedoston fyysistä riviä. Tässä tapauksessa rivi alkaa tiedoston kirjanmerkin kohdalta (eli arvosta, joka kertoo missä kohdin tiedostoa olemme menossa) ja päättyy joko rivinvaihtomerkkiin tai tiedoston loppuun. Jos kirjanmerkki on tiedoston lopussa, palauttaa jäsenfunktio tyhjän merkkijonon, jonka pituus on 0, ja tällöin tiedämme, että voimme lopettaa lukemisen ja poistua toistorakenteesta. Mikäli `readline()`-palautti rivin, tulostamme sen `print()`-käskyllä, mutta lisäämättä `print()`:n rivinvaihtomerkkiä tiedostosta luetun rivinvaihtomerkin perään. Lopuksi suljemme tiedoston käytön päätyttyä.

Tiedoston sulkeminen tapahtuu `close()`-funktiolla. Tiedoston sulkeminen estää tiedostosta lukemisen ja siihen kirjoittamisen, kunnes tiedosto avataan uudelleen. Lisäksi se vapauttaa tiedostokahvan sekä ilmoittaa käyttöjärjestelmälle, että tiedostonkäsittelyn varaama muisti voidaan vapauttaa. Muista aina sulkea käyttämäsi tiedostot! Vaikka Pythonissa onkin automaattinen muistinhallinta ja varsin toimiva automatiikka estämään ongelmien muodostumisen, on silti tärkeää, että kaikki ohjelmat siivoavat omat sotkunsa ja vapauttavat käyttämänsä käyttöjärjestelmän resurssit.

Tiedoston kirjoittamisen onnistuminen kannattaa tarkistaa etsimällä kirjoitettu tiedosto tietokoneelta, normaalisti samasta hakemistosta ohjelmatiedoston kanssa, ja avaamalla se koodieditorilla (esim. IDLE). Tiedostossa pitäisi näkyä ohjelman alussa olleet kolme merkkijonoa, jotka ohjelma myös tulosti juuri näytölle. Tekstitiedostossa pitää olla tarkkana rivinvaihtomerkkien kanssa, sillä ne ovat erittäin tärkeässä roolissa tiedostoja luettaessa.

Monirivisen tiedoston käsittely

Monirivisen tiedoston käsittely perustuu esimerkissä 6.1 nähtyihin käskyihin ja ideoihin eli tiedosto avaan, tiedot kirjoitetaan/luetaan, tiedosto suljetaan saman aliohjelman sisällä ja käyttäjälle kerrotaan ohjelman etenemisestä tulosteilla. Esimerkin 6.2 mukaisesti kirjoitettaessa tiedostoon useita alkioita lähtökohta on tyypillisesti toistorakenne, jolla kirjoitetaan tiedostoon aina yksi valmis rivi kerrallaan. Monirivisen tiedoston lukemiseen on useita vaihtoehtoja, mutta tässä oppaassa suositellaan käytettäväksi aina samaa rakennetta eli luetaan tiedostosta yksi rivi kerrallaan `readline()`-jäsenfunktiolla, jonka jälkeen riviä voidaan käsitellä sopivalla tavalla eli tyypillisesti poistaa rivin lopussa oleva rivinvaihtomerkki ja mahdollisesti pilkkoa merkkijono tietoalkioihin tms. Tyypillisesti luettavan tiedoston rivimäärä ei ole tiedossa, joten alkuehtoinen `while`-rakenne on tähän sopiva rakenne. Loppuehtoinen toistorakenne edellyttää ensimmäisen rivin lukemista ennen lopetusehdon tarkastamista ja toista rivin luku -käskyä toistorakenteen sisällä.

Tekstitiedoston keskeinen yksikkö on rivi. Siksi kirjoitettaessa kannattaa aina toimia samalla tavalla eli kirjoittaa rivin sisältö ja sen perään rivinvaihtomerkki. Tällä periaatteella viimeiselle riville tulee aina tyhjä rivi, jota voidaan lukiessa käyttää tiedoston lukemisen loppu-merkkinä.

Esimerkki 6.2. Monirivisen tiedoston käsittely

```
def kirjoitaLuvut(Nimi, Lukuja):
    Tiedosto = open(Nimi, "w")
    for i in range(Lukuja):
        Tiedosto.write(str(i) + "\n")
    Tiedosto.close()
    print("Tiedosto '"+Nimi+"' kirjoitettu.")
    return None

def lueLuvut(Nimi):
    Tiedosto = open(Nimi, "r")
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        Luku = int(Rivi)
        print(Luku, end=" ")
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    print()
    print("Tiedosto '"+Nimi+"' luettu ja tulostettu.")
    return None

def paaohjelma():
    TiedostoNimi = "L06E2.txt"
    Lukumaara = int(input("Montako lukua tiedostoon kirjoitetaan: "))
    kirjoitaLuvut(TiedostoNimi, Lukumaara)
    lueLuvut(TiedostoNimi)
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Montako lukua tiedostoon kirjoitetaan: 5
Tiedosto 'L06E2.txt' kirjoitettu.
0 1 2 3 4
Tiedosto 'L06E2.txt' luettu ja tulostettu.
Kiitos ohjelman käytöstä.
```

Monen tiedoston käyttö rinnakkain

Usein tiedostoja käsiteltäessä tulee tarve lukea yhdestä tiedostosta ja kirjoittaa toiseen samanaikaisesti eli käytännössä lukea ja kirjoittaa useita tiedostoja vuorotellen. Tämä ei ole ongelma Pythonille, sillä avonaisten tiedostojen lukumäärää ei ole rajoitettu, vaan tiedostoja voi ottaa käyttöön tarvittavan määrän.

Esimerkissä 6.3 luetaan yhdestä tiedostosta riveillä olevia nimiä ja kirjoitetaan alle 6 merkkiä sisältävät rivit toiseen tiedostoon. Aluksi otetaan käyttöön kaksi tiedostokahvaa, joista yhdessä on luettava tiedosto ja toisessa kirjoitettava tiedosto. Tämän jälkeen käydään luettava tiedosto läpi esimerkin 6.2 mukaisesti ja jokaisen rivin kohdalla tarkastetaan sen pituus sekä kirjoitetaan alle 6 merkkiä pitkät rivit toiseen tiedostoon. Kun rivinpituus on 0 eli tiedostosta on luettu viimeisenä oleva tyhjä rivi, suljetaan tiedostot, kerrotaan käyttäjälle mitä on tehty ja lopetetaan ohjelma. Tässä esimerkissä kannattaa tarkistaa, minkä mittaiset nimet on kirjoitettu uuteen tiedostoon.

Esimerkki 6.3. Monen tiedoston käyttö rinnakkain

```
def paaohjelma():
    TiedostoLue = open("L06E3D1.txt", "r")
    TiedostoKirjoita = open("L06E3T1.txt", "w")
    Rivi = TiedostoLue.readline()
    while (len(Rivi) > 0):
        if (len(Rivi) < 6):
            TiedostoKirjoita.write(Rivi)
        Rivi = TiedostoLue.readline()
    TiedostoLue.close()
    TiedostoKirjoita.close()
    print("Lyhyet rivit kirjoitettu toiseen tiedostoon. Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Lyhyet rivit kirjoitettu toiseen tiedostoon. Kiitos ohjelman käytöstä.
```

Käsiteltävät tiedostot

L06E3D1.txt

```
Kalle
Mia
Johanna
Eero
Joonatan
Anu
```

L06E3T1.txt

```
Mia
Eero
Anu
```

Tiedostonkäsittely laajemmin

Tiedoston avaaminen

Tiedoston avaamisvaihe määrittelee useita tiedoston käytön kannalta keskeisiä asioita. Kuten aiemmin oli puhetta, tiedosto voidaan avata eri moodeissa eli lukemista, kirjoittamista tai lisäämistä varten, ts. ”r”, ”w” tai ”a” parametrilla. Selkeyden vuoksi nämä tilat mahdollistavat vain valitun operaation tekemisen ja jos tarve muuttuu, tulee tiedosto sulkea ja avata uudestaan toisessa moodissa.

Tietoa kirjoitettaessa tulee usein tarve poistaa vanhat tiedot eli tyhjentää tiedosto. Siksi kirjoitus-tila luo uuden tiedoston, jos sellaista ei ole, ja jos tiedosto on jo olemassa, tyhjentää se tiedoston. Käytännössä tiedosto voidaan siis tyhjentää avaamalla se kirjoitus-tilassa ja sulkemalla se, kuten esimerkin 6.4 `tyhjennaTiedosto`-aliohjelmassa tehdään. Samassa esimerkissä näkyy, miten tätä tiedoston tyhjentämistä voidaan käyttää lisäys-tilan kanssa.

Esimerkissä 6.4 näkyy myös tiedoston koodauksen määrittely. Käytännössä tämä liittyy skandinaavisiin merkkeihin, jotka eivät tulostu aina oikein Pythonin oletuskoodauksella. Skandinaaviset merkit näkyvät oikein, kun kirjainmerkkejä sisältävien tiedostojen kanssa käytetään UTF-8 -koodausta. Numeerisen tiedon kanssa tätä ongelmaa ei ole, joten numeerista tietoa sisältävien tiedostojen kanssa koodausta ei tarvitse määritellä. `open`-käskyn kolmannen parametrin käyttö näkyy `lueMerkkitiedosto`-aliohjelmassa ja se on muotoa `open(Nimi, "r", encoding="UTF-8")`.

Oletuskoodaus on erilainen eri käyttöjärjestelmissä eli joskus ääkköset saattavat toimia oikein oletusarvoilla. Tämä ei kuitenkaan ole hyvää ohjelmointityyliä, joten ohjelman ja skandinaavisten merkkien oikean toiminnan varmistamiseksi merkkejä sisältävien tiedostojen koodaus tulee määritellä UTF-8:ksi.

Esimerkki 6.4. Tiedoston avausmoodi ja koodaus

```
def kirjoitaTiedosto(Nimi, Moodi, Lukumaara):
    for i in range(Lukumaara):
        Tiedosto = open(Nimi, Moodi)
        Tiedosto.write(str(i) + " ")
        Tiedosto.close()
    return None

def tyhjennaTiedosto(Nimi):
    Tiedosto = open(Nimi, "w")
    Tiedosto.close()
    return None

def lueMerkkitiedosto(Nimi, Koodaus=None):
    Tiedosto = open(Nimi, "r", encoding=Koodaus)  # Oletusarvo None, ks. ToolTip
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        print(Rivi, end="")
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    print()
    return None

def paaohjelma():
    kirjoitaTiedosto("L06E4T1.txt", "w", 10)
    lueMerkkitiedosto("L06E4T1.txt")
    kirjoitaTiedosto("L06E4T1.txt", "a", 10)
    lueMerkkitiedosto("L06E4T1.txt")
    tyhjennaTiedosto("L06E4T1.txt")
    kirjoitaTiedosto("L06E4T1.txt", "a", 10)
    lueMerkkitiedosto("L06E4T1.txt")
    lueMerkkitiedosto("L06E4D1.txt")
    lueMerkkitiedosto("L06E4D1.txt", "UTF-8")
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
9
9 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
TÃ¤ssÃ¤ tiedostossa on merkkejÃ¤ sisÃ¤ltÃ¤en Ã¤Ã¤kkÃ¶siÃ¤.
Toisella rivillÃ¤ on Ã¥ Ã¶ Ã¤ ja Ã… Ã– Ã„.

Tässä tiedostossa on merkkejä sisältäen ääkkösiä.
Toisella rivillä on å ö ä ja Å Ö Ä.

Kiitos ohjelman käytöstä.
```

Käsiteltävät tiedostot

L06E4D1.txt

```
Tässä tiedostossa on merkkejä sisältäen ääkkösiä.
Toisella rivillä on å ö ä ja Å Ö Ä.
```

L06E4T1.txt

```
0 1 2 3 4 5 6 7 8 9
```

Jäsenfunktio-konsepti

Jäsenfunktio tarkoittaa tiettyyn muuttujatyyppiin liittyvää funktioita. Tiedostonkäsittelyn yhteydessä `open()` on tavallinen funktio, joka palauttaa tiedostokahvan. Tiedostokahva on tarkemmin ottaen olio, johon liittyy jäsenfunktioita. Näin ollen tiedostokahva-muuttujan määrittelyn jälkeen voimme viitata ko. tiedostokahvan jäsenfunktioihin pistenotaatiolla, esim. `Tiedosto.write()`, `Tiedosto.Readline()` tai `Tiedosto.close()`. Tässä vaiheessa meille riittää, että jäsenfunktion toinen nimi on metodi ja palaamme luokka/olio -konseptiin tarkemmin myöhemmin. Tiivistetysti käytämme jäsenfunktioita tarpeen mukaan, koska se on Pythonin tapa toteuttaa tietyt asiat.

Esimerkki 6.5. Jäsenfunktio-konsepti

```
def paaohjelma():
    Tiedosto = open("L06E5D1.txt", "w")
    Tiedosto.write("Testiteksti.")
    # Rivi = Tiedosto.readline()
    Tiedosto.close()
    return None

paaohjelma()
```

Tiedoston lukemisen vaihtoehtoja

Tähän mennessä olemme lukeneet tiedostoja aina `readline()`-jäsenfunktiolla, vaikka Python mahdollistaa tiedostojen lukemisen monella muullakin tavalla. Esimerkissä 6.6 näkyy neljä erilaista tapaa lukea tekstitiedosto, joista on hyvä huomata niiden pienet tekniset erot. `readline()`-jäsenfunktio on siis yksi tapa lukea tiedosto rivi kerrallaan, kun taas `readlines()`-jäsenfunktio lukee koko tiedoston yhdellä kerralla. `readlines()`-palauttaa kaikki luetut rivit yhdessä lista-tietorakenteessa, joten jos haluamme tulostaa tai käyttää näitä tietoja, pitää meidän käydä lista läpi ja tehdä vastaavat operaatiot listassa oleville tiedoille kuin `readline()`:n palauttamille riveille. Palaamme lista-rakenteeseen seuraavassa luvussa, mutta kokonaisuutena se ei muuta lukuoperaatiota merkittävästi. Kolmas saman tyylinen jäsenfunktio on `read()`, joka lukee koko tiedoston ja palauttaa sen yhtenä merkkijonona. Kuten tulosteessa näkyy, ovat rivinvaihdot alkuperäisillä paikoillaan ja käymällä tämän pitkän merkkijonon läpi, saamme paloiteltua sen rivinvaihtomerkeistä palasiin eli vastaavat asiat tehdään taas hieman eri tavalla. Tyypillinen ongelma `read()`:n

palauttaman merkkijonon kanssa on, että tiedoston loppuna oleva tyhjä rivi jää helposti kummittelemaan merkkijonoon, mikä näkyy nyt kahtena rivinvaihtona tulosteessa. Neljäs aliohjelma esittelee tiedoston lukemista `for`-rakenteella, joka tarjoaa tehokkaan tavan tiedoston rivien läpikäymiseen aiempien `for`-esimerkkien tyyliin.

Kaikki nämä ratkaisut ovat toimivia ja kaikilla niillä saa luettua tiedoston. Ratkaisuissa on pieniä eroja ja kokemattoman ohjelmoijan voi olla vaikea muistaa niitä. Siksi tässä oppaassa käytetään pääsääntöisesti vain yhtä tapaa tiedoston lukemiseen, jotta siihen liittyvät tekniset yksityiskohdat oppisi muistamaan ja toteuttamaan aina oikein. Tapauksesta riippuen tiedostojen lukeminen voi olla tehokkaampaa jollain toisella tapaa, mutta toisaalta tekniset erot johtavat helposti virheisiin. Siksi tässä oppaassa lähtökohta on opetella ensin yksi toimiva tapa lukea tekstitiedosto, jotta voimme sen jälkeen keskittyä tiedostojen sisältöön. Kun sisältöön liittyvät seikat ovat hallinnassa, voi palata näihin erilaisiin tapoihin lukea tiedosto ja miettiä optimaalisia tapoja tarpeen mukaan. Käytännössä tiedoston sisältö on tärkeämpi asia kuin tiedoston lukutapa ja siksi suositus on opetella yksi toimiva lukutapa ja käyttää sitä tämän oppaan ajan.

`readline()`-jäsenfunktio palauttaa tiedostosta yhden fyysisen rivin, joka alkaa rivinvaihdon jälkeisestä merkistä – tai tiedoston alusta – ja jatkuu rivinvaihtomerkkiin – tai tiedoston loppuun. Luetussa rivissä on viimeisenä merkkinä aina rivinvaihtomerkki paitsi, jos luetaan tiedoston viimeinen rivi ja sen perässä ei ole rivinvaihtomerkkiä. Tiedoston lopun saavutettuaan `readline()` palauttaa tyhjän merkkijonon.

Esimerkki 6.6. Tiedoston lukemisen erilaisia vaihtoehtoja

```
def riveittainWhile(Nimi):
    Tiedosto = open(Nimi, "r")
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        print(Rivi, end="")
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    print()
    return None

def kaikkiRivit(Nimi):
    Tiedosto = open(Nimi, "r")
    Rivit = Tiedosto.readlines()
    print(Rivit)
    Tiedosto.close()
    print()
    return None

def tiedostoYhdellaKertaa(Nimi):
    Tiedosto = open(Nimi, "r")
    Sisalto = Tiedosto.read()
    print(Sisalto)
    Tiedosto.close()
    print()
    return None

def riveittainFor(Nimi):
    Tiedosto = open(Nimi, "r")
    for Rivi in Tiedosto:
        print(Rivi, end="")
    Tiedosto.close()
    print()
    return None
```

```
def paaohjelma():
    riveittainWhile("L06E6D1.txt")
    kaikkiRivit("L06E6D1.txt")
    tiedostoYhdellaKertaa("L06E6D1.txt")
    riveittainFor("L06E6D1.txt")
    return None

paaohjelma()
```

Tuloste

```
Rivi1
Rivi2
Rivi3

['Rivi1\n', 'Rivi2\n', 'Rivi3\n']

Rivi1
Rivi2
Rivi3


Rivi1
Rivi2
Rivi3
```

Käsiteltävät tiedostot

L06E6D1.txt

```
Rivi1
Rivi2
Rivi3
```

Tekstitiedoston perusyksikkö on rivi

Tekstitiedostojen kirjoittaminen ja lukeminen ovat suoraviivaisia operaatioita ja sen vuoksi niitä käytetään paljon tiedon jakamisessa. Toinen asia on, että tekstitiedostossa on rajalliset keinot tietoalkioiden erotteluun ja yksilöintiin. Sisällön lähtökohtana on aiemmin mainittu rivi-orientaatio eli yhdellä rivillä on aina yksi tietoalkio. Tämä ratkaisu toimii tämän oppaan ohjelmointitehtävissä, mutta jos tietoa on paljon – ja tietoalkioita useita – ei tämä ratkaisu ole kovin höydyllinen. Siksi seuraava tehokkaampi tapa välittää tietoa on laittaa samalla riville useita tietoalkioita ja erottaa ne erotinmerkillä eli kenttäerottimella. Tällöin tiedostossa on rivejä ja kenttäerottimella erotettuja sarakkeita, ja siten tämä tiedostossa voi olla taulukkolaskentaohjelmien numerotaulukoita eli rivejä ja sarakkeita. Käytännössä taulukkolaskentaohjelmista voi tallentaa tietoja tekstitiedostoon ja näitä tietoja voi lukea itse tehdyillä ohjelmilla rivi- ja kenttäerotin-merkkejä hyödyntäen.

On syytä huomata, että taulukkolaskentaohjelmat on tyypillisesti tehty numeerisen tiedon käsittelyyn, mikä on ristiriidassa tekstitiedostojen teksti-muodon kanssa. Käytännössä tekstitiedostoissa kaikki tietoa on kirjainmerkkeinä eli merkkijonoina, myös numerot. Siksi tekstitiedostoja kirjoitettaessa kaikki tallennettavat tiedot muutetaan merkkijonoiksi ja kirjoitetaan merkkijonoja tiedostoon.

Esimerkissä 6.7 on ohjelma, joka lukee riveillä olevia tietoja ja erottaa rivillä olevat tietoalkiot sekä tulostaa ne näytölle yksittäisinä tietoalkioina. Tällaisissa sarake-dataa sisältävissä tiedostoissa on tyypillisesti otsikkorivi, jossa nimetään jokaisessa sarakkeessa olevat tiedot. Näitä otsikkotietoja ei käytetä laskennassa, joten usein tällaiset otsikkorivit luetaan pois ennen kuin siirrytään lukemaan dataa toistorakenteella. Tässä esimerkissä otsikkorivin sarakkeiden leveydet poikkeavat datarivien kenttien leveyksistä, mutta varsinaiset datarivit ovat vakiolevyisissä kentissä, jonka vuoksi kentät voidaan erottaa merkkijonosta sarake-indekseillä. Luettaessa tietoa tiedostosta on myös tarkistettava, mitä tietotyyppiä eri tiedot ovat ja sen vuoksi tämä on luonnollinen paikka muuttaa

merkkipohjainen tieto sen luonnolliseen tietotyyppiin. Tässä esimerkissä kaikki muut tiedot ovat merkkijonoja ja ainoa kokonaisluku-tieto on opiskelijan pistemäärä.

Esimerkki 6.7. Tekstitiedostot perustuvat riveihin

```
def paaohjelma():
    Tiedosto = open("L06E7D1.txt", "r")
    Tiedosto.readline() # Otsikkorivin lukeminen pois
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        Kurssi = Rivi[0:9]
        Opiskelija = Rivi[10:18]
        Tehtava = Rivi[19:24]
        Pisteet = int(Rivi[25:26])
        print(Kurssi, Opiskelija, Tehtava, Pisteet)
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    print()
    return None

paaohjelma()
```

Tuloste

```
CT60A0203 01234567 L01T1 1
CT60A0203 01234567 L01T2 1
CT60A0203 01234567 L01T3 0
```

Käsiteltävät tiedostot

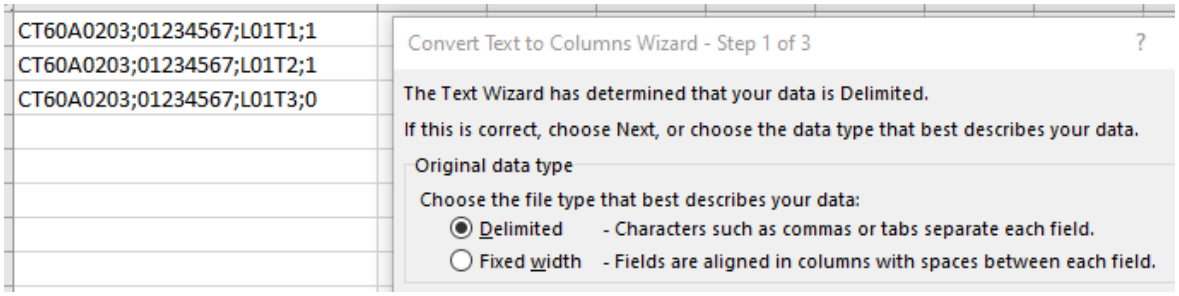
L06E7D1.txt

```
Kurssi;Opiskelija;Tehtävä;Pisteet
CT60A0203;01234567;L01T1;1
CT60A0203;01234567;L01T2;1
CT60A0203;01234567;L01T3;0
```

Esimerkissä 6.7 varsinaiset tiedot olivat vakio-levyisissä kentissä ja tiedot voitiin näin ollen erottaa toisistaan merkkijonon indekseillä. Tämä on yksi tyypillinen tapa jakaa tietoa ja siksi taulukkolaskentaohjelmissa on mahdollisuus tuoda tietoa sekä jakaa se sarakkeisiin. Kuvassa 6.1 näkyy taulukkolaskentaohjelman velho auttamassa tiedon jakamisessa sarakkeisiin. Tässä tapauksessa velho on tunnistanut tiedon perustuvan erotinmerkkiin (;) ja ehdottaa, että tieto jaetaan sarakkeisiin näiden merkkien perusteella. Toisena vaihtoehtona tässä kuvassa näkyy vakio-levyiset kentät, mikä esimerkissä 6.7 käytettiin hyväksi.

Kuvassa 6.2 esimerkki sähkölaitokselta saatavan sähkönkulutus-tiedoston alusta. Tiedostossa on ensin otsikkorivi ja sen jälkeen halutulta ajalta kulutustietoja tunnin välein. Tässä tiedostossa on myös lämpötila, sillä se vaikuttaa tyypillisesti sähkönkulutukseen erityisesti sähkölämmitteisissä taloissa. Tässäkin tiedostossa tiedot on jaettu kenttiin erotinmerkillä, joka on puolipiste (;). Monissa muissa maissa kenttäerottimena toimii pilkku, mutta Suomessa käytetään desimaalipisteen sijaan desimaalipilkkua, joten kenttäerottimena käytetään tyypillisesti muita merkkejä kuten puolipiste (;) ja tabulaattori (\t).

Kuva 6.1. Riviorientoituneet tietoalkiot, erotinmerkki ja kenttäleveydet



Esimerkissä 6.7 kenttien tunnistamiseen käytettiin sarakeleveyksiä, mutta kuvan 6.1 mukaisesti useimmiten kentät erotetaan kenttäerottimella. Palaamme tähän asiaan myöhemmin tässä luvussa, kun tutustumme tarkemmin merkkijonojen jäsenfunktioihin.

Kuva 6.2. Esimerkki tekstitiedostona sähkölaitokselta saatavasta sähkönkulutustiedoston alusta

```
Ajankohta;Kulutus (kWh);Ulkolämpötila (°C)
maanantai 1.1.2018 00:00;0,8;-0,1
maanantai 1.1.2018 01:00;0,85;-0,2
maanantai 1.1.2018 02:00;0,85;-0,2
```

Tiedostonkäsittelyn rajoitteita

Tiedostonkäsittelyyn liittyy paljon erilaisia asioita ja tässä oppaassa keskitymme tekstitiedoston perusoperaatioihin. Käytännössä jätämme monta kiinnostavaa konseptia käsittelemättä ja jokainen voi tutustua niihin tarkemmin esim. Python-dokumentaation avulla tarpeen mukaan. Alla on muutamia tällaisia keskeisiä asioita.

- **Kirjanmerkki**. Python-tulkki tietää tiedoston käsittelykohdan ja ylläpitää sitä kirjanmerkki-tiedossa. Käytännössä kirjoitettaessa tiedostoa kirjanmerkki on aina tiedoston lopussa, mutta kirjanmerkin sijainti voidaan selvittää jäsenfunktiolla ja siirtää se toiseen paikkaan toisella jäsenfunktiolla jne. Edistyneen tiedostonkäsittelyn yhteydessä nämä ovat normaaleja operaatioita, mutta tässä vaiheessa tärkeintä on perusoperaatioiden virheetön suorittaminen eikä kirjanmerkkiä tarvita silloin. Luettaessa tiedostosta yksi rivi, siirtyy kirjanmerkki luetun rivinvaihtomerkin jälkeiseen merkkiin odottamaan seuraavaa lukuoperaatiota ja tämä riittää meille.
- **Virheenkäsittely**. Kuten edellä oli puhetta, etenkin tiedostojen luku-operaatiot ovat virhe-herkkiä operaatioita. Tässä vaiheessa tavoitteemme on tehdä ohjelmia, jotka toimivat oikein käytettynä oikein. Toisin sanoen luemme vain olemassa olevia tiedostoja jne. emmekä näin tarvitse tässä vaiheessa virheenkäsittelyä. Palaamme virheenkäsittelyyn omassa luvussa ja tällöin esittelemme tämän oppaan laajuudessa sopivan virheenkäsittelykonseptin.

Merkkijonojen muotoilu

Kuten edellä on käynyt ilmi, perustuvat tekstitiedostot merkkijonoihin ja merkkijonojen käsittely on keskeinen osa tekstitiedostojen käsittelyä. Olemme aiemmin muodostaneet merkkijonoja yhdistämällä niitä +-merkillä, muuttaneet numeroiden esitystarkkuutta round()-funktiolla ja tehneet numeroista merkkijonoja str()-funktiolla. Merkkijonoista taas on erotettu alimerkkijonoa leikkauksilla ([::]) ja merkkijonoista on saatu numeroita tietotyyppimuunnoksilla int() ja float(). Näillä tekniikoilla pystyy edelleen tekemään suurimman osan tarvittavista operaatioista, mutta tekstitiedostojen kanssa merkkijono-operaatioita tulee helpolla paljon ja suurien datamäärien kanssa toimiessa käsittelyssä

tarvitaan usein aiempaa enemmän mahdollisuuksia. Siksi tutustumme seuraavaksi aiempaa tehokkaampiin tapoihin käsitellä merkkijonoja.

Tässä luvussa tutustumme merkkijonojen muokkaamiseen jäsenfunktioilla ja muodostamiseen format-jäsenfunktiolla. Näillä pystymme muodostamaan tehokkaasti halutunlaisia merkkijonoja, jotka voidaan kirjoittaa tekstitiedostoon tai tulostaa näytölle. Näillä työkaluilla pystyy myös selvittämään tekstitiedostoista luetuista merkkijonoista monia hyödyllisiä tietoja, mikä helpottaa ja tehostaa niiden jatkokäsittelyä ohjelmilla.

Merkkijonojen jäsenfunktioita

Taulukossa 6.1 on nimetty yleisimpiä Pythonin merkkijonojen jäsenfunktioita ja esimerkissä 6.8 on esitelty niiden käyttöä. Tässä vaiheessa kannattaa katsoa, minkälaisia jäsenfunktioita merkkijonoilla on ja opetella käyttämään niitä aina tarpeen mukaan. Käytön yksityiskohdat voi tarkistaa aina helpistä ja ToolTip on hyvä pika-apu niiden käyttöön.

Taulukko 6.1. Pythonin yleisiä merkkijonojen jäsenfunktioita.

Jäsenfunktio	Kuvaus
capitalize()	Palauttaa merkkijonosta kopion, jossa ensimmäinen merkki on muutettu isoksi kirjaimeksi.
endswith(*testi*[, *start*[, *end*]]**)**	Palauttaa arvon `True`, jos merkkijono päättyy merkkeihin *testi*, muutoin palauttaa arvon `False`. Lisäkytkimet start ja end, joilla voidaan määrätä testattava alue.
startswith(*prefix*[, *start*[, *end*]]**)**	Sama kun endswith() mutta testaa merkkijonon alkua.
expandtabs([*tabsize*])	Palauttaa merkkijonosta kopion, jossa kaikki sisennysmerkit (välilyönti ja sarkainmerkki(tab)) on korvattu tabsize –määrällä välilyöntejä. `Tabsize` on oletusarvoisesti 8. (Huom! IDLEssä 4)
find(*sub*[, *start*[, *end*]]**)**	Palauttaa ensimmäisen merkin sijainnin, jos annettu merkkijono *sub* löytyy testijonosta. Lisäkytkimet start ja end mahdollistavat hakualueen rajaamisen. Palauttaa arvon -1, jos merkkijonoa ei löydy.
format(*parametrilista***)**	Muotoilee parametrilistassa olevat arvot ja sijoittaa ne merkkijonoon. Ks. tarkemmin yllä *Muotoiltu tulostus* –kohta.
index(*sub*[, start[, end]]**)**	Kuin `find()`, mutta palauttaa virheen `ValueError`, jos merkkijonoa *sub* ei löydy.
isalnum()	Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat alphanumeerisia merkkejä, eli kirjaimia tai numeroita, ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.
isalpha()	Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat kirjaimia ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.
isdigit()	Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat numeroita ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.
islower()	Palautttaa arvon `True`, jos kaikki merkkijonon merkit ovat pieniä kirjaimia ja merkkijonossa on ainakin yksi kirjain. Muutoin `False`.
isupper()	Sama kuin `islower()`, mutta isoille kirjaimille.
isspace()	Palautttaa arvon `True`, jos kaikki merkkijonon merkit ovat välilyöntejä ja merkkijonossa on ainakin yksi merkki. Muutoin `False`.
lower()	Palauttaa merkkijonon kopion, jossa kaikki kirjaimet on muutettu

	pieniksi kirjaimiksi.
lstrip([chars])	Palauttaa merkkijonon, josta on vasemmasta reunasta poistettu kaikki määritellyt merkit. Jos poistettavia merkkejä ei määritellä, poistetaan pelkästään välilyönnit ja sisennykset. Tasaus on aina vasemmassa reunassa, ensimmäinen ei-poistettava merkki määrää tasauksen paikan. Oikean reunan tasaamiseen käytetään vastaavaa funktiota **rstrip()**.
strip([*chars*])	Palauttaa merkkijonosta kopion, josta on poistettu merkkijonon alusta ja lopusta kaikki määritellyt merkit. Käytännössä yhtäaikainen `lstrip()` ja `rstrip()`. Jos poistettavia merkkejä ei määritellä, poistetaan välilyönnit ja sisennykset.
replace(*old*, *new*[, *count*]**)**	Korvaa merkkijonosta merkkiryhmän *old* jäsenet merkkiryhmällä *new*. Voidaan myös antaa lisätietona parametri *count*, joka määrää kuinka monta korvausta maksimissaan tehdään.
split([sep[, *maxsplit*]]**)**	Halkaisee merkkijonon erotinmerkin mukaisesti listaksi. Voi saada myös parametrin *maxsplit*, joka kertoo miten monesta kohtaa merkkijono korkeintaan halkaistaan (eli listassa on silloin *maxsplit*+1 alkiota). Jos erotinmerkkiä ei anneta, käytetään oletuserottimena välilyöntiä. Itse lista-asiaan palataan myöhemmin.
upper()	Muuttaa kaikki merkkijonon kirjaimet isoiksi kirjaimiksi.

Esimerkissä 6.8 näkyy useiden merkkijonojen jäsenfunktioiden käyttö eli miten ja mihin niitä käytetään, kun taas Taulukossa 6.1 on kerrottu lyhyesti jäsenfunktioiden käytöstä. Usein kannattaa katsoa dokumenteista jäsenfunktion toiminta ja sitten kokeilla itse, miten se toimii ja vastaako se omaa tarvetta. Esimerkiksi isalnum()-jäsenfunktio testaa, ovat kaikki merkkijonon merkit kirjaimia tai numeroita, ja isalpha() taas testaa, ovatko kaikki kirjaimia. Myös isdigit() käy merkkijonon kaikki merkit läpi yksitellen selvittääkseen ovatko kaikki merkit numeroita ja näin ollen sen mukaan merkkijono ”-1” ei ole numero, koska -merkki ei ole numero. Esimerkin koodi näyttää miten jäsenfunktiot toimivat, kun taas sen tulosteesta näkee nopeammin, mitä ne tekevät.

Yksittäisen jäsenfunktioiden jälkeen esimerkissä 6.8 näkyy, miten merkkijonon kaikki merkit voidaan käydä läpi ja testata sopivilla jäsenfunktioilla. Tämä auttaa usein ymmärtämään, mitkä merkit ovat kiinnostavia ja/tai höydyllisiä missäkin kohdassa, tai aiheuttavat ongelmia ohjelmassa.

Esimerkki 6.8. Merkkijonojen jäsenfunktioita. Tämä esimerkki ei ole ohjelma vaan se esittelee käskyjen käyttöä eikä tässä siksi ole pääohjelma-rakennetta.

```
Merkkijono = "Tämä on LAUSE.   "
print("Tietoa merkkijonosta is-jäsenfunktioilla:")
print("'merkkijono'.isalnum() ==", "merkkijono".isalnum())
print("'merkkijono'.isalpha() ==", "merkkijono".isalpha())
print("'merkkijono'.islower() ==", "merkkijono".islower())
print("'Merkkijono'.islower() ==", "Merkkijono".islower())
print("'12345'.isalpha() ==", "12345".isalpha())
print("'12345'.isdigit() ==", "12345".isdigit())
print("'-1'.isdigit() ==", "-1".isdigit())
print("' \\t\\n'.isspace() ==", " \t\n".isspace())
print()

print("Merkkijonon muokkaus jäsenfunktioilla:")
print("'Tämä on LAUSE.'.lower() =>", "Tämä on LAUSE.".lower())
print("'Tämä on LAUSE.'.upper() =>", "Tämä on LAUSE.".upper())
print("'Tämä on LAUSE.'.capitalize() =>", "Tämä on LAUSE.".capitalize())
Merkkijono = "  Tämä on LAUSE.         "
print("'"+Merkkijono+"'.strip() =>", "'"+Merkkijono.strip()+"'")
print("'www.example.com'.strip(\"cmow.\") =>",
            "'"+"www.example.com".strip("cmow.")+"'")
print("'"+Merkkijono+"'.replace(\" \", \"#\") =>",
            "'"+Merkkijono.replace(" ", "#")+"'")
print()

Merkkijono = "Nro 1."
print("Merkkijono '"+Merkkijono+"' käsiteltynä merkki kerrallaan:")
for i in range(len(Merkkijono)):
    print(Merkkijono[i]+".isalpha() ==", Merkkijono[i].isalpha())
print()
```

Tuloste

```
Tietoa merkkijonosta is-jäsenfunktioilla:
'merkkijono'.isalnum() == True
'merkkijono'.isalpha() == True
'merkkijono'.islower() == True
'Merkkijono'.islower() == False
'12345'.isalpha() == False
'12345'.isdigit() == True
'-1'.isdigit() == False
' \t\n'.isspace() == True

Merkkijonon muokkaus jäsenfunktioilla:
'Tämä on LAUSE.'.lower() => tämä on lause.
'Tämä on LAUSE.'.upper() => TÄMÄ ON LAUSE.
'Tämä on LAUSE.'.capitalize() => Tämä on lause.
'  Tämä on LAUSE.         '.strip() =>  'Tämä on LAUSE.'
'www.example.com'.strip("cmow.") =>  'example'
'  Tämä on LAUSE.         '.replace(" ", "#") =>
'##Tämä#on#LAUSE.#########'

Merkkijono 'Nro 1.' käsiteltynä merkki kerrallaan:
N.isalpha() True
r.isalpha() True
o.isalpha() True
 .isalpha() False
1.isalpha() False
..isalpha() False
```

Merkkijonojen muotoilu format()-jäsenfunktiolla

Merkkijonojen muodostamisen kannalta keskeisin jäsenfunktio on format(), jonka lähtökohtana on merkkijonon prototyypin määrittely. Tässä merkkijonon prototyypissä näkyy ensinnäkin kohdat, joihin muuttujien arvot tullaan laittamaan ja toiseksi prototyypissä voidaan määritellä tarkasti näiden arvojen esitysasu.

format()-jäsenfunktion lähtökohta on merkkijono, jossa tulostettaville arvoille on varattu omat paikat ja itse arvot ovat jäsenfunktion parametreina. Käytännössä tämä helpottaa tulevan merkkijonon hahmottamista kokonaisuutena, koska ensin määritellään merkkijono ja muuttujien kohdalle jätetään merkit, joiden perusteella niihin voidaan sijoittaa halutut arvot. Esimerkissä 6.9 näkyy kaksi tapaa muodostaa merkkijono eli aiempi yhdistämiseen perustuva tapa (Rivi1) ja format()-jäsenfunktioon perustuva tapa (Rivi2). Molemmissa tavoissa on pääpiirteissään yhtä paljon koodia, mutta format()-jäsenfunktio ryhmittelee asiat eri tavoin. format-jäsenfunktiota kutsutaan merkkijonolle, joka voi olla muuttuja tai lainausmerkeissä esimerkin 6.9 mukaisesti. Merkkijono sisältää tulostettavat merkit eli tässä tapauksessa monta kenttäerotinta ”;” ja rivinvaihtomerkki ”\n”, joiden lisäksi merkkijonossa on tulostettavien muuttujien paikat merkitty { }-suluilla ja niiden sisällä on tulostettavien muuttujien indeksi 0-3. Itse muuttujat ovat indeksien mukaisessa järjestyksessä format():n parametreina. Molemmat esimerkin 6.9 tavat johtavat samanlaiseen tulosteeseen, mutta yhdistämällä ei pääse paljoa tätä pidemmälle, kun taas format()-jäsenfunktion osalta muuttujien tulostusasun muotoiluun tutustumme seuraavaksi.

Esimerkki 6.9. Merkkijonon tietoalkioiden paikat ja arvot format-jäsenfunktiossa

```
Kurssi = "CT60A0203"
Opiskelija = "01234567"
Tehtava = "L01T1"
Pisteet = 1
Rivi1 = Kurssi+";"+Opiskelija+";"+Tehtava+";"+str(Pisteet)+"\n"
print(Rivi1, end="")
Rivi2 = "{0};{1};{2};{3}\n".format(Kurssi, Opiskelija, Tehtava, Pisteet)
print(Rivi2, end="")
```

Tuloste

```
CT60A0203;01234567;L01T1;1
CT60A0203;01234567;L01T1;1
```

format()-jäsenfunktio mahdollistaa muuttujien tulostusasun muotoilun, jolloin voimme toteuttaa esimerkiksi pyynnöt ”tulosta nelinumeroinen luku” tai ”tulosta desimaaliluku kahdella desimaalilla”. Todellisuudessa emme ole tähän asti pystyneet muotoilemaan *tulosteita* vaan esim. round()-funktio pyöristää luvun haluttuun tarkkuuteen ja mahdollistaa siten halutun *numeerisen tarkkuuden* saavuttamisen. Emme siis ole tähän asti pystyneet määrittämään tulostettavien desimaalien määrää tai tulosteen käyttämän kentän leveyttä. format()-jäsenfunktio mahdollistaa näiden lisäksi monta muuta muotoiluun liittyvää asiaa esimerkin 6.10 mukaisesti. Kuten esimerkissä 6.9 näkyy, sijoitetaan arvot tulosteen prototyyppiin { } -sulkujen paikalle, esim. {0}, jolloin Rivi2 -merkkijonoon tulee tähän kohtaa Kurssi-muuttujan arvo ”CT60A0203”. Muuttujan arvon muotoilu taas näkyy hyvin esimerkin 6.10 *Tyypillisiä tulosteita* -kohdassa ”Desimaaliluku ’{0:5.2f}’”, jossa heittomerkkien sisälle sijoitetaan luku 12.3456 muodossa 5.2 eli 5-merkkiä leveään kenttään kahdella desimaalilla. Tulosteessa näkyy kaksi desimaalia pyöristettynä, niiden edessä piste ja kokonaislukuosa kahdella numerolla eli ’12.35’. Esimerkissä näkyy myös muita tulosteiden muotoilumahdollisuuksia ja parhaiten niiden toiminta selviää kokeilemalla itse vastaavia tulosteita ja vaihtamalla muotoiluja sopivasta. Viimeisessä esimerkissä desimaaliluvun tarkkuus määritellään muuttujalla, joka on harvoin tarpeen, mutta

mahdollistaa tilanteeseen sopivien tulosteiden tekemisen.

Esimerkki 6.10. Tulosteiden esitysasun muotoilu format-jäsenfunktiolla

```
Kokonaisluku = 51
Desimaaliluku = 12.3456
Tarkkuus = 8

print("format-jäsenfunktion oletusarvot ja tietotyypit:")
print("Kokonaisluku '{0}', desimaaliluku '{1}', numero '{2}'".
            format(Kokonaisluku, Desimaaliluku, 3))
print("Kokonaisluku '{0:d}', desimaaliluku '{1:f}', numero '{2:d}'".
            format(Kokonaisluku, Desimaaliluku, 3))
print()

print("Tyypillisiä tulosteita:")
print("Kokonaisluku '{0:1d}'".format(Kokonaisluku))
print("Desimaaliluku '{0:.2f}'".format(Desimaaliluku))
print("Desimaaliluku '{0:5.2f}'".format(Desimaaliluku))
print()

print("Hyödyllisiä ominaisuuksia:")
print("Desimaaliluku '{0:10.2f}'".format(Desimaaliluku))
print("Desimaaliluku '{0:<10.2f}'".format(Desimaaliluku))
print("Merkkijono '{0:^10s}'".format("Ville"))
print("Desimaaliluku '{0:.1f}', '{0:.2f}', '{0:.3f}'".format(Desimaaliluku))
print("Desimaaliluku tarkkuus muuttujasta '{0:.{1}f}'".format(Desimaaliluku,
                                                              Tarkkuus))
print()
```

Tuloste

```
format-jäsenfunktion oletusarvot ja tietotyypit:
Kokonaisluku '51', desimaaliluku '12.3456', numero '3'
Kokonaisluku '51', desimaaliluku '12.345600', numero '3'

Tyypillisiä tulosteita:
Kokonaisluku '51'
Desimaaliluku '12.35'
Desimaaliluku '12.35'

Hyödyllisiä ominaisuuksia:
Desimaaliluku '     12.35'
Desimaaliluku '12.35     '
Merkkijono '  Ville   '
Desimaaliluku '12.3', '12.35', '12.346'
Desimaaliluku tarkkuus muuttujasta '12.34560000'
```

Tässä oppaassa tulostuksen muotoilu on edistynyt konsepti ja kaikki oppaan tulosteet pystyy tekemään ilman sitäkin. Tulosteiden muotoilu on kuitenkin niin tärkeä asia, että tästä konseptista on syytä olla tietoinen ja siihen kannattaa tutustua tarkemmin viimeistään silloin, kun aiemmat merkkijonojen muokkauskeinot tuntuvat työläiltä ja asiaan rupeaa menemään liikaa aikaa. format()-jäsenfunktio on niin monipuolinen, että sen tehokkaan käytön opettelu vaatii oman aikansa, mutta vastapainoksi se mahdollistaa monipuolisen tulosteiden muotoilun. Taulukossa 6.2 näkyy keskeisimmät tulostuksen muotoilukoodit ja niiden tarkoitukset.

Taulukko 6.2. Muotoillun tulostuksen muotoilukoodit

Muotoilukoodi	Merkitys
i (tai d)	Etumerkillinen 10-kantainen kokonaisluku
o	Etumerkillinen oktaaliluku
x	Etumerkillinen heksadesimaaliluku (pienillä kirjaimilla)
X	Etumerkillinen heksadesimaaliluku (isoilla kirjaimilla)
e	Desimaaliluku eksponentaaliesityksellä (pienillä kirjaimilla)
E	Desimaaliluku eksponentaaliesityksellä (isoilla kirjaimilla)
f (tai F)	Desimaaliluku
g	Desimaaliluku. Käyttää eksponentiaaliesitystä (pienillä kirjaimilla), jos eksponentti on vähemmän kuin -4 tai vähemmän kuin käytössä oleva tarkkuus – muuten desimaalimuoto.
G	Desimaaliluku. Käyttää eksponentiaaliesitystä (isoilla kirjaimilla), jos eksponentti on vähemmän kuin -4 tai vähemmän kuin käytössä oleva tarkkuus – muuten desimaalimuoto.
c	Yksi merkki (hyväksyy kokonaisluvun tai yhden merkin merkkijonon)
r	Merkkijono (muuntaa minkä tahansa Python objektin merkkijonoksi käyttäen repr()-funktiota)
s	Merkkijono (muuntaa minkä tahansa Python objektin merkkijonoksi käyttäen str()-funktiota)
a	Merkkijono (muuntaa minkä tahansa Python objektin merkkijonoksi käyttäen ascii()-funktiota)
%	%-merkki
>	Tulosteen tasaus oikealle
<	Tulosteen tasaus vasemmalle
^	Tulosteen keskittäminen kenttään.

Tyypillisiä tiedostoihin liittyviä tehtäviä

Esimerkissä 6.11 näkyy merkkijonon pilkkominen tietoalkioihin erotinmerkkien osoittamista kohdista. Tyypillisesti tällainen merkkijono on tiedostosta luettu rivi ja meillä on tiedossa tietoalkioiden erottimena toimiva merkki. Merkkijono käydään läpi merkki kerrallaan ja muodostetaan niiden välissä olevista merkeistä erilliset tietoalkiot. Tässä esimerkissä tietoalkiot jäävät merkkijonoiksi, mutta tämän jälkeen ne voi muuttaa tarvittaessa sopivaan tietotyyppiin ja sijoittaa muuttujaan jne. Huomaa, että esimerkin lopussa merkkijono jaetaan osiin erotinmerkin kohdasta split()-jäsenfunktiolla, joka tekee saman asian paljon helpommin. Tässä vaiheessa ongelma on, että split()-jäsenfunktio palauttaa lista-tyyppisen tietoalkion, joka ei ole meille vielä tuttu vaan se käsitellään vasta seuraavassa luvussa.

Esimerkki 6.11. Rivin pilkkominen osiin erotinmerkistä

```
def pilkoRivi(Rivi):
    Tietoalkio = ""
    print("Rivi '"+Rivi+"' jaettuna osiin ;-merkin kohdalta:")
    for i in range(len(Rivi)):
        if (Rivi[i] == ";"):
            print(Tietoalkio)
            Tietoalkio = ""
        elif (Rivi[i] != "\n"):
            Tietoalkio = Tietoalkio + Rivi[i]
    print(Tietoalkio)
    print()

    ListaTietoalkioita = Rivi.split(";")
    print("Sama rivi jaettuna osiin .split(\";\")-jäsenfunktiolla: ")
    print(ListaTietoalkioita)
    return None
```

```
def paaohjelma():
    pilkoRivi("CT60A0203;01234567;L01T1;1\n")
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Rivi 'CT60A0203;01234567;L01T1;1
' jaettuna osiin ;-merkin kohdalta:
CT60A0203
01234567
L01T1
1

Sama rivi jaettuna osiin .split(";")-jäsenfunktiolla:
['CT60A0203', '01234567', 'L01T1', '1\n']
Kiitos ohjelman käytöstä.
```

Esimerkissä 6.12 etsitään tiedostosta sopivin alkio, joka tarkoittaa tällä kertaa lyhyimmän nimen pituuden selvittämistä. Tämä löytyy käymällä läpi kaikki tiedostossa olevat nimet ja etsimällä niistä lyhyimmän pituus eli ottamalla vertailukohdaksi ensimmäisen nimen pituus ja aina kun löytyy sitä lyhyempi pituus, vaihtamalla se uudeksi vertailupituudeksi. Tällä algoritmilla löytyy lyhyin pituus, mutta luonnollisesti käyttäjä haluaa normaalisti tietää myös mikä tämä nimi oli. Ongelmaksi saattaa muodostua se, että tiedostossa on monta yhtä lyhyttä nimeä, kuten esim. Mia ja Ari. Alla oleva ohjelma raportoisi aina ensimmäisenä löytyvän lyhyimmän nimen, mutta pienellä muokkauksella se raportoisi viimeisen lyhyimmän nimen. Tiivistetysti esimerkin 6.12 ohjelma sopii hyvin sopivimman arvon etsimiseen, mutta tyypillisesti sitä pitää aina muokata hieman, jotta se löytyy juuri pyydetyn arvon eikä sinne päin.

Esimerkki 6.12. Sopivimman arvon etsiminen

```
def lyhyin(Nimi):
    LyhyinPituus = None
    Tiedosto = open(Nimi, "r", encoding="UTF-8")
    Rivi = Tiedosto.readline()
    while (len(Rivi)> 0):
        NimiPituus = len(Rivi) - 1
        if ((LyhyinPituus == None) or (LyhyinPituus > NimiPituus)):
            LyhyinPituus = NimiPituus
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    return LyhyinPituus

def paaohjelma():
    Pituus = lyhyin("L06E3D1.txt")
    print("Lyhyin nimi oli "+str(Pituus)+" merkkiä.")
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Lyhyin nimi oli 3 merkkiä.
Kiitos ohjelman käytöstä.
```

Esimerkissä 6.13 oleva ohjelma lukee tiedoston ja yhdistää samaan kuuluvat tiedot yhteen eli ryhmittelee tiedot. Tämä tehtävä on tyypillinen tehtävä isojen tietojoukkojen kanssa, sillä esim. sähkökäyttödata voi olla saatavilla tunneittain ja niiden sijaan tarvitaankin päiväkulutuksia. Tehtävä edellyttää luonnollisesti kaikkien tietorivien läpikäyntiä ja yhdistelyä sopivasti eli tilanteeseen sopivaa algoritmia, joka seuraa rivin alussa olevaa ryhmätunnistetta ja sen vaihtuessa tulostaa edellisen ryhmän tiedot ja rupeaa laskemaan uuden ryhmän tietoja.

Esimerkki 6.13. Tietojen ryhmittely, operointi osajoukon kanssa

```
def ryhmittele(Nimi):
    Rivimaara = 0
    Summa = 0
    Tiedosto = open(Nimi, "r")
    Rivi = Tiedosto.readline()
    RyhmaEdellinen = Rivi[0:2]
    while (len(Rivi)> 0):
        Rivimaara = Rivimaara + 1
        Ryhma = Rivi[0:2]
        Data = int(Rivi[3:4])
        if (Ryhma == RyhmaEdellinen):
            Summa = Summa + Data
        else:
            print("Ryhmän {0:s} summa on {1:d}".format(RyhmaEdellinen, Summa))
            Summa = Data
            RyhmaEdellinen = Ryhma
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    print("Ryhmän {0:s} summa on {1:d}".format(RyhmaEdellinen, Summa))
    print("Tiedostossa oli {0} riviä.".format(Rivimaara))
    return None

def paaohjelma():
    ryhmittele("L06E13D1.txt")
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Ryhmän Ma summa on 5
Ryhmän Ti summa on 21
Tiedostossa oli 6 riviä.
Kiitos ohjelman käytöstä.
```

Käsiteltävät tiedostot

L06E13D1.txt

```
Ma;4
Ma;1
Ti;2
Ti;3
Ti;9
Ti;7
```

Esimerkissä 6.14 on valikkopohjainen ohjelma, joka perustuu tiedostojen lukemiseen ja kirjoittamiseen. Itse tehtävän ydin on sama kuin edellisessä luvussa, mutta tällä kertaa merkkijono luetaan tiedostosta ja tallennetaan etu- tai takaperin tiedostoon. Tämä tehtävä on pääosin aiemmin läpikäytyjen asioiden kertausta, mutta kannattaa huomata pääohjelmassa oleva muuttuja MerkkijonoLuettu. Tällä muuttujalla estetään ohjelman kaatuminen siinä

tapauksessa, että käyttäjä yrittäisi kirjoittaa tietoja ennen kuin Merkkijono-muuttuja on saanut arvon. Palaamme virheenkäsittelyyn tarkemmin myöhemmin, mutta tämä muuttuja on yksi yksinkertainen keino estää ohjelman kaatuminen.

Esimerkki 6.14. Valikkopohjainen ohjelma tiedostoilla

```
def valikko():
    print("Valitse haluamasi toiminto:")
    print("1) Lue merkkijono tiedostosta")
    print("2) Tallenna merkkijono etuperin")
    print("3) Tallenna merkkijono takaperin")
    print("0) Lopeta")
    Valinta = int(input("Anna valintasi: "))
    return Valinta

def lueMerkkijono(Nimi): # Lukee aina ensimmäisen rivin
    Tiedosto = open(Nimi, "r", encoding="UTF-8")
    Rivi = Tiedosto.readline()
    Tiedosto.close()
    Merkkijono = Rivi[:-1]
    return Merkkijono

def tallenna(Nimi, Merkit):
    Tiedosto = open(Nimi, "a", encoding="UTF-8")
    Rivi = Merkit + '\n'
    Tiedosto.write(Rivi)
    Tiedosto.close()
    Tulosta = "Kirjoitettu tiedostoon '"+Nimi+"' merkkijono '"+Merkit+"'."
    print(Tulosta)
    return None

def paaohjelma():
    Valinta = 1
    MerkkijonoLuettu = False
    TiedostoLue = "L06E13D1.txt"
    TiedostoKirjoita = "L06E13T1.txt"
    while (Valinta != 0):
        Valinta = valikko()
        if (Valinta == 1):
            Merkkijono = lueMerkkijono(TiedostoLue)
            MerkkijonoLuettu = True
            print("Luettu tiedostosta merkkijono '{0}'.".format(Merkkijono))
        elif (Valinta == 2):
            if (MerkkijonoLuettu == True):
                tallenna(TiedostoKirjoita, Merkkijono)
            else:
                print("Lue merkkijono ennen sen tallentamista.")
        elif (Valinta == 3):
            if (MerkkijonoLuettu == True):
                tallenna(TiedostoKirjoita, Merkkijono[::-1])
            else:
                print("Lue merkkijono ennen sen tallentamista.")
        elif (Valinta == 0):
            print("Lopetetaan.")
        else:
            print("Tuntematon valinta, yritä uudestaan.")
        print()
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
```

Tuloste

```
Valitse haluamasi toiminto:
1) Lue merkkijono tiedostosta
2) Tallenna merkkijono etuperin
3) Tallenna merkkijono takaperin
0) Lopeta
Anna valintasi: 2
Lue merkkijono ennen sen tallentamista.

Valitse haluamasi toiminto:
1) Lue merkkijono tiedostosta
2) Tallenna merkkijono etuperin
3) Tallenna merkkijono takaperin
0) Lopeta
Anna valintasi: 1
Luettu tiedostosta merkkijono 'Tässä tiedostossa on merkkijonoja.'.

Valitse haluamasi toiminto:
1) Lue merkkijono tiedostosta
2) Tallenna merkkijono etuperin
3) Tallenna merkkijono takaperin
0) Lopeta
Anna valintasi: 2
Kirjoitettu tiedostoon 'L06E13T1.txt' merkkijono 'Tässä tiedostossa on
merkkijonoja.'.

Valitse haluamasi toiminto:
1) Lue merkkijono tiedostosta
2) Tallenna merkkijono etuperin
3) Tallenna merkkijono takaperin
0) Lopeta
Anna valintasi: 0
Lopetetaan.

Kiitos ohjelman käytöstä.
```

Yhteenveto

Osaamistavoitteet

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

- Tiedostonkäsittely
  - Tiedoston kirjoitus ja luku (E6.1)
  - Tiedostossa monta riviä ja monta tiedostoa (E6.2, 6.3)
  - Tiedoston avaaminen (E6.4)
  - Tiedostonkäsittelyn jäsenfunktiot (E6.5)
  - Tiedoston lukemisen vaihtoehtoja (E6.6)
  - Tekstitiedostojen rivit ja tietoalkiot (E6.7, Kuva 6.1, E6.14)
  - Tiedostonkäsittelyn rajoitteita tässä oppaassa
  - Tyypillisiä ongelmia ja virheitä
- Muotoiltu tulostus
  - Merkkijonojen jäsenfunktioita (Taulukko 6.1, E6.8)
  - Merkkijonojen muotoilu format-jäsenfunktiolla
    - tietoalkioiden paikat ja arvot (E6.9)
    - tulosteiden esitysasujen muotoilu (E6.10, Taulukko 6.2)

- Tyypillisiä tehtäviä tiedostoihin liittyen
    - Sopivimman arvon etsiminen (E6.11)
    - Tietojen ryhmittely (E6.12)
    - Valikkopohjainen ohjelma tiedostojen käsittelyyn (E6.13, E6.14)

Tiedostonkäsittelyssä on taas paljon huomioitavia asioita, joten kannattaa käydä läpi keskeiset asiat ja aloittaa tehtävien tekeminen niiden pohjalta sekä tutustua muihin asioihin tarpeen mukaan. Myös merkkijonoilla on paljon jäsenfunktioita, joten niiden käyttöä ei kannata opetella ulkoa vaan kannattaa hahmottaa, minkälaisia jäsenfunktioita on, että osaa etsiä sopivaa sitten kun tarve tulee. Useimmat jäsenfunktiot ovat yksinkertaisia käyttää, mutta dokumenteista kannattaa katsoa niiden keskeiset ominaisuudet, jotta ymmärtää ne oikein eikä yritä käyttää niitä sopimattomissa paikoissa.

Tyyliohjeet

Tähän lukuun liittyvät tyyliohjeet on koottu alle. Tiedostojen käsittelyyn on paljon erilaisia vaihtoehtoja ja tärkeintä on opetella yksi toimiva tapa, jotta pääsee keskittymään tiedostojen sisältöön ja käyttöön. Alla olevia ohjeita noudattamalla vältät useimmat tiedostonkäsittelyyn liittyvät virheet.

1. Tiedostonkäsittely tehdään omassa aliohjelmassa
2. Aliohjelma saa ensimmäisenä parametrinä luettavan tiedoston nimen
3. Tiedosto avataan ja suljetaan `open`/`close` -käskyillä
4. Avattu tiedosto suljetaan aina samassa pää/aliohjelmassa, jossa se on avattu
5. Jos tiedostossa on kirjainmerkkejä, tulee avatessa käyttää UTF-8 koodausta. Pelkän numeerisen datan yhteydessä tätä ei käytetä
6. Tiedosto luetaan riveittäin `readline`-käskyllä
7. Mahdollinen otsikkorivi ohitetaan erillisellä `readline`-käskyllä ennen toistorakenteessa olevaa datarivien lukemista
8. Tiedosto päättyy aina tyhjään riviin, joka on tiedoston lukemisen lopetusehto
9. Tiedostosta luetun rivin lopusta tulee poistaa rivinvaihtomerkki
10. Kirjoitettaessa tiedosto avataan kirjoitustilassa, `"w"`
11. Tiedosto kirjoitetaan `write`-käskyllä, kirjoitettava merkkijono kannattaa usein muodostaa erillisenä käskynä ennen kirjoittamista
12. Jäsenfunktioita kutsuttaessa sulkuihin (ja) laitetaan lähtevät parametrit. Jos parametrejä ei ole, sulut jätetään tyhjiksi, esim. `lueTiedosto("L06T1D1.txt")` ja `Tiedosto.close()`
13. Tiedostonkäsittelyn standardifraaseja ovat mm. seuraavat:
    a. `"Tiedosto 'tiedostoNimi' luettu."`
    b. `"Tiedosto 'tiedostoNimi' luettu ja tulostettu."`
    c. `"Tiedosto 'tiedostoNimi' kirjoitettu."`
14. Tässä oppaassa ei käytetä f-string ja with-open -rakenteita.

Kokoava esimerkki

Esimerkissä 6.15 näkyy luvun kokoava esimerkki eli tiedostonkäsittelyyn keskittyvä valikkopohjainen ohjelma. Tässä ohjelmassa näkyy mm. muotoiltu tulostus, useiden tietojen kysyminen ennen kuin ne kaikki tallennetaan tiedostoon kerralla yhdelle riville ja virheenesto eli käyttäjä ei voi lisätä tietoja tiedostoon ennen kuin ne on kysytty käyttäjältä. Lisäksi tiedostoa luettaessa rivit pilkotaan kenttäerottimesta eri tietoalkioiksi, jotka sijoitetaan muuttujien arvoiksi.

Esimerkki 6.15 Kokoava esimerkki

```
def valikko():
    print("Valitse haluamasi toiminto:")
    print("1) Kysy tiedot")
    print("2) Lisää tiedot tiedostoon")
    print("3) Tulosta tiedosto")
    print("0) Lopeta")
    Valinta = int(input("Valintasi: "))
    return Valinta

def kysyNimi():
    Nimi = input("Anna lisättävä nimi: ")
    return Nimi

def kysyIka():
    Ika = int(input("Anna ikä: "))
    return Ika

def lisaa(TiedostoNimi, Nimi, Ika):
    Tiedosto = open(TiedostoNimi, 'a', encoding="UTF-8")
    Tiedot = "{0};{1}\n".format(Nimi, Ika)
    Tiedosto.write(Tiedot)
    Tiedosto.close()
    return None

def tulosta(TiedostoNimi):
    Nimi = ""
    Ika = ""
    Tiedosto = open(TiedostoNimi, 'r', encoding="UTF-8")
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        Tietoalkio = ""
        for i in range(len(Rivi)):
            if (Rivi[i] == ";"):
                Nimi = Tietoalkio
                Tietoalkio = ""
            elif (Rivi[i] != "\n"):
                Tietoalkio = Tietoalkio + Rivi[i]
        Ika = int(Tietoalkio)
        print("Nimi on '{0:s}' ja ikä on {1:d}.".format(Nimi, Ika))
        Rivi = Tiedosto.readline()
    Tiedosto.close()
    return None
```

```
def paaohjelma():
    TiedostoNimi = "L06E14T1.txt"
    Valinta = 1
    TiedotKysytty = False
    while (Valinta != 0):
        Valinta = valikko()
        if (Valinta == 1):
            Nimi = kysyNimi()
            Ika = kysyIka()
            TiedotKysytty = True
        elif (Valinta == 2):
            if (TiedotKysytty == True):
                lisaa(TiedostoNimi, Nimi, Ika)
            else:
                print("Kysy tiedot ennen tiedostoon lisäämistä.")
        elif (Valinta == 3):
            tulosta(TiedostoNimi)
        elif (Valinta == 0):
            print("Lopetetaan.")
        else:
            print("Tuntematon valinta, yritä uudestaan.")
        print()
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
#############################################################################
# eof
```

Tuloste

```
Valitse haluamasi toiminto:
1) Kysy tiedot
2) Lisää tiedot tiedostoon
3) Tulosta tiedosto
0) Lopeta
Valintasi: 2
Kysy tiedot ennen tiedostoon lisäämistä.

Valitse haluamasi toiminto:
1) Kysy tiedot
2) Lisää tiedot tiedostoon
3) Tulosta tiedosto
0) Lopeta
Valintasi: 1
Anna lisättävä nimi: Ville
Anna ikä: 5

Valitse haluamasi toiminto:
1) Kysy tiedot
2) Lisää tiedot tiedostoon
3) Tulosta tiedosto
0) Lopeta
Valintasi: 2

Valitse haluamasi toiminto:
1) Kysy tiedot
2) Lisää tiedot tiedostoon
3) Tulosta tiedosto
0) Lopeta
Valintasi: 3
Nimi on 'Sanna' ja ikä on 22.
Nimi on 'Ville' ja ikä on 5.
```

```
Valitse haluamasi toiminto:
1) Kysy tiedot
2) Lisää tiedot tiedostoon
3) Tulosta tiedosto
0) Lopeta
Valintasi: 0
Lopetetaan.

Kiitos ohjelman käytöstä.
```