Python-kurssin oppikirja (nimi työn alla)

Mikko Tampio

9. marraskuuta 2017

Sisältö

Luku 1

Pythonin asennus ja tekstin tulostaminen

1.1 Yleistietoa Pythonista

Python on yleiskäyttöinen ohjelmointikieli, johon tämä teos keskittyy. Aloittelijoille kieltä suositellaan sen yksinkertaisuuden ja käyttäjäystävällisyyden vuoksi; ammattikäytössä Pythoniin törmää usein tieteellisessä tutkimuksessa, mutta sillä on mahdollista myös esimerkiksi palvelinohjelmointi ja peliohjelmointi. Ominaisuuksiltaan Python on lähellä 2010-luvun muita käytetyimpiä ohjelmointikieliä (mm. Java, C, C++, JavaScript), minkä vuoksi sen parissa oppii varmasti hyödyllisiä taitoja, jotka helpottavat muihin kieliin siirtymistä.

Ensimmäisen version Pythonista julkaisi Guido van Rossum vuonna 1991, ja nykyisin sen kehityksestä vastaa Python Software Foundation (`https://www.python.org/`). Säätiön sivuilta löytyy kattava englanninkielinen Python-opas (`https://docs.python.org/3/tutorial/`) sekä Pythonin dokumentaatio (`https://docs.python.org/3/`) eli yksityiskohtainen kuvaus kaikista Pythonin sisäänrakennetuista ominaisuuksista (ymmärtäminen vaatii perustiedot Pythonista).

1.2 Tästä kirjasta

Tämä hyvin keskeneräinen oppikirja on tarkoitettu lukion kurssimateriaaliksi ohjelmointia vähän tai ei lainkaan harrastaneille. Pyrin Pythonin alkeiden opettamisen lisäksi yleissivistämään lukijaa tietojenkäsittelytieteen maailmasta; tätä varten käytän muutamia käsitteitä, joita ei erikseen tarvitse opetella, jos se ei mielekkäältä tunnu. Sanasto kirjan lopussa toivottavasti auttaa, jos jonkin sanan merkitys on epäselvä.

Oppimisen tukena kannattaa käyttää (tai on pakko käyttää, jos kirja valmistuu hitaammin kuin opiskelet) tässä luvussa listattuja muita Python-

oppaita.

Kirjan tehtävät ovat pääasiallisesti peräisin Helsingin yliopiston Python-kurssimateriaalista (`https://www.cs.helsinki.fi/group/linkki/materiaali/python-perusteet/materiaali.html`) sekä Ohjelmointiputkan Python-oppaasta (`https://www.ohjelmointiputka.net/oppaat/opas.php?tunnus=python_01`). Tarkemmat lähdetiedot on merkitty tehtäviin.

1.3 Python 2 vai Python 3?

Python-ohjelmointia aloittava voi törmätä yllättävään ongelmaan: edes oppaan ensimmäinen, yhden rivin esimerkkikoodipätkä ei toimi. Kyse on siitä, että Pythonista on yhä käytössä useita eri versioita: vanhempi Python 2 ja uudempi Python 3.

Kirjoitushetkellä Python 3:n julkaisuhetkestä alkaa olla jo kymmenisen vuotta, mikä on valtavan pitkä aika tietotekniikan maailmassa. Erinäisistä syistä (kuten siitä, että useat kirjastot toimivat yhä vain Python 2:lla) Python-ohjelmoijat ovat kuitenkin vaihtaneet uudempaan versioon melko hitaalla tahdilla, eikä ole epätodennäköistä, että uudetkin Python-oppaat opettavat vanhaa Python 2:ta.

Totuus on kuitenkin se, että aloittelijan ei ole mitään syytä olla käyttämättä Python 3:a. Se on ainut Python-versio, johon enää tulee uusia ominaisuuksia ja bugikorjauksia – Python 2:n aktiivinen kehitys loppui jo viisi vuotta sitten. Suomenkielistä ohjelmoijaa ilahduttaa lisäksi se, että ääkköset toimivat Python 3:ssa ilman sen erityisempää säätämistä. Siispä tämä kirja opettaa asiat Python 3 -tavalla; jos näet virheilmoituksia, kokeile muuttaa koodistasi tällaiset rivit

```
1 print "tekstin tulostaminen"
```

tällaisiksi

```
1 print("tekstin tulostaminen")
```

Toinen usein esiin tuleva ero on, että `raw_input()`-funktion nimi on Python 3:ssa pelkkä `input()`.

Jos tietokoneessasi on jo asennettuna Python, varmista, että kyseessä on uudempi versio. Useissa Linux-pohjaisissa käyttöjärjestelmissä on molemmat; jos `python`-komento avaa Python 2:n, kokeile komentoa `python3` (sama pätee myös kaikkiin Python-työkaluihin).

1.4 Ympäristön asentaminen

Pythonin voi ladata osoitteesta `https://www.python.org/downloads/` (muista valita Python 3!). Windowsilla ja Macilla mukana tulee IDLE-niminen

ohjelma, jonka avulla Python-koodia voi kirjoittaa ja suorittaa kätevästi.

Linux-pohjaisilla käyttöjärjestelmillä jokin versio Pythonista on hyvin todennäköisesti jo asennettu. Jos näin ei ole (tai koneella on Python 2), Python tulee asentaa omaa pakettienhallintaohjelmaa käyttäen; lisäksi IDLE-ohjelma todennäköisesti puuttuu. Esimerkiksi Ubuntussa ne asennetaan syöttämällä konsoliin seuraavat komennot:

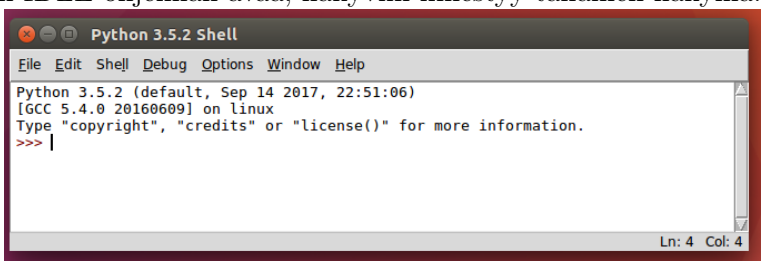
```
$ sudo apt install python3
$ sudo apt install idle3
```

Lyhyt huomio notaatiosta: merkkiä $ komennon alussa ei syötetä konsoliin, vaan sitä käytetään tavanomaisesti erottamaan ohjeissa komennot ja se, mitä komennot tulostavat konsoliin. Asentaakseen Pythonin käyttäjä siis kirjoittaa `sudo apt install python3`

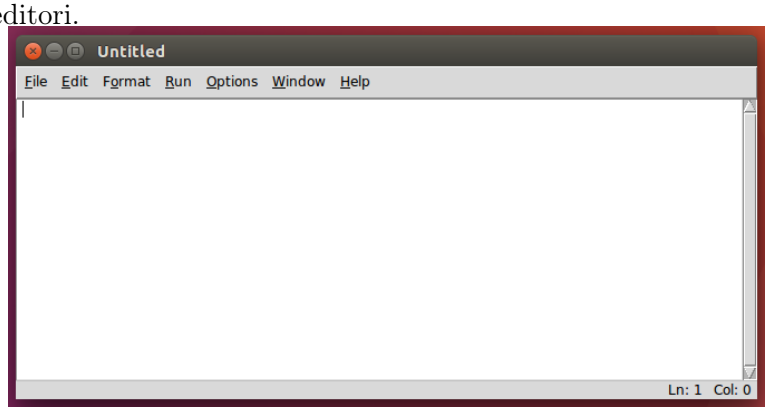
Pythonin asennukseen liittyviin ongelmiin löytyy todennäköisesti vastaus Googlesta.

1.5 Lyhyt IDLE-esittely

Kun IDLE-ohjelman avaa, näkyviin ilmestyy tällainen näkymä.



Kyseessä on Python-komentotulkki – ohjelma, joka suorittaa välittömästi siihen syötetyn Python-koodin. Jos niin haluaa, kaikki Python-ohjelmansa voi syöttää rivi riviltä komentotulkkiin, mutta kätevämpää on pitää ne erillisissä tiedostoissa. Valitsemalla `File -> New File` avautuu IDLE:n tiedostoeditori.



Kun Python-ohjelmansa on kirjoittanut IDLE:n koodieditorilla, sen voi suorittaa painamalla `F5`.

1.6 Pythonin käyttäminen komentoriviltä

Aloittelija pärjää hyvin graafisilla työkaluilla, mutta halutessaan kaiken tässä kirjassa esitetyn voi tehdä Pythonin komentorivityökaluilla. Konsolin saa auki Windowsilla painamalla `Windows + R` ja kirjoittamalla `cmd`, Unix-pohjaisilla käyttöjärjestelmillä (mm. OSX ja eri Linuxit) yleensä painamalla `CTRL + ALT + T`.

Alla on muistin virkistämiseksi taulukoituna peruskomennot, joiden avulla esimerkkien suorittamisen pitäisi onnistua.

Windows-komento	**Unix-komento**	**Selitys**
`cd` *kansio*	`cd` *kansio*	Siirtyy annettuun kansioon
`dir`	`ls`	Tulostaa nykyisen kansion sisältämät tiedostot
`help` *komento*	`man` *komento*	Näyttää lisätietoja annetusta komennosta
`python`	`python` tai `python3`	Avaa interaktiivisen Python-komentotulkin
`python` *tiedosto*	`python` *tiedosto* tai `python3` *tiedosto*	Suorittaa annetun Python-tiedoston

Seuraa lyhyt esimerkkisessio, jonka aikana Unix-pohjaista käyttöjärjestelmää käyttävä henkilö siirtyy `python`-kansioon tiedostojärjestelmässään, listaa sen sisältämät tiedostot ja suorittaa `kiva_ohjelma.py`-nimisen tiedoston, joka tulostaa ruudulle viestin `Se toimii!`.

```
$ cd python
$ ls
kiva_ohjelma.py
toinen_ohjelma.py
oppikirja.pdf
$ python kiva_ohjelma.py
Se toimii!
```

Kuten edellisessä esimerkissä, $ komentojen alussa ei ole osa komentoa vaan erottaa komennot ja niiden tulostaman tekstin.

Aihetta voisi käsitellä enemmänkin, mutta tässä vaiheessa Python-opintoja se ei ole kovin mielekästä. Konsolin käytöstä innostunut tai ongelmia kohdannut lukija voi turvautua Googleen.

1.7 Ensimmäinen ohjelma

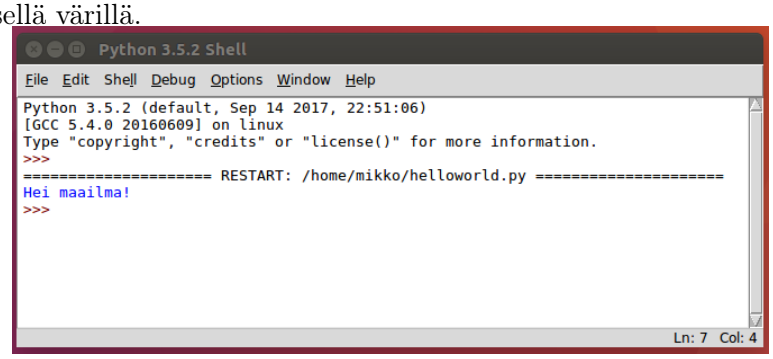
Ensimmäisistä ohjelmista klassisin on yksinkertainen sovellus, joka tulostaa ruudulle tekstin `Hei maailma!`

Alla on esitetty Hei maailma -ohjelman Python-toteutus. Kirjoita se IDLE:n avulla tiedostoon ja suorita painamalla `F5`.

Esimerkki 1.1: Hei maailma!

```
1 print("Hei maailma!")
```

Jos onnistuit, teksti `Hei maailma!` tulostui IDLE:n komentotulkkiin sinisellä värillä.



Vastedes sitä, mitä Python-koodi tulostaa ruudulle, merkitään tässä kirjassa näin.

```
Hei maailma!
```

Jos saat virheilmoituksen, palaa ensin tämän kirjan selostukseen Python 2:n ja Python 3:n eroista ja varmista, että sinulla on asennettuna Python 3. Vika voi olla myös koodissa itsessään – tarkista, että kopioit esimerkkiohjelman merkilleen oikein. Aloitteleva ohjelmoija huomaa nopeasti, että toisin kuin ihminen, tietokone ei yritä arvailla, mitä käyttäjä on voinut tarkoittaa: jos vaikkapa viimeisen sulun unohtaa, on seurauksena virhe, vaikka onkin ilmiselvää, mitä ohjelmoija on halunnut tehdä.

Kun ohjelman saa toimimaan, koittaa aika leikkiä. Muuta tulostettavaa tekstiä muuttamalla lainausmerkkien sisältöä tai kokeile laittaa useampi tulostus peräkkäin kirjoittamalla useita rivejä, joissa on jokaisessa `print()` ja sulkujen sisällä haluttu teksti merkkijonojen välissä. Tehtävissä on lisäehdotuksia.

Vaikka luvun ainut esimerkki onkin yksinkertainen, se täyttää Python-ohjelman määritelmän. Python-ohjelma koostuu (yksinkertaistetusti) lauseista. Ohjelmassamme on yksi ainoa lause: `print("Hei maailma!")`. Se on tarkemmin sanottuna *funktiokutsu*, jossa funktiolle `print` annetaan *argumenttina* teksti `Hei maailma!`, joka on tarkennettuna merkkijono. Myöhemmin

tutustutaan toisiin funktioihin, joilla voimme tehdä muutakin kuin vain tulostaa tekstiä.

Funktioihin ja merkkijonoihin palataan tulevissa luvuissa.

1.8 Useiden rivien tulostaminen

Edellisen osion lyhyt esimerkkiohjelma tulostaa Python-konsoliin yhden ainoan rivin tekstiä. Jos rivejä haluaa useampia, voi toimia monella tavalla; helpointa on yksinkertaisesti kirjoittaa monta `print`-lausetta, joista jokainen tulostaa oman rivinsä. Tässä havainnollistava esimerkki.

```
1 print("Ensimmäinen rivi...")
2 print("... ja toinen.")
```

Esimerkki tulostaa näytölle seuraavan tekstin:

```
Ensimmäinen rivi...
... ja toinen.
```

Muitakin tapoja on. Kenoviivaa (\) voidaan käyttää koodinvaihtomerkkinä, jolloin sen avulla voidaan kirjoittaa rivinvaihtomerkki (`\n`). Rivinvaihtomerkin kohdatessaan Python jatkaa tekstiä uudelta riviltä, joten seuraavalla esimerkillä on tismalleen sama ulostulo.

```
1 print("Ensimmäinen rivi...\n... ja toinen.")
```

Näin on tehtävä, koska Python ei salli merkkijonon jatkuvan seuraavalle riville. Tämä rajoitus kuitenkin poistuu, jos yhden lainausmerkin sijaan merkkijonon alussa ja lopussa käytetään kolmea.

```
1 print("""Ensimmäinen rivi...
2 ... ja toinen.""")
```

1.9 Kommentointi

Kommentit ovat hyödyllinen työkalu, jonka avulla koodiin voi jättää pieniä merkintöjä. Seuraava esimerkki havainnollistaa kommenttien eri käyttötarkoituksia; kommentti alkaa #-merkistä ja jatkuu rivin loppuun saakka. Python-tulkki jättää huomiotta kaiken kommentissa olevan, joten kommenttien sisältö ei vaikuta mitenkään ohjelman toimintaan, mutta joitakin käyttötarkoituksia niillä siitä huolimatta on.

Esimerkki 1.2: Kommennointi

```
1 # Koodannut Olli Ohjelmoija 4.10.2017
2 # Erkki Esimerkki korjasi bugin 6.10.2017
3
4 # Tulostaa tekstin "kissa"
5 print("kissa")
6
7 # print("koira")
```

Ensinnäkin kommenteilla voi jättää koodiin tietoja siitä, minkälaisia muutoksia eri henkilöt ovat siihen tehneet. Nykyisin tämä hoidetaan yleensä erillisellä versiohallintaohjelmalla, mutta aloittelijoiden ryhmätyössä voi olla kätevintä käyttää koodin alkuun sijoitettavia kommentteja.

Tämän lisäksi kommenteilla voidaan selittää, mitä hankalat tai muuten epäselvät pätkät koodia tekevät. Aloittelija voi toki jättää itselleen esimerkin kaltaisia kommentteja, jotka edistyneemmälle ohjelmoijalle ovat täysin turhia, mutta yleensä kannattaa miettiä kriittisesti kommenttien tarpeellisuutta – tarpeettomat kommentit voivat lisätä koodin lukemisen hankaluutta. Hyvä nyrkkisääntö on, että kommenttien pitäisi kertoa, *miksi* sen kirjoittanut ohjelmoija päätyi juuri valitsemaansa ratkaisuun; koodi voi puhua omasta puolestaan.

Esimerkin kolmas kommentti näyttää, kuinka tietyn koodirivin voi nopeasti ottaa pois käytöstä muuttamalla sen kommentiksi. Todettakoon kuitenkin, että on huonoa tyyliä jättää koodiin sadoittain pois kommentoituja rivejä – kuten turhat ja itsestäänselvät kommentit, nekin saavat koodia lukevan henkilön kiinnittämään huomionsa itseensä. Jos kommentoituja koodirivejä projektiinsa kuitenkin jättää, on hyvätapaista kirjoittaa selventävä kommentti, joka selittää, miksi rivit on otettu pois käytöstä.

1.10 Tehtäviä

1.10.1 Muuta esimerkkiohjelma 1.1 tulostamaan oma nimesi.

1.10.2 Muokkaa esimerkkiohjelmaa 1.1. Kokeile, mitä käy, jos...

(a) ... viimeisen sulun poistaa

(b) ... viimeisen lainausmerkin poistaa

(c) ... lainausmerkkien sisällä ei ole mitään

(d) ... `print`-funktion kirjoittaa väärin (vaikkapa `pirnt`)

1.10.3 Kokeile eri tapoja tulostaa useita rivejä tekstiä. Tee ainakin kaksi ohjelmaa, jotka tulostavat

```
Hei maailma!
Englanniksi se on Hello, World!
```

1.10.4 Mitä hyötyä kommenteista on, jos ne voisi poistaa, eikä ohjelma muuttuisi mitenkään?

Luku 2

Muuttujat ja tietotyypit

2.1 Muuttujien idea

Tähän asti olemme antaneet `print`-funktiolle suoraan merkkijonon. Sen lisäksi voimme käyttää aiemmin määriteltyä muuttujaa.

```
1 x = "Tulostettava teksti"
2 print(x)
```

Ohjelman tuloste on

```
Tulostettava teksti
```

Esimerkki havainnollistaa, että muuttuja viittaa sille aiemmin annettuun arvoon. Muuttuja määritellään *`muuttuja = arvo`*; kuten seuraava esimerkki näyttää, muuttuja voidaan myös määritellä uudelleen, jolloin se viittaa aina viimeiseksi määriteltyyn arvoonsa.

Esimerkki 2.1: Muuttujan määrittely ja uudelleenmäärittely

```
1 var = "muuttujan arvo"
2 print(var)
3 var = "toinen arvo"
4 print(var)
```

Tämä ohjelma tulostaa

```
muuttujan arvo
toinen arvo
```

Esimerkkien muuttujat olivat nimiltään `x` ja `var`. Muuttujilleen voi antaa nimeksi lähes mitä tahansa, mutta tärkeitä rajoituksia on kaksi.

- Muuttuja ei saa sisältää erikoismerkkejä, joita Python syntaksissaan käyttää. Esimerkiksi `muut(tuja` ja `muut"tuja` ovat virheellisiä muuttujia, joiden käyttämisestä on seurauksena syntaksivirhe.
- Muuttuja voi sisältää numeroita, mutta se ei saa alkaa sellaisella. `nimi4` on sallittu muuttuja, mutta `2kissaa` ei.
- Muuttuja ei saa olla avainsana. Avainsanoja ovat tietyt Pythonin varaamat termit, kuten `if` ja `def`.

Tämän lisäksi on kiellettyä viitata sellaiseen muuttujaan, jota ei ole määritelty. Muuttujan voi joko määritellä itse (kuten esimerkeissä tehdään) tai saada valmiiksi Pythonilta.

2.2 Tekstinsyöttö

Muuttujien todellinen hyöty käy esille, kun täytyy käsitellä arvoja, jota ei tunneta ennen koodin suorittamista. Tällaisia ovat mm. tiedostojen sisältö, kellonaika, päivämäärä ja käyttäjän syöttämä teksti. Seuraava esimerkki havainnollistaa näistä viimeistä: käyttäjän nimeä kysytään `input`-funktiolla, minkä jälkeen se säilötään muuttujaan.

Esimerkki 2.2: Tekstinsyöttö

```
1 nimi = input("Syötä nimesi: ")
2
3 print("Hei, " + nimi + "!")
```

`input`-funktiolle annetaan merkkijono, joka näytetään käyttäjälle, joka voi sitten kirjoittaa Python-konsoliin haluamansa vastauksen. Esimerkin ulostulossa käyttäjä syöttää nimekseen `sari`, ja sitten ohjelma tulostaa tervehdyksen, jossa `nimi`-muuttuja on korvattu käyttäjän syöttämällä arvolla.

```
Syötä nimesi: sari
Hei, sari!
```

Toinen uusi asia esimerkissä on se, että merkkijonoja voi yhdistää `+`-merkillä. Esimerkiksi `"kis"+"sa"` on sama asia kuin `"kissa"`.

2.3 Tietotyypit

Tähän asti on selvitty pelkillä merkkijonoilla, mutta Pythonin muihkin tyyppeihin on tarpeen tutustua. Tyyppiä voi ajatella joukkona arvoja, joilla on samanlaisia ominaisuuksia. Esimerkiksi kaikilla merkkijonoilla, kuten `"hevonen"`

ja `"veropetos"`, on se ominaisuus, että niitä voi edellä opitun mukaan yhdistellä `+`-merkillä.

Pythonin perustyyppeihin lukeutuvat merkkijono, totuusarvo ja numeeriset tyypit, joihin seuraavaksi tutustutaan. Totuusarvoja käsitellään ehtolauseiden yhteydessä. Tyyppejä on lisääkin, ja niitä voi jopa määritellä itse – lisää olio-ohjelmointia käsittelevässä luvussa.

Eri tyyppisiä arvoja voi myös muuttaa toiseen tyyppiin. Tämä on tarpeen esimerkiksi silloin, kun haluamme käsitellä käyttäjän syötettä (`input`-funktio palauttaa aina merkkijonon) lukuna. Lisää aiheesta kokonaislukuosion esimerkissä.

2.4 Merkkijonot

On aika syventää tietämystä merkkijonoista. Syntaksi niiden määrittelyyn tunnetaan jo: lainausmerkkien sisällä oleva osa koodia tulkitaan merkkijonoksi. Tässä lisää tarpeellisia ominaisuuksia, joita tarvitaan merkkijonojen kanssa työskentelyssä.

Esimerkki 2.3: Merkkijonojen ominaisuuksia

```
# Merkkijonojen yhdistäminen
"a" + "b" # 'ab'

# Merkkijonon pituuden selvittäminen
len("kissa") # merkkijonon pituus eli 5

# Merkkijonon toistaminen
"ha" * 4 # 'hahahaha'

# Isot ja pienet kirjaimet
"isolla".upper() # 'ISOLLA'
"PIENELLÄ".lower() # 'pienellä'

# Merkkijonoksi muuttaminen
# (6 on tässä kokonaisluku; lue seuraava osio)
str(6) # merkkijono '6'
```

Esimerkkilausekkeita voi testata syöttämällä ne Python-konsoliin, joka laskee niiden arvon ja tulostaa sen. Tiedostossa työskennellessään on muistettava tulostaa haluamansa operaation tulos `print`-funktiolla.

2.5 Kokonaisluvut

Kokonaislukuja voi kaikessa yksinkertaisuudessaan määritellä vain kirjoittamansa halutun luvun: `36`. Lainausmerkkejä ei tule käyttää, koska muuten

Python tulkitsee arvon merkkijonoksi. Monet funktiot, kuten äsken esitelty merkkijonon pituuden laskemiseen käytetty `len`, palauttavat kokonaisluvun.

Useissa ohjelmointikielissä kokonaisluvuilla on määrätyt minimi- ja maksimiarvot, mutta Pythonissa kiinteitä rajoja ei ole. On teoreettisesti mahdollista luoda niin iso kokonaisluku, että tietokoneen muisti loppuu kesken, mutta käytännössä laskeminen käy sitä ennen niin hitaaksi, ettei Pythonin käyttö onnistu.

Esimerkki 2.4: Kokonaislukujen ominaisuuksia

```
# Peruslaskutoimituksia
4 + 6 # 7
3 - 8 # -5
2 * 3 # 6

# Eri jakolaskut
7 / 3 # 2.3333333333333335
7 // 3 # 2

# Potenssilasku
13 ** 21 # 247064529073450392704413
```

Kuten esimerkki osoittaa, kokonaisluvuilla on samanlaisia laskutoimituksia kuin matematiikassa yleensä. Erilaisia jakolaskuoperaattoreita on kaksi: `/` tuottaa liukuluvun (katso seuraava osio) ja `//` kokonaisluvun. Kokonaislukujakolaskussa pyöristetään alaspäin; esimerkiksi `5//2` on `2`, ei `3`.

Usein aloittelijoilta unohtuva asia on se, että arvoja täytyy muuttaa toiseen tyyppiin, jos niitä haluaa käsitellä eri tavalla. Käyttäjän syöte `input`-funktiolla on merkkijono; vain merkkijonoja voi yhdistää `+`-operaattorilla. Tämän vuoksi seuraavassa esimerkissä muutamme ensin käyttäjän syötteen kokonaisluvuksi ja sitten jälleen merkkijonoksi, kun haluamme yhdistää sen toiseen merkkijonoon.

Esimerkki 2.5: Käyttäjän iän kysyminen

```
luku = int(input("Syötä ikäsi: "))

print("Vuoden päästä olet " + str(luku+1) + " vuotta vanha.")
```

Tässä ohjelman ulostulo, kun käyttäjä syöttää iäkseen 19.

```
Syötä ikäsi: 19
Vuoden päästä olet 20 vuotta vanha.
```

Ensimmäisellä rivillä kysytään käyttäjän ikää, joka muutetaan kokonaisluvuksi `int`-funktiolla, koska sitä halutaan pian käsitellä aritmeettisesti. Las-

kutoimituksen `luku+1` tulos on kokonaisluku, joten se on muutettava merkkijonoksi `str`-funktiolla, jotta sen voi yhdistää muihin merkkijonoihin.

2.6 Liukuluvut

Liukuluvut ovat ohjelmoinnissa desimaalilukujen vastine. Niiden käyttö on muuten samanlaista kuin kokonaislukujen, mutta kokonaislukuosa ja desimaaliosa erotetaan pisteellä (ei pilkulla, kuten suomen kielessä yleensä). `float`-funktiolla voi muuttaa merkkijonoja ja kokonaislukuja liukuluvuiksi.

Esimerkki 2.6: Liukulukulaskuri

```
1 luku = float(input("Syötä liukuluku: "))
2
3 print("4.6 * " + str(luku) + " = " + str(4.6 * luku))
```

Kun käyttäjä syöttää `7.8`, ohjelman ulostulo on seuraava:

```
Syötä liukuluku: 7.8
4.6 * 7.8 = 35.879999999999995
```

Tarkkaavainen lukija saattaa huomata, että laskun tulos on epätarkka. Liukulukujen ominaisuuksiin palataan matemaattista laskentaa käsittelevässä luvussa.

2.7 Tehtäviä

2.7.1 Tee ohjelma, joka kysyy käyttäjältä nimeä ja tulostaa lyhyen tarinan, jossa päähenkilön nimi on korvattu annetulla nimellä.

2.7.2 Mitkä seuraavista muuttujista ovat sallittuja? Jos et tiedä, kokeile Python-tulkissa ja perustele, miksi asia on näin.

(a) `muut63`
(b) `Kokonaisluku`
(c) `5kulmio`
(d) `kissan_nimi`
(e) `luku(käyttäjänikä)`

2.7.3 Tee ohjelma, joka kysyy käyttäjältä merkkijonoa ja tulostaa sen pituuden.

2.7.4 Tee ohjelma, joka kysyy käyttäjän nimeä ja ikää ja tulostaa ne muodossa `Hei, [nimi], [ikä] vuotta!`

2.7.5 Katso jotakin esimerkeistä, jossa arvoja muutetaan toiseen tyyppiin. Minkälaisen virheen saat, kun poistat funktiot, jolla muunnos tehdään? (kuten `str` tai `int`)

2.7.6 Tee ohjelma, joka kysyy käyttäjältä kahta lukua `a` ja `b` ja tulostaa laskutoimitukset `a+b`, `a-b`, `a*c` ja `a/b`.

2.7.7 Kokeile, mitä seuraava ohjelma tekee:

```
1 tulostus = print
2 tulostus("Tulostetaan...")
```

Mitä tämä kertoo sinulle siitä, mitä `print` ja muut funktiot ylipäätään ovat?

Luku 3

Ehtolauseet

3.1 `if`-lause yksinkertaisimmillaan

Tähänastiset ohjelmamme ovat edenneet täysin suoraviivaisesti: Python-tulkki työskentelee rivi kerrallaan, kunnes saapuu tiedoston loppuun. `if`-lauseiden avulla ohjelma voi tehdä valintoja ja toimia eri tavalla riippuen vaikkapa käyttäjän syötteestä, päivämäärästä tai mistä tahansa ehdosta.

Seuraava esimerkki esittää ehtolauseen kaikista yksinkertaisimmillaan: yksi ainoa ehto, jonka täyttyessä tulostetaan tekstiä.

```
1 salasana = input("Syötä salasana: ")
2
3 if salasana == "correct horse battery staple":
4         print("Oikein!")
```

Ehtolause alkaa `if`-avainsanalla ja sen ehdolla. `==`-operaattori tarkastaa, ovatko sen oikea ja vasen puoli yhtäsuuret – lauseke `salasana == "..."` siis testaa, täsmääkö muuttuja `salasana` annettuun arvoon. Ehdon jälkeen seuraa kaksoispiste.

`if`-lausetta kirjoittaja ohjelmoija haluaa, että ehdon ollessa tosi ohjelma suorittaa joitakin koodirivejä. Tämä onnistuu siten, että ne kirjoittaa kaksoispisteen jälkeen sisennettynä eli jonkin määrän välilyöntejä tai tabulaattoreita jälkeen. Esimerkissä `if`-lauseen sisällä on yksi ainoa lause, mutta niitä voi olla useitakin, kunhan kaikki on sisennetty tismalleen samalla tavalla.

Tätäkin monimutkaisempia ehtolauseita voi rakentaa, mutta sitä ennen on tarpeen tutustua tarkemmin siihen, mitä ehdot oikeastaan ovat.

3.2 `bool`-tyyppi

Tarkemmin sanottuna `if`-lauseen ehtona on oltava lauseke, joka on tyypiltään totuusarvo eli `bool`. Totuusarvoja on kaksi: `True` eli tosi ja `False` eli

epätosi.

Aiemmin näimme `==`-operaattorin, joka palauttaa totuusarvon. `x == y` on `True`, jos `x` ja `y` ovat yhtä suuria, mutta `False`, jos ne eivät ole. Vertailla voi muillakin operaattoreilla; tässä ne taulukoituna.

Operaattori	**Merkitys**
`x == y`	Palauttaa, ovatko `x` ja `y` yhtä suuria
`x != y`	Palauttaa, ovatko `x` ja `y` eri suuria
`x > y`	Palauttaa, onko `x` suurempi kuin `y`
`x >= y`	Palauttaa, onko `x` suurempi tai yhtä suuri kuin `y`
`x < y`	Palauttaa, onko `x` pienempi kuin `y`
`x <= y`	Palauttaa, onko `x` pienempi tai yhtä suuri kuin `y`

Neljä viimeistä vertaavat merkkijonoja aakkosjärjestyksessä; esimerkiksi `"z"` on suurempi kuin `"a"`.

Vertailuoperaattorien avulla voi muodostaa yksinkertaisia ehtoja `if`-lauseisiin. Jos haluaa tarkistaa, onko käyttäjä täysi-ikäinen, voi kirjoittaa ehtolauseen `if ika >= 18: ...` Monimutkaisemmaksi kuitenkin menee. Jos on tarpeen tarkastaa useita ehtoja samaan aikaan, voi käyttää sisäkkäitä `if`-lauseita tällä tavoin:

```
if ika >= 18:
        if nimi == "Jussi":
                ...
```

Siistimpää on kuitenkin yhdistää ehdot. Jos `a` ja `b` ovat joitakin ehtolausekkeita, Python tarjoaa seuraavat operaattorit, joilla niistä voi muodostaa uusia ehtoja.

Operaattori	**Merkitys**
`a and b`	Tosi vain silloin, jos sekä `a` että `b` ovat tosia
`a or b`	Tosi, jos `a`, `b` tai molemmat ovat tosia
`not a`	Tosi, jos `a` on epätosi

Aiemmin esitetyn ehdon voi siis yhdistää `and`-operaattorilla.

```
if ika >= 18 and nimi == "Jussi":
        ...
```

On hyvin mahdollista kirjoittaa sama ehto monella eri tavalla. Jos haluaa muodostaa ehdon, joka on tosi vain silloin, kun muuttuja `x` ei ole arvoltaan `7`, voi kirjoittaa esimerkiksi `x != 7` tai `not x == 7`. Sellaista muotoa kannattaa suosia, jota on helpoin ymmärtää – Python ei siitä välitä, kuinka monimutkaisia tai yksinkertaisia ehtoja kirjoittaa, mutta koodia lukevat

muut ihmiset välittävät.

3.3 Monimutkaisempia `if`-lauseita

Aiemmin käsittelimme vain yksinkertaisimpia `if`-lauseita, joiden sisältämät lauseet joko suoritetaan tai ei suoriteta sen mukaan, onko ehto tosi vai ei. Entä jos haluammekin tehdä jotakin muuta siinä tapauksessa, että ehto on epätosi? Äsken näkemämme `not`-operaattorin avulla se on mahdollista.

```
1 if luku == 7:
2         print("Luku on 7!")
3
4 if not luku == 7:
5         print("Luku on jotain muuta!")
```

Itsensä toistaminen on kuitenkin pahasta. Ylläoleva ohjelma on varsinainen bugipesä – on helppoa kuvitella, että käy muokkaamassa toisen `if`-lauseen ehtoa, mutta unohtaakin muokata toista, minkä seurauksena jotakin odottamatonta tapahtuu. Onneksi asian voi tehdä siistimminkin.

Esimerkki 3.1: `else`-haara

```
1 if luku == 7:
2         print("Luku on 7!")
3 else:
4         print("Luku on jotain muuta!")
```

`if`-lauseeseensa voi esimerkin mukaisesti liittää `else`-haaran, joka suoritetaan silloin ja vain silloin, kun annettu ehto on epätosi.

Sisennyksellä on väliä: `else`-avainsanan on oltava samalla sisennystasolla (siis sitä ennen on oltava sama määrä välilyöntejä) kuin sen `if`-lauseen, johon se liittyy. `else`-lohkoon liittyvien lauseiden täytyy olla sisennetty enemmän kuin itse avainsanan. Kaksoispiste `else`-avainsanan jälkeen on myös muistettava; Python hälyttää syntaksivirheestä, jos mikään `if`-lauseen anatomiassa on pielessä.

Entä jos ei haluakaan suorittaa `else`-haaraa joka kerta? Voiko sillekin asettaa oman ehtonsa? Aina olisi mahdollista laittaa sisään toinen `if`-lause.

```
1 if nimi == "Mari":
2         print("Hei Mari!")
3 else:
4         if nimi == "Jasper":
5                 print("Terve Jasper!")
6         else:
7                 print("Tuntematon henkilö.")
```

Kuten arvata saattaa, siistimpi tapa on olemassa. `elif`-avainsanalla voi liittää `if`-lauseeseen lohkoja, jotka suoritetaan, jos jokin toinen ehto on tosi.

Esimerkki 3.2: Palaute arvosanasta

```
arvosana = int(input("Syötä arvosanasi: "))

if arvosana == 10:
        print("Loistavasti tehty.")
elif arvosana >= 8:
        print("Hyvin tehty.")
elif arvosana >= 5:
        print("Kohtuullisen hyvin tehty.")
elif arvosana == 4:
        print("Et päässyt läpi.")
else:
        print("Outo arvosana.")
```

Huomaa, että `if`-lauseesta suoritetaan vain ensimmäinen haara, jonka ehto on tosi. Arvosana 9 täyttää kyllä ehdon `arvosana >= 5`, mutta koska se täyttää ensin esiintyvän ehdon `arvosana >= 8`, ohjelma tulostaa `Hyvin tehty.`

Osaatko päätellä, mitä ohjelma tulostaa arvosanalla 6? Entä 4? Entä -2?

3.4 Tehtäviä

3.4.1 Tee ohjelma, joka kysyy käyttäjältä salasanaa ja tulostaa joko `Oikein!` tai `Väärin!` siitä riippuen, oliko salasana oikea.

3.4.2 Olkoon `a`, `b`, `c` ja `d` kokonaislukuja. Muodosta `if`-lauseet, jotka tarkastavat, ovatko seuraavat ehdot tosia.

(a) `a` on suurempi kuin `b`

(b) `c` on pienempi tai yhtä suuri kuin `d`

(c) `a` ei ole yhtä suuri kuin `b`

(d) `a` on suurempi kuin `b` ja `c` on suurempi kuin `d`

(e) `a` on `7` tai `b` on `3`

3.4.3 Tee ohjelma, joka kysyy kahta lukua ja kertoo, oliko ensimmäiseksi vai toiseksi syötetty suurempi.

3.4.4 **Karkausvuositarkistin.** Ohjelmointitehtävien klassikko; tee ohjelma, joka kysyy käyttäjältä vuosilukua ja kertoo, onko se karkausvuosi vai ei. Karkausvuosia ovat sellaiset vuodet, jotka ovat neljällä jaollisia, mutta jos vuosi on myös jaollinen sadalla, se on karkausvuosi vain,

jos se on jaollinen myös luvulla 400. Esimerkiksi 2014 ja 2400 ovat karkausvuosia, mutta 2007 ja 1900 eivät ole.

Jaollisuutta voit tarkastella `%`-operaattorilla, joka palauttaa jakojäännöksen; esimerkiksi `7 % 3` on `1`. Jos `x`:llä jaettaessa jakojäännös on `0`, luku on jaollinen `x`:llä.

3.4.5 Milloin seuraavat ehtolausekkeet ovat tosia? `x` ja `y` ovat kokonaislukuja.

(a) `x < 10 or y < 10`

(b) `x > y and x != 8`

(c) `not x != 0 and x == 0` (kuinka voisit ilmaista tämän yksinkertaisemmin?)

3.4.6 **Nelilaskuri.** Tee ohjelma, joka kysyy käyttäjältä kahta lukua ja laskutoimitusta ja tulostaa sitten laskun tuloksineen. Ohjelma voi toimia esimerkiksi näin:

```
Syötä ensimmäinen luku: 4
Syötä toinen luku: 5
Syötä laskutoimitus (+, -, *, /): +

4 + 5 = 9
```

Luku 4

Toisto

4.1 Toisto `while`-silmukalla

Nyt esiteltävä `while`-lause muistuttaa hyvinkin paljon `if`-lausetta, joten lukijan kannattaa kerrata edellisen luvun ehtolauseita käsittelevää osiota, jos tarvetta siihen on. Itse asiassa `while`-lause on syntaksiltaan lähes täydellinen `if`-lauseen kopio; ainut ero on se, että käytettävä avainsana on `while`, ei `if`. Samaan tapaan tarvitaan kaksoispiste ehdon jälkeen ja kasvava sisennys sisälohkolle.

Mitä eroja sitten on lauseiden semantiikassa? Jos `if`-lause suorittaa sisältämänsä lauseet siitä riippuen, onko sen ehto tosi, `while`-lause toistaa niitä niin kauan. Esimerkki auttanee.

Esimerkki 4.1: Lukujen iteroiminen `while`-silmukalla

```
1 x = 0
2 while x < 10:
3         print(x)
4         x = x + 1
```

Esimerkin ehto `x < 10` on tosi, jos `x` on alle 10. Silmukan sisällä rivillä 4 `x`:n arvoa kasvatetaan yhdellä, joten lopulta ehdosta tulee epätosi, kun `x` saa arvon 10, ja silmukasta poistutaan. `x`:n arvo tulostetaan silmukan kierroksella, joten ulostulo näyttää tältä:

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

Ehdoksi voi `while`-silmukalle antaa mitä tahansa. Yksi hyödyllinen erikoistapaus on ääretön silmukka, jonka avulla ohjelmansa toimintoa voi toistaa, kunnes käyttäjä sen sulkee. Sen voi muodostaa yksinkertaisesti antamalla `while`-lauseelle ehdoksi totuusarvon `True`.

```
while True:
        # varsinainen ohjelma
```

Ehto on aina tosi, joten ohjelma toistaa sisälohkoaan ikuisesti. On kuitenkin mahdollista myös poistua silmukasta `break`-avainsanan avulla.

Esimerkki 4.2: `break`-avainsana

```
while True:
        salasana = input("Syötä salasana: ")
        if salasana == "kissa":
                break

print("Pääsit sisään.")
```

Kun käyttäjä syöttää oikean salasanan, kaikessa yksinkertaisuudessaan `break` poistuu silmukasta, jolloin koodin suoritus jatkuu eteenpäin ja ruudulle tulostuu teksti `Pääsit sisään!`.

On hyvä huomata, että ohjelman voisi toteuttaa myös ilman `break`-ominaisuutta tarkistamalla `while`-silmukan ehdossa, milloin salasana on oikea. Periaatteessa `break` ei mahdollistakaan mitään uutta, mutta joissain tapauksissa sillä voi tulla siistimpää koodia. Omaa harkintaa kannattaa käyttää.

4.2 `for`-lause

Esimerkissä 4.1 nähtin yksi tapa käydä läpi kaikki tietyn lukuvälin luvut. Yksi siistin ja selkeän ohjelmoinnin säännöistä on se, että kuhunkin käyttötarkoitukseen käytetään siihen parhaiten sopivaa työkalua, ja sellainen lu-

kuvälin läpikäymiseen löytyy: `for`-lause. Havainnollistamisen vuoksi tässä aiempi esimerkki toteutettuna `for`-silmukalla.

Esimerkki 4.3: Lukujen iteroiminen `for`-silmukalla

```
1 for x in range(10):
2         print(x)
```

Koodirivien määrä puolittui – `for`:in käyttö selvästi kannattaa, mikäli välittää koodinsa siisteydestä. Sen toinen hyvä puoli on se, että kokeneelle Python-ohjelmoijalle `for` viestii välittömästi, mikä koodipätkän tarkoitus todennäköisesti on. `while`-lauseen käyttötarkoitukset ovat sen verran laajat, että koodia lukeva ei välttämättä heti ymmärrä, mikä on sen tarkoitus.

Kuinka `for`-silmukat siis toimivat? `for`-avainsanan jälkeen tulee muuttuja, johon silmukan nykyinen arvo sijoitetaan joka iteraatiolla. Sitten seuraa `in`-avainsana ja jokin lauseke, jota käydään läpi. Tässä luvussa näytetään muutamia esimerkkejä siitä, mitä voi käydä läpi `for`-silmukalla, ja lisää tulee myöhemmin tietorakenteista puhuttaessa.

`range`-funktio palauttaa siis jonkin lukuvälin, jonka voi iteroida läpi `for`-lauseessa. Sitä voidaan käyttää useilla tavoilla.

Käyttötapa	**Merkitys**
`range(x)`	Luvut välillä $[0, x-1]$
`range(a, b)`	Luvut välillä $[a, b-1]$
`range(a, b, c)`	Joka `c`:s luku väliltä $[a, b-1]$

Kaikissa tapauksissa lukualueen jälkimmäinen raja ei sisälly siihen, mutta ensimmäinen sisältyy. `range(3)` on luvut 0, 1, ja 2; `range(4, 7)` on luvut 4, 5, 6. Aivan viimeisimmän tavan käyttää `range`-funktiota `c`-parametri voi olla vaikeaselkoinen, mutta esimerkki voi auttaa.

```
1 for i in range(0, 10, 3):
2         print(i)
```

Ohjelman ulostulo näyttää tältä:

```
0
3
6
9
```

4.3 Käänteisesti iteroiminen

Entä jos haluamme käydä läpi jonkin lukuvälin väärin päin? Matematiikkaa voi hyödyntää: kun vähentää nykyisen luvun maksimiarvosta, tulee iteroineeksi käänteisesti. Tätä varten on kuitenkin olemassa hyödyllinen `reversed`-

funktio, jota käyttämällä tekee selvemmäksi sen, mitä koodi itse asiassa yrittää tehdä. Esimerkiksi seuraavanlainen esimerkki

```
for i in reversed(range(5)):
        print(i)
```

tulostaa

```
4
3
2
1
0
```

4.4 Merkkijonon iteroiminen

`for`-silmukassa voi käyttää `range`-funktion sijaan myös merkkijonoja, jolloin jokainen merkki käydään läpi. Tässä esimerkki, joka hyödyntää myös edellisessä osiossa esiteltyä `reversed`-funktiota.

Esimerkki 4.4: Merkit takaperin

```
merkkijono = input("Syötä merkkijono: ")
uusijono = ""

for merkki in reversed(merkkijono):
        uusijono = uusijono + merkki

print(uusijono)
```

Jos käyttäjä syöttää `kissa`, tulostuu `assik`. Merkkijonon läpi iteroiminen on hyödyllistä silloin, kun haluaa tehdä merkkijonolle merkistä riippuvia operaatioita: poistaa kaikki a-kirjaimet, muuttaa joka toinen merkki isoksi kirjaimeksi, laskea vokaalit...

4.5 Tehtäviä

4.5.1 Valitse jokin vanha tehtävä tai itse tekemäsi ohjelma, joka kysyy käyttäjältä syötettä ja tekee jotakin sen mukaisesti. Muokkaa ohjelma sellaiseksi, että se toistaa itseään ikuisesti `while`-silmukan avulla.

4.5.2 Kirjoita seuraava ohjelma uudestaan käyttäen `while`-silmukkaa.

```
for luku in range(0, 10, 2):
        print(luku)
```

4.5.3 Tee ohjelma, joka laskee käyttäjän antaman merkkijonon vokaalien ja konsonanttien määrän.

4.5.4 Kirjoita seuraava ohjelma tiiviimpään ja siistimpään muotoon `for`-silmukan avulla.

```
luku = 0
while luku < 20:
        print("Luvun " + str(luku) + " neliö on " +
    ↪  str(luku * luku))
        luku = luku + 1
```

4.5.5 Tee ohjelma, joka kysyy käyttäjältä luvun ja toistaa niin monta kertaa viestin `Terve` *n*`. kerran!`, missä `n` kasvaa joka iteraatiolla.

4.5.6 Tee ohjelma, joka kysyy käyttäjältä luvun ja kertoo, monennella termillään summa $1^3 + 2^3 + 3^3 ...$ ylittää sen.

Luku 5

Funktiot

5.1 Funktiokutsu

Funktio on monikäyttöinen työkalu. Sillä voi vähentää toistoa, selkeyttää koodia tai käyttää ominaisuuksia, joita Pythonissa ei muuten ole. Pian opimme määrittelemään omia funktioitamme, jotka toimivat aivan kuten Pythonissa valmiiksi määritellyt. Ensin on tarpeen syventää ja täsmällistää tietämystä siitä, kuinka funktioita käytetään – siis millainen on funktiokutsu.

Tarkka syntaksi funktiokutsulle on tämä:

`funktio``(`*`arg1, arg2, ... , arg3`*`)`

Ensin on funktion nimi, sitten sulkujen sisällä lista argumenteista eli funktiolle annetuista arvoista pilkuilla erotettuna. Argumentteja on oltava sopiva määrä; riippuu funktiosta, minkälaisia yhdistelmiä se hyväksyy. Esimerkiksi hyvin tutulle `print`-funktiolle voi yhden merkkijonon lisäksi antaa minkä tahansa määrän mitä tahansa arvoja, ja se tulostaa ne välilyönneillä erotettuna.

```
1 print(1)
2 print(1, 2)
3 print(1, 2, 3)
```

Ulostulo näyttää seuraavalta:

```
1
1 2
1 2 3
```

Jos argumentteja antaa väärän määrän, seuraa syntaksivirhe. Esimerkiksi funktiokutsu `len("merkkijono", "toinen)` saa aikaan virheen `TypeError: len() takes exactly one argument (2 given)`.

Funktiokutsun arvo on funktion paluuarvo; esimerkiksi lausekkeen `int("53")` arvo on `53`. Kaikilla funktioilla on paluuarvo, mutta jotkin – kuten `print` –

palauttavat erityisen arvon `None`, jolloin sanomme, ettei paluuarvoa ole. Jos vaikkapa kirjoitamme `x = print("Hello World!")`, `x` saa arvon `None`. Tässä on enemmän järkeä, kun on oppinut määrittelemään omia funktioitaan.

5.2 Funktioiden määritteleminen

Otetaan esimerkiksi yksinkertainen funktio ja tutkitaan sen avulla funktioiden määrittelemisen syntaksia.

Esimerkki 5.1: Identiteettifunktio

```
1 def identiteetti(argumentti):
2         return argumentti
```

Identiteettifunktio on matematiikassa ja tietojenkäsittelytieteessä usein käytetty funktio, joka palauttaa sille annetun argumentin. Kun funktio on määritelty, sitä voisi kutsua kirjoittamalla vaikkapa `identiteetti(5)`, jolloin funktiokutsun arvo olisi `5`, tai `identiteetti("aaa")`, jolloin arvo olisi `"aaa"`.

Funktion määrittely alkaa `def`-avainsanalla. Sen jälkeen kirjoitetaan funktion nimi ja lista argumenteista suluissa. Jos haluamme useita argumentteja, ne erotetaan funktiokutsujen tavoin pilkulla. Lopuksi seuraa kaksoispiste.

Tämän lisäksi jokaisella funktiolla on vartalo, joka on vaikkapa `if`-lauseiden tavoin sisennetty lohko. Funktion sisälle voi kirjoittaa lähes samaa koodia kuin ulkopuolellekin; yksi ero on se, että funktioiden sisällä voi käyttää `return`-avainsanaa, joka palauttaa halutun arvon funktiosta. Lisäksi aiemmin määritellyt argumentit asetetaan samannimisiin muuttujiin. Identiteettifunktio palauttaa ainoan argumenttinsa, joka on funktion määrittelyn mukaisesti muuttujassa `argumentti`.

Monimutkaisempi esimerkki auttaa hahmottamaan, mitä funktiokutsussa tapahtuu. Tarkastellaan funktiota, joka ottaa kaksi argumenttia ja laskee ne yhteen.

```
1 def laskeyhteen(a, b):
2         return a + b
3
4 print(laskeyhteen(3, 7))
5 print(laskeyhteen("a", "b"))
```

```
10
ab
```

Funktio `laskeyhteen` ottaa kaksi argumenttia, joita sen sisällä merkitään `a` ja `b`. `return`-avainsanaa käyttäen se palauttaa paluuarvonaan `a + b`.

Alempana `laskeyhteen`-funktiota kutsutaan arvoilla `3` ja `7`. Funktion sisällä muuttujaan `a` asetetaan siis arvo `3` ja muuttujaan `b` arvo `7`.

Sitten funktiota kutsutaan toisen kerran, nyt arvoilla `"a"` ja `"b"`. Paluuarvon lauseke `a + b` muuttuu siis lausekkeeksi `"a"+ "b"`, mikä on Pythonissa sallittua, ja funktio palauttaa `"ab"`. Nimestä voi päätellä, että ohjelmoija ei ehkä tarkoittanut, että funktiolle voitaisiin antaa merkkijonoja, mutta se kuitenkin toimii. Jos ohjelma ei toimi, voi olla hyväksi tarkastaa, antaako vahingossa jollekin funktiolle vääränlaisia argumentteja – Pythonissa jo määritellyt funktiot, kuten `int` tai `range`, antavat yleensä selkeän virheviestin, mutta kuten esimerkki osoittaa, omiin funktioihin voi joskus päästä vahingossa vääränlaisia arvoja.

5.3 Sivuvaikutukselliset funktiot

Tähän mennessä käsitellyt funktioesimerkit ovat olleet yksinkertaisia ja jopa tarpeettomia. Kun muistaa, että funktion sisään saa kirjoittaa mitä koodia hyvänsä, on mahdollista tehdä hyödyllisempiä funktioita, joilla on sivuvaikutuksia – ne siis tekevät muutakin kuin vain laskevat ja palauttavat paluuarvonsa. Jos vaikkapa huomaamme, että kysymme käyttäjältä lukua monta kertaa ohjelman aikana, koodia voi siisteyttää se, että tämä laitetaan omaan funktioonsa.

Esimerkki 5.2: Kokonaisluvun kysyvä funktio

```
1 def kysyluku():
2         luku = int(input("Syötä kokonaisluku: "))
3         print("Hyväksytty.")
4         return luku
5
6 a = kysyluku()
7 b = kysyluku()
8 c = kysyluku()
9 print(a * b * c)
```

Ohjelma kysyy käyttäjältä kolme lukua ja tulostaa niiden tulon. Jos käyttäjä syöttää sallitun arvon (eikä esimerkiksi `kissa`), ohjelma tulostaa kohteliaasti `Hyväksytty`. Ulostulo voi näyttää vaikkapa tältä:

```
Syötä kokonaisluku: 8
Hyväksytty.
Syötä kokonaisluku: 3
Hyväksytty.
Syötä kokonaisluku: 2
Hyväksytty.
48
```

Ohjelma on siistimpi ja helpommin ymmärrettävä kuin sellainen, jossa luvun kysyminen ja viestin tulostaminen olisi toistettu kolme kertaa. Tässä onkin yksi funktioiden tärkeä käyttötarkoitus: koodin siistiminen ja jakaminen osiin.

Varsinkin sivuvaikutuksellisissa funktioissa emme aina halua palauttaa mitään arvoa. Tyhjä `return`-lause palaa funktiosta, kuten seuraava esimerkki näyttää.

Esimerkki 5.3: Salasanankysyjäfunktio

```
def salasanasuojaus():
        while True:
                salasana = input("Syötä salasana: ")
                if salasana == "correcthorse":
                        print("Oikein!")
                        return
                else
                        print("Yritä uudelleen.")

salasanasuojaus()
# Salaisuuksia
```

Kun `salasanasuojaus`-funktiota kutsutaan, käyttäjältä kysytään `while`-silmukan avulla toistuvasti salasanaa, kunnes oikea vaihtoehto syötetään. `return`-avainsana ilman mitään arvoa yksinkertaisesti poistuu funktiosta. Kuten esimerkki näyttää, `return`-lause voi sijaita missä tahansa funktion sisällä. Niitä voi kirjoittaa useammankin tai ei yhtään: jos `return` puuttuu, funktiosta poistutaan sitten, kun sisennetty alue sen sisällä päättyy.

Kuten aiemmin ohimenevästi mainittiin, tällainenkin funktio, joka ei `return`-avainsanalla mitään palauta, palauttaa oikeasti erityisen `None`-arvon. Se ei ole erityisen jännittävä, mutta on yleissivistävänä seikkana hyvä tietää, miksi vaikkapa seuraava esimerkki

```
x = print("Hei")
print(x)
```

tulostaa tekstin `None`.

5.4 Oletusargumentit

Kuinka on mahdollista, että jotkin funktiot voivat ottaa vaihtelevan määrän argumentteja? Eräs tapa (joka ei tosin `print`-funktiota selitä) on käyttää oletusargumentteja eli argumentteja, joita funktiolle ei ole pakollista antaa, koska niillä on jo jokin oletusarvo. Esimerkiksi tutulle `int`-funktiolle voi halutessaan antaa toisen argumentin, joka kertoo, mitä lukujärjestelmää käytetään.

```
print(int("10"))
print(int("10", 2))
```

```
10
2
```

Toisessa funktiokutsussa `int`-funktiolle syötetään arvo `2`, minkä seurauksena se tulkitsee merkkijonon `"10"` 2-järjestelmässä eli binäärijärjestelmässä. Oletuksena argumentin arvo on `10`, jolloin luku tulkitaan kymmenjärjestelmää käyttäen.

Omiin funktioihinsa saa oletusargumentteja seuraavalla syntaksilla:

Esimerkki 5.4: Oletusargumentti

```
def tervehdi(tervehdys="Päivää"):
        nimi = input("Syötä nimesi: ")
        print(tervehdys + ", " + nimi + "!")

tervehdi()
tervehdi("Hei")
```

Eli funktion määrittelyn kohdalla yksinkertaisesti kirjoittaa haluamansa argumentin kohdalle yhtäsuuruusmerkin ja oletusarvon. Ohjelma toimii esimerkkisyötteillä näin:

```
Syötä nimesi: tuukka
Päivää, tuukka!
Syötä nimesi: taavi
Hei, taavi!
```

5.5 Näkyvyysalueet

Yleinen murheiden lähde funktioita käyttäessä on näkyvyysalueisiin liittyvä hämmennys. Aloitetaan esimerkillä: osaatko sanoa, mitä seuraava ohjelma tulostaa?

```
def kokeilu1(a):
        print(a)

a = 8
kokeilu1(3)
print(a)
```

Tulos, joka ei missään nimessä ole ilmiselvä, on seuraava:

```
3
8
```

Kyse on näkyvyysalueista eli siitä, missä kohdissa koodia missäkin määritellyt muuttujat näkyvät.

Miksi ensiksi tulostuu `3`? Vaikka ennen funktiokutsua määriteltiinkin `a = 8`, funktion sisällä `a`:n arvo on sille argumenttina annettu `3`. Miksi sen jälkeen tulostuu `8`? Vaikka funktion näkyvyysalueella `a`:n arvo onkin `3`, sillä ei ole enää mitään vaikutusta sen jälkeen, kun funktiosta on poistuttu.

Samasta ilmiöstä on kyse seuraavassa esimerkissä.

```
1 def kokeilu2():
2         b = 5
3
4 kokeilu2()
5 print(b)
```

Mitä ohjelma tulostaa? `5`? Ei mitään, koska tulee virhe, sillä `b`-muuttujaa ei ole määritelty. Lause `b = 5` vaikuttaa vain niin kauan, kun ollaan funktion sisällä; siispä `kokeilu2`-funktiolla ei ole mitään vaikutusta.

Entä jos haluamme välttämättä muuttaa jonkin muuttujan arvoa funktion sisältä? Se onnistuu `global`-avainsanalla.

```
1 def kokeilu3():
2         global b
3         b = 5
4
5 kokeilu3()
6 print(b)
```

Nyt tulostuu `5`, sillä `global`-avainsanalla määriteltiin, että `b` funktion sisällä viittaa ylemmän näkyvyysalueen muuttujaan. Siispä `b = 5` asetti `b`:n arvon siten, että se näkyi funktion ulkopuolellekin.

Siistin koodin nimissä on hyvä muistaa, että selkeämpää on yksinkertaisesti palauttaa haluttu arvo funktiosta. On toki tilanteita, joissa `global`-avainsanan käyttäminen on siistein ratkaisu.

5.6 Rekursio

Rekursio on hankaluudestaan huolimatta äärimmäisen hyödyllinen työkalu. Rekursiivinen funktio on sellainen funktio, joka kutsuu itseään; tämän hyödyllisyys tulee ilmi parhaiten esimerkillä. Luvun kertoma lasketaan kertomalla keskenään luku itse ja kaikki luvut sen ja nollan välissä – esimerkiksi $4! = 4 * 3 * 2 * 1 = 24$. Eräs rekursiivinen kertoman toteutus on esitetty alla.

Esimerkki 5.5: Kertoma rekursiivisena funktiona

```
def kertoma(n):
        if n == 1:
                return 1
        else:
                return n * kertoma(n-1)

luku = int(input("Syötä luku: "))
print("Kertoma on " + str(kertoma(luku)))
```

`kertoma`-funktio kutsuu itseään `if`-lauseen toisessa haarassa siten, että sille annetusta argumentista vähennetään yksi. Jos ohjelma siis alkaisi vaikkapa funktiokutsulla `kertoma(5)`, tuossa kohtaa laskettaisiin `5 * kertoma(4)`, missä `kertoma(4)` laskee `4 * kertoma(3)` ja niin edelleen.

On oleellista, että ehto `n == 1` tarkastetaan ja rekursio lopetetaan siinä tapauksessa, että näin on. Jos tarkastusta ei tee, funktio jatkaa kertoman laskemista `... * 2 * 1 * 0 * -1 * -2 * ...`, mikä paitsi on matemaattisesti väärin myös saa aikaan `RecursionError`-virheen, kun Pythonin rekursioraja ylittyy.

Voidaan tietenkin kyseenalaistaa, onko rekursio siistein tapa toteuttaa kertoman laskeva funktio. `while`-silmukkaa käyttävä ratkaisu on varmasti nopeampi – useimmissa ohjelmointikielissä rekursio on hitaampaa kuin iteraatio. On kuitenkin tapauksia, joissa rekursio on selvästi selkein ratkaisu, ja sitä varten sitä on hyvä ymmärtää.

5.7 Tehtäviä

5.7.1 Tee funktio, joka ottaa argumenttinaan jonkin luvun x ja palauttaa lausekkeen $x^2 + 1$ arvon.

5.7.2 Seuraavan koodipätkän ohjelmoija on pilkkonut sovelluksensa ansiokkaasti osiin funktioiden avulla, mutta sitä suorittaessa tulee virhe. Mistä tämä johtuu? Korjaa mielestäsi siisteimmällä tavalla.

```
def tervehdi():
        nimi = "Jenni"
        print("Hei, " + nimi + "!")

def hyvastele():
        print("Hyvästi, " + nimi + "!")

tervehdi()
hyvastele()
```

5.7.3 Edellisessä luvussa oli esimerkkinä ohjelma, joka käänsi käyttäjän syöttämän merkkijonon. Ohjelmoi funktio, joka tekee argumentilleen saman asian.

5.7.4 Fibonaccin lukujono määritellään seuraavasti: ensimmäiset luvut ovat 1 ja 1, ja seuraavat saadaan kahden edellisen summasta. Lukujono alkaa siis 1, 1, 2, 3, 5, 8, ... Toteuta rekursiivinen funktio, joka laskee argumenttinsa `n` mukaan järjestyksessä `n`:nnen Fibonaccin luvun.

Luku 6

Tietorakenteet

6.1 Listat

Tähän asti edennyt Python-ohjelmoija osaa jo aika paljon, mutta tietämyksessä on yksi vakava aukko. Kun olemme halunneet kuvata jonkinlaista tietoa, olemme yksinkertaisesti määritelleet sille muuttujan. Mutta entä jos emme tiedä, kuinka monta arvoa haluamme käsitellä? Entä jos haluamme, että lukija saa syöttää mielivaltaisen määrän lukuja, joiden keskiarvo sitten lasketaan?

Tähän on ratkaisu: listat. Lista on tietotyyppi, joka sisältää mielivaltaisen määrän muita arvoja (listan alkioita) – siispä voimme määritellä muuttujan listaksi ja sitten käsitellä sitä. Lista määritellään seuraavalla syntaksilla:

```
1 x = [1, 4, 2]
```

Muuttujaan `x` asetetaan lista, jossa on valmiina arvot 1, 4 ja 2. Tyhjää listaa voi merkitä `[]`.

Kuinka listan arvoja voi muuttaa? Kirjoittamalla `lista[indeksi]` voi käsitellä listassa `lista` kohdassa `indeksi` olevaa arvoa tavallisena muuttujana. Numerointi aloitetaan nollasta: äskeisessä esimerkkilistassa `x[0]` on arvoltaan `1`, `x[1]` on `4` ja `x[2]` on 2.

Toinen perusoperaatio listoille on se, että `for`-lauseella voi käydä niiden arvot läpi aivan kuten merkkijonon yksittäiset merkit tai lukualueen luvut. Seuraa havainnollistava esimerkki.

Esimerkki 6.1: Listojen perusominaisuudet

```
lista = [1, 2, 3, 4]

# Asetetaan toiseksi arvoksi 5
lista[1] = 5

# Tulostetaan jokainen alkio erikseen
for luku in lista:
        print(luku)

# Tulostetaan vielä koko lista
print(lista)
```

Ohjelman ulostulo on

```
1
5
3
4
[1, 5, 3, 4]
```

Tärkeä huomio on, että listojen sisällä voi olla mitä tahansa. Esimerkiksi `["kissa", "koira", "jänis"]` on lista, jonka alkiot ovat merkkijonoja. Listoja voi jopa olla sisäkkäin: `[[1, 2], [3, 4]]`

Aiemmin opittiin, että `for`-lauseella voi käydä merkkijonon merkit läpi. Merkkijonot muistuttavat listoja myös siinä mielessä, että niidenkin yksittäisiä merkkejä voi tarkastella `[]`-syntaksilla: `"kissa"[0]` on merkkijonon ensimmäinen merkki `"k"`.

Äsken käsiteltiin vain murto-osa listojen hyödyllisistä ominaisuuksista. Jos muuttujassa `lista` on jokin lista, seuraava taulukko esittää, miten eri tavoin sitä voi muokata ja tutkia.

Syntaksi	Merkitys
`lista.append(alkio)`	Lisää listaan uuden alkion
`lista.clear()`	Poistaa kaikki alkiot listasta
`lista.copy()`	Tekee listasta kopion, jota voi muokata ilman, että alkuperäinen lista muuttuu
`lista.count(alkio)`	Laskee, kuinka monta tiettyä alkiota listassa on
`lista.extend(toinen)`	Lisää listan perään kaikki toisen listan alkiot
`lista.index(alkio)`	Palauttaa, missä indeksissä tietty alkio on
`lista.insert(indeksi, alkio)`	Lisää listaan uuden alkion tiettyyn indeksiin
`lista.pop()`	Poistaa ja palauttaa listan viimeisen arvon
`lista.remove(alkio)`	Poistaa tietyn alkion listasta
`lista.reverse()`	Kääntää listan toisin päin
`lista.sort()`	Järjestää listan
`len(lista)`	Listan pituus

Listojen ominaisuuksia kannattaa opetella käyttämään tarpeen mukaan; jos jokin tuntuu hämärältä, aina voi itse kokeilla, mitä se tekee. Alla esimerkki, joka hyödyntää listojen ominaisuuksista kahta hyvin oleellista: `append()` ja `sort()`.

Esimerkki 6.2: Lukujen järjestäjä

```
lista = []

while True:
        alkio = input("Syötä uusi luku tai paina enteriä
 ↪  lopettaaksesi: ")
        if alkio == "":
                break
        else:
                lista.append(int(alkio))

lista.sort()

for luku in lista:
        print(luku)
```

Yksi ohjelman käyttökerta voi näyttää vaikkapa tältä:

```
Syötä uusi luku tai paina enteriä lopettaaksesi: 5
Syötä uusi luku tai paina enteriä lopettaaksesi: 2
Syötä uusi luku tai paina enteriä lopettaaksesi: -6
Syötä uusi luku tai paina enteriä lopettaaksesi: 1
Syötä uusi luku tai paina enteriä lopettaaksesi:
-6
1
2
5
```

Listojen lisäksi on hyvä kiinnittää huomiota ohjelman toimintatapaan: se kysyy toistuvasti syötettä, kunnes käyttäjä antaa jotain tiettyä (tyhjän merkkijonon) ja sitten siirtyy varsinaiseen datan käsittelyyn. Tämä on kätevä käytäntö, kun haluaa täyttää listan käyttäjältä pyydetyillä arvoilla.

Pieni terminologinen huomio: englannin sanat *ordered* (jossakin järjestyksessä) ja *sorted* (tietyssä järjestyksessä) suomennetaan kummatkin *järjestetty*. Ero on siinä, että lista on aina *ordered*, koska sen alkiot ovat tietyssä järjestyksessä, mutta siitä voidaan myös `sort()`-funktiolla tehdä *sorted*, jolloin sen alkiot menevät suuruusjärjestykseen. On pääteltävä kontekstista, kumpaan viitataan; *ordered*-sanalle käytetään tässä tekstissä myös vastinetta *indeksoitu*.

6.2 Joukot

Pythonin joukot muistuttavat läheisesti matematiikan joukkoja: niiden alkioilla ei ole järjestystä, eikä samassa joukossa voi olla useaa identtistä alkiota. Jälkimmäinen on hyödyllinen ominaisuus, jos tarkoituksena on esimerkiksi luetella kaikki tekstissä olleet sanat – joukko pitää automaattisesti huolen siitä, että jokainen sana listataan vain kerran.

Joukoille ei ole omaa syntaksia, vaan ne täytyy luoda `set`-funktiolla, jolle annetaan joukoksi muutettava lista. Seuraava esimerkkiohjelma pitää listaa tapahtumaan osallistuvista henkilöistä siten, että saman nimen voi syöttää useita kertoja (voidaan ajatella, että ohjelmaa käyttää useampia henkilöitä, joilla kaikilla on oma, osittain päällekäinen osallistujalista syötettäväksi ohjelmaan).

Esimerkki 6.3: Joukkoesimerkki

```
osallistujat = []

while True:
        osallistuja = input("Lisää osallistuja (tyhjä
 ↪  lopettaa): ")
        if osallistuja == "":
                break
        else:
                osallistujat.append(osallistuja)

osallistujajoukko = set(osallistujat)

for nimi in osallistujajoukko:
        print(nimi)
```

Kuten esimerkki näyttää, joukon alkiot voi listojen tapaan käydä läpi `for`-lauseella. Koska joukoilla ei ole järjestystä, tietyn indeksin alkiota ei kuitenkaan voi tarkistaa: `joukko[indeksi]` on kiellettyä syntaksia.

Joukoilla on iso joukko matematiikasta tuttuja ominaisuuksia, joihin ei tässä sen syvemmin paneuduta. Jos tuntee vaikkapa unionin ja osajoukon käsitteet, voi itse Googletella, kuinka ne Python-joukoilla toimivat.

6.3 Tuplet

Ainoa tuplejen oleellinen ero listoihin verrattuna on se, että ne ovat muuttumattomia – tuplejen alkiot eivät voi muuttua sen jälkeen, kun ne on määritelty, joten tupleilta puuttuvat listojen alkioita muuttavat ominaisuudet. Tuplejen määrittelemiseen on oma syntaksinsa, joka hyödyntää sulkumerkkejä: `("kissa", "koira")` määrittelee tuplen, jonka alkiot ovat merkkijonot `"kissa"` ja `"koira"`.

Tuple-sanalla on useita vaihtoehtoisia suomennoksia. Virallisinta lienee puhua *monikoista*, mutta myös termi *tuppeli* on vakiintuneessa käytössä. Pythonin tuple-tyyppiä voi kuvailla myös esimerkiksi *muuttumattomaksi listaksi*.

Koska tuplet ovat muuttumattomia, niiden yhdistämiseen on oma syntaksinsa. Listoilla saattoi käyttää `lista.append(toinen)`-metodia – se ei tupleille kävisi, koska se muuttaisi alkuperäistä tuplea. Sen sijaan on mahdollista käyttää `+`-merkkiä.

Esimerkki 6.4: Tuple-esimerkki

```
1 tuple1 = (1,2)
2 tuple2 = (3,4)
3
4 print("Tuplet yhdistettynä ovat: " + str(tuple1 + tuple2))
```

Esimerkki tulostaa

```
Tuplet yhdistettynä ovat: (1, 2, 3, 4)
```

Samaa syntaksia voi käyttää listoillakin, mutta alkioiden lisääminen `append`-metodilla on nopeampaa, koska kokonaista uutta listaa ei tarvitse luoda.

Käytännössä tuplet voisi aina korvata listoilla, mutta niilläkin on etunsa: muuttumattomuus viestii siitä, mitä ohjelmoija haluaa, ja estää vahingollisia virheitä. Jos siis tietää, ettei halua tietorakenteen muuttuvan, voi olla hyvä ajatus valita listojen sijasta tuplet.

6.4 Hajautustaulut

Viimeisenä tietorakenteena esitetään hajautustaulut, joiden idea saattaa vaatia tarkempaa selostusta. Mietitään ensin vaikkapa listaa `["maanantai", "tiistai", "keskiviikko", ...]`. Jokaista alkiota vastaa tietty indeksi: indeksissä `0` on merkkijono `"maanantai"`, indeksissä `1` on `"tiistai"` ja niin edelleen. Jos tiedämme indeksin, voimme saada sitä vastaavan arvon syntaksilla *`lista`*`[`*`indeksi`*`]`. Taulukoidaan mahdolliset indeksit ja niitä vastaavat alkiot.

Indeksi	**Alkio**
0	`"maanantai"`
1	`"tiistai"`
2	`"keskiviikko"`
3	`"torstai"`
4	`"perjantai"`
5	`"lauantai"`
6	`"sunnuntai"`

Voi ajatella, että `lista[3]` itse asiassa vain tarkoittaa alkiota, joka löytyy taulukon kohdasta, jossa indeksin arvo on `3`. Hajautustaulut toimivat samalla tavalla – ainoa ero on se, että indekseinä voi olla mitä tahansa arvoja. Kuvitellaan vaikkapa hajautustaulu, johon on talletettu, mitä mieltä eri viikonpäivistä olemme.

Indeksi	Alkio
`"maanantai"`	`"huono päivä"`
`"tiistai"`	`"ihan ok"`
`"keskiviikko"`	`"it's wednesday my dudes"`
`"torstai"`	`"ihan ok"`
`"perjantai"`	`"viikonloppu alkaa"`
`"lauantai"`	`"viikonloppu"`
`"sunnuntai"`	`"viikonloppu"`

Kuten listoillakin, syntaksi *`hajautustaulu`*`[`*`indeksi`*`]` antaa tietyssä indeksissä olevan alkion. Jos ylläolevaa taulukkoa vastaava hajautustaulu olisi muuttujassa `taulu`, `taulu["tiistai"]` tarkoittaisi hajautustaulun arvoa `"ihan ok"`.

Hajautustaulujen määritellään hieman aiemmin nähtyjä raskaammalla syntaksilla, mutta sen merkitys on selvä, kunhan muistaa, mitä hajautustaulut ovat – tietorakenteita, jossa jokaista indeksiä vastaa alkio. Indeksi ja alkio erotetaan kaksoispisteellä, indeksi-alkio-parit toisistaan tarvittaessa pilkuilla; lopulta kokonaisuus ympäröidään aaltosuluilla `{}`.

```
1 taulu = { 1 : "a", 2 : "b", 3 : "c" }
```

Esimerkin hajautustaulussa indeksissä `1` on arvo `"a"`, indeksissä `2` arvo `"b"` ja niin edelleen.

Indekseistä käytetään hajautustaulujen yhteydessä termiä avain (engl. *key*). Termien tekninen ero on siinä, että sisäisesti hajautustaulu käyttää numeroindeksejä, joiksi sille annetut avaimet muutetaan. Tästä ei Python-ohjelmoijan tarvitse välittää. Lisäksi hajautustaulun alkioita kutsutaan useimmiten arvoiksi (engl. *value*).

Jos haluaa käydä läpi kaiken hajautustaulussa olevan, voi käyttää kolmea eri tapaa: `taulu.keys()`, joka käy läpi avaimet, `taulu.values()`, joka käy läpi arvot sekä `taulu.items()`, joka käy läpi avain-arvo-parit tupleina. Esimerkki selkeyttää.

Esimerkki 6.5: Hajautustaulun iteroiminen

```
taulu = { "kissa": "en katt", "koira" : "en hund", "hevonen" :
    "en häst"}

# Käy läpi avaimet
print("Taulun avaimia ovat:")
for avain in taulu.keys():
        print(avain)

# Käy läpi arvot
print("Taulun arvot ovat:")
for arvo in taulu.values():
        print(arvo)

# Käy läpi avain-arvo-parit tupleina
print("Taulukon avain-arvo-parit: ")
for tuple in taulu.items():
        print(tuple)
```

Ohjelman ulostulo on seuraava:

```
Taulun avaimia ovat:
hevonen
koira
kissa
Taulun alkioita ovat:
en häst
en hund
en katt
Taulukon avain-arvo-parit:
('hevonen', 'en häst')
('koira', 'en hund')
('kissa', 'en katt')
```

6.5 Tietorakenteiden vertailua

Oikean tietorakenteen valitseminen voi helpottaa ohjelmointia dramaattisesti, joten on hyväksi muodostaa itselleen yleismielikuva siitä, mihin eri tietorakenteita useimmiten käytetään. Alla taulukko, joka tiivistää luvun tietorakenteiden perusominaisuudet.

Tietorakenne	Muuttuvuus	Indeksoitu	Erityistä
Lista	Muuttuva	Kyllä	Voidaan järjestää
Joukko	Muuttuva	Ei	Ei säilö useita identtisiä alkioita
Tuple	Muuttumaton	Kyllä	Viestii muuttumattomuudellaan
Hajautustaulu	Muuttuva	Ei	Indekseinä mielivaltaisia arvoja

Tässä luvussa ei käyty läpi kaikkia Pythonin tietorakenteista, mutta aloittelijalle nämä neljä riittävät.

6.6 Tehtäviä

6.6.1 Mitä tietorakennetta käyttäisit seuraavassa tilanteessa?

(a) Haluat säilöä ruotsi-suomi-sanakirjan

(b) Haluat säilöä käyttäjän syöttämiä lukuja, joiden keskiarvon aiot laskea

(c) Haluat säilöä aineistossa esiintyneet sanat

6.6.2 Tee ohjelma, joka kysyy käyttäjältä lukuja ja tulostaa niiden keskiarvon.

6.6.3 Tee ohjelma, joka etsii listan suurimman luvun ja tulostaa sen.

6.6.4 Luettele tuntemasi tietorakenteet, jotka...

(a) ... ovat muuttuvia

(b) ... ovat järjestettyjä

(c) ... sallivat syntaksin `tietorakenne[indeksi]`

6.6.5 Tee ohjelma, joka ensin täydentää hajautustaulun käyttäjän antamilla suomenkielisillä sanoilla ja niiden ruotsinkielisillä käännöksillä ja sitten kyselee sanoja satunnaisesti käyttäjältä.

Luku 7

Tiedostot

Tiedostot ovat hyvä työkalu moneen käyttötarkoitukseen: niillä voi tallentaa tietoja seuraavaa käyttökertaa varten, lukea kätevästi suuria määriä dataa tai antaa ohjelman ulostulon käyttäjäystävällisessä muodossa. Ensin ajaudumme kuitenkin sivupolulle, sillä on hyvä hetki opiskella virheidenhallintaa Pythonissa.

7.1 Virheidenhallinta

Virheitä tähän asti selvinnyt Python-ohjelmoija on jo nähnyt melkoisesti. Esimerkiksi yrittämällä muuttaa virheellisen merkkijonon kokonaisluvuksi (vaikkapa näin)

```
1 int("tämä ei ole kokonaisluku")
```

saa virheen, kuten

```
Traceback (most recent call last):
  File "<stdin>", line ..., in <module>
ValueError: invalid literal for int() with base 10: 'tämä ei
↪  ole kokonaisluku'
```

Aina ei ole tyydyttävää, että ohjelman suoritus yksinkertaisesti loppuu, jos jokin menee pieleen. Kuvitellaan esimerkiksi, että olemme pyytämässä käyttäjää syöttämään jonkin luvun; jos käyttäjä syöttääkin jotakin virheellistä, voisi olla järkevämpää pyytää kokeilemaan uudestaan. Tähän on keino: `try`-lause. Sen anatomia on yksinkertainen:

```
try:
        # Koodia, joka voi heittää virheen
except Virhe:
        # Virhe tuli! Reagoi tekemällä jotakin järkevää
```

Esimerkissä `Virhe`-kohtaan laitetaan sen virheen nimi, jonka sattuessa halutaan toimia jotenkin. Esimerkiksi silloin, kun `int()`-funktiolle annetaan virheellinen arvo, on seurauksena `ValueError`-virhe, kuten virheviesti kertoo. Jos haluaisimme kysyä käyttäjältä kokonaislukua, kunnes hyväksytty arvo syötetään, voitaisiin siis toimia vaikka näin:

Esimerkki 7.1: Virheidenhallinta lukua kysyttäessä

```
while True: # Toistetaan ikuisesti
        try:
                luku = int(input("Syötä kokonaisluku: "))
                break # Jos virhettä ei tullut, poistutaan
 ↪  silmukasta
        except ValueError:
                print("Yritä uudelleen")

# Tiedetään, että tässä kohtaa luku on hyväksytty
 ↪  kokonaisluku
print("Lukusi oli " + str(luku))
```

Ohjelman suoritus voisi edetä esimerkiksi näin:

```
Syötä kokonaisluku: kissa
Yritä uudelleen
Syötä kokonaisluku: 5.6
Yritä uudelleen
Syötä kokonaisluku: 2
Lukusi oli 2
```

Ohjelman rakennetta kannattaa tutkia varovaisesti. `while`-silmukan avulla lukujen kysymistä toistetaan ikuisesti. Jos syötetty merkkijono ei kelpaa, syntyy `ValueError`, jolloin siirrytään virheen hallintaan – muussa tapauksessa saavutetaan `break` ja poistutaan silmukasta.

Omia virheitäänkin voi – ja kannattaa – käyttää. Jos vaikka olisi toteuttanut oman funktionsa, joka laskee luvun kertoman, voisi virheellä ilmaista, että sen argumentiksi kelpaavat vain epänegatiiviset arvot. Tämä tehdään `raise`-avainsanalla.

Esimerkki 7.2: Virheiden käyttäminen omissa funktioissa

```
1 def kertoma(n):
2         if n < 0:
3                 raise ValueError("Luku ei voi olla
  ↪  negatiivinen")
4         # Laske tulos...
```

Jos joku yrittäisi nyt kutsua funktiota virheellisellä argumentilla, olisi tuloksena virheilmoitus.

```
Traceback (most recent call last):
  File "<stdin>", line ..., in <module>
  File "<stdin>", line 3, in kertoma
ValueError: Luku ei voi olla negatiivinen
```

Käyttäjälle kannattaa aina kirjoittaa hyödyllinen virheilmoitus, joka selventää, mikä meni pieleen.

Pythonissa on lukuisia virheitä, joihin törmää eri tilanteissa. Alla on taulukoituna yleisimpiä.

Virhetyyppi	**Merkitys**
`ValueError`	Funktiolle annettiin virheellinen arvo
`TypeError`	Jossakin käytettiin arvoa, jonka tyyppi oli väärä
`ZeroDivisionError`	Nollalla jakaminen
`IndexError`	Listasta tai tuplesta yritettiin hakea virheellistä indeksiä
`KeyError`	Hajautustaulusta yritettiin hakea virheellistä avainta
`IOError`	Tiedoston lukeminen epäonnistui (tarkempaa tietoa seuraavassa osiossa)

Lista ei ole täysin kattava; jos haluaa selvittää, mikä virhe mistäkin tulee, helpointa on tarkistaa Pythonin dokumentaatiosta tai yksinkertaisesti aiheuttaa virhe ja katsoa.

Halutessaan antaa virhen omasta funktiostaan kannattaa valita järkevä virhetyyppi. Jos annettu merkkijono on liian lyhyt, ei kannata antaa käyttäjälle `ZeroDivisionError`-virheenä.

Viimeisenä huomiona virheistä voidaan todeta, että samassa `try`-lohkossa voi tarkistaa useita eri virheitä – on vain laitettava peräkkäin useita `except`-lohkoja.

```
try:
        # Täältä tulee kamalasti virheitä
except Virhe1:
        # Tuli Virhe1-tyypin virhe
except Virhe2:
        # Tyli Virhe2-tyypin virhe
```

7.2 Tiedostojen lukeminen ja kirjoittaminen

Tiedostojen yksinkertainen hallinta onnistuu `open()`-funktiolla ja sen palauttamilla tiedostoa kuvaavilla arvoilla. Funktiolla on lukuisia argumentteja, mutta sitä voi käyttää myös antamalla vain yhden: polun, joka kuvaa haluttua tiedostoa. Polku voi olla kokonainen tiedostopolku (esimerkiksi Windowsilla `C:\user\koodaaja\kivatiedosto.txt` tai Unix-pohjaisilla järjestelmillä `/home/koodaaja/kivatiedosto.txt`) tai paikallinen tiedosto (`kivatiedosto.txt`), jolloin Python etsii sen suorituskansiosta.

`open()` palauttaa tiedosto-objektin, jonka viittaa tiedosto voidaan lukea tai siihen voidaan tehdä muutoksia. Tässä yksinkertainen esimerkki, joka tulostaa kokonaan paikallisen tiedoston `kivatiedosto.txt`.

```
tiedosto = open("kivatiedosto.txt")
print(tiedosto.read())
tiedosto.close() # Tärkeää!
```

Esimerkin ulostulo on tiedoston `kivatiedosto.txt` sisältö, mitä se sitten onkaan.

Viimeisellä rivillä tulee tärkeä huomio: käsittelemänsä tiedostot täytyy aina sulkea. Seuraavassa osiossa opitaan tapa, jolla tämän voi automatisoida, mutta siihen asti on muistettava sulkea kiltisti kaikki käyttämänsä tiedostot. Käyttöjärjestelmästä riippuen sillä, että käyttämiään tiedostoja ei sulje, voi olla erilaisia epämiellyttäviä sivuvaikutuksia – tiedostoihin tehdyt muutokset eivät ehkä oikeasti tapahdu tai muut ohjelmat eivät välttämättä voi käsitellä tiedostoa.

`open()`-funktiolle voi antaa myös toisen parametrin, joka on pakollinen, jos tiedostoa haluaa muokata. Alla on taulukoituna yleisimmät vaihtoehdot.

Käyttötapa	**Merkitys**
`open(`*`tiedosto`*`, "r")`	Tiedostoa luetaan (oletus)
`open(`*`tiedosto`*`, "w")`	Tiedosto tyhjennetään ja siihen kirjoitetaan
`open(`*`tiedosto`*`, "a")`	Tiedoston perään kirjoitetaan lisää
`open(`*`tiedosto`*`, "x")`	Tiedosto luodaan (tulee virhe, jos se on jo olemassa)

Tiedosto-objektien ominaisuuksia on puolestaan lueteltu seuraavassa taulukossa. Tiedosto on aina muistettava avata oikeassa tilassa; jos tiedostoa luetaan, sen on oltava avattu `"r"`-tilassa.

Syntaksi	**Merkitys**
Tiedoston perusominaisuuksia	
tiedosto`.name`	Tiedostonimi merkkijonona
tiedosto`.closed`	`bool`-arvo, joka kertoo, suljettiinko tiedosto jo
tiedosto`.close()`	Sulkee tiedoston
tiedosto`.mode`	Tila, jossa tiedosto avattiin
Tiedoston lukeminen	
tiedosto`.read()`	Lukee koko tiedoston
tiedosto`.readlines()`	Palauttaa tiedoston rivit listassa
Tiedostoon kirjoittaminen	
tiedosto`.write(`*sisältö*`)`	Kirjoittaa annetun merkkijonon tiedostoon
tiedosto`.writelines(`*lista*`)`	Kirjoittaa annetun listan merkkijonot tiedostoon jokainen omalla rivillään

7.3 `with`-lause

Ennen monimutkaisempia esimerkkejä on tarpeen opiskella tapa, jota käyttämällä tiedostot sulkeutuvat automaattisesti: `with`-lause. Kun sitä käyttää, unohtunut `close()` ei enää aiheuta mystisiä virheitä. Rakenteeltaan `with` on yksinkertainen.

```
1 with open(...) as tiedosto:
2         # Käsitellään tiedostoa
3 # Kun with-lohkosta poistutaan, tiedosto suljetaan
  ↪  automaattisesti
```

`with`-lauseen sisällä tiedostoa käsitellään tavallisesti; kun siitä poistutaan, tiedosto suljetaan. Käytännön esimerkki lienee tarpeen, joten kokeillaan samalla tiedostoon kirjoittamista.

Esimerkki 7.3: Tiedoston käsittely `with`-lauseella

```
1  with open("lista.txt", "a") as lista:
2          print("Kirjoita tiedostoon lisää rivejä; tyhjä rivi
   ↪  lopettaa.")
3          while True:
4                  rivi = input("Syötä rivi: ")
5                  if rivi == "":
6                          break
7                  else:
8                          lista.write(rivi + "\n")
9
10 print("Lopetettu, tiedostossa on nyt:")
11
12 with open("lista.txt", "r") as lista:
13         print(lista.read())
```

Esimerkki on pitkä, joten käydään se läpi huolellisesti. Ensimmäisellä rivillä tiedosto avataan `"a"`-tilassa, joten käyttäjän tekemät muutokset säilyvät: jos ohjelman ajaa monta kertaa, se lisää uudet rivit tiedostoon edellisten perään.

Kuten aiemmatkin esimerkit, tämä hyödyntää ikuista `while`-silmukkaa, josta poistutaan `break`-avainsanalla sitten, kun käyttäjä syöttää tyhjän rivin ja tulkitaan, että ohjelman suoritus saa pysähtyä.

Rivillä 8 tiedostoon kirjoitetaan käyttäjän syöttämä rivi yhdistettynä koodinvaihtomerkillä aikaansaatuun rivinvaihtoon `\n`, mikä saa aikaan sen, että jokainen rivi on omalla rivillään. Jos näin ei tehtäisi, käyttäjän syöte ilmestyisi tiedostoon yhdellä rivillä ilman mitään erottimia.

Rivillä 12 sama tiedosto avataan uudestaan, mutta tällä kertaa `"r"`-tilassa, jolloin se voidaan lukea.

Ohjelman suoritus voisi edetä vaikkapa näin:

```
Kirjoita tiedostoon lisää rivejä; tyhjä rivi lopettaa.
Syötä rivi: kissa
Syötä rivi: koira
Syötä rivi: hevonen
Syötä rivi:
Lopetettu, tiedostossa on nyt:
kuha
kissa
koira
hevonen
```

Ulostulon rivi `kuha` oli tiedostossa jo valmiiksi.

7.4 Virheidenhallinta tiedostojen kanssa

Luvun ensimmäisestä osiosta tutulla `try`-syntaksilla voi käsitellä myös tiedostoista aiheutuvia virheitä. `IOError`-virhe syntyy, kun haluttua tiedostoa ei ole olemassa tai sitä ei muusta syystä voida käsitellä. Esimerkissä luetaan käyttäjän syöttämän tiedoston sisältö ja tulostetaan avulias virhe, jos jokin menee pieleen.

Esimerkki 7.4: Tiedostot ja virheidenkäsittely

```
try:
        with open(input("Syötä tiedostonimi: ")) as tiedosto:
                print("Tiedoston sisältö on:")
                print(tiedosto.read())
except IOError:
        print("Virhe lukiessa tiedostoa; onko se varmasti
 ↪  olemassa?")
```

Jos käyttäjän syöttämää tiedostoa ei ole, ulostulo näyttää seuraavalta:

```
Syötä tiedostonimi: tätätiedostoa.eiolemassa
Virhe lukiessa tiedostoa; onko se varmasti olemassa?
```

Muussa tapauksessa tulostetaan tiedoston sisältö.

7.5 Tehtäviä

7.5.1 Muokkaa esimerkkiä 7.4 sellaiseksi, että tiedostonimeä kysytään, kunnes tiedosto on olemassa. Käytä `while`-silmukkaa.

7.5.2 Tee ohjelma, joka kääntää annetun tiedoston rivit siten, että ensimmäisestä rivistä tuleekin viimeinen ja niin edelleen.

7.5.3 Keksi jokin Python-koodirivi, joka tuottaa seuraavan virheen:

(a) `ValueError`
(b) `TypeError`
(c) `IndexError`

7.5.4 Oleta, että on olemassa jokin tiedosto, jossa on jokaisella rivillä lämpötilamittauksen tulos, esimerkiksi

```
15.4
17.5
16.2
...
```

Tee ohjelma, joka lukee mittaustulokset käyttäjän antamasta tiedostosta ja laskee niiden keskiarvon. Jos käyttäjän antamaa tiedostoa ei ole olemassa, tulosta hyödyllinen virheviesti. Testaa ohjelmaasi omalla esimerkkitiedostollasi.

7.5.5 `open()`-funktiolle voi antaa parametrinä myös *merkistökoodauksen*, esimerkiksi `open("tiedosto.txt", "r", encoding="utf-8")`. Selvitä Googlen avulla, mikä on merkistökoodaus ja miksi se täytyy toisinaan määritellä, kun lukee tiedostoa.

Luku 8

Matematiikka ja satunnaisuus

Ilmestyy pian.

Luku 9

Olio-ohjelmointi

9.1 Käsitteet luokka ja olio

Olemme tähän mennessä tutustuneet erilaisiin tyyppeihin. *Luokaksi* kutsutaan yleensä ohjelmoinnissa sellaista tyyppiä, jolla on seuraavat ominaisuudet:

- Luokalla voi olla *tila*, eli siihen voi liittyyä jonkinlaisia arvoja. Esimerkiksi tuotetta kuvaavan luokan tila voisi koostua sen nimestä ja hinnasta, joita kutsutaan luokan *attribuuteiksi*.
- Luokalla voi olla *metodeita*, eli voi olla jonkinlaisia toimintoja, joita luokka voi suorittaa. Lista voidaan järjestää siten, että sen alkiot ovat suuruusjärjestyksessä; merkkijono voidaan lisätä toiseen merkkijonoon.
- Luokka voidaan *periyttää* toisesta luokasta, jolloin luokka saa yläluokkansa tilan ja metodit. Tästä lisää Periytys-osiossa.

Python 2:ssa tyypit oli jaoteltu luokkiin ja muihin tyyppeihin, mutta Python 3:ssa kaikki tyypit ovat luokkia, joten käytännössä luokka ja tyyppi ovat Pythonissa nykyisin synonyymejä.

Arvoa, joka on jonkin luokan tyyppinen, sanotaan luokan *olioksi* (sekä *esiintymäksi*, *objektiksi* tai *instanssiksi*). Esimerkiksi arvot `"Hei!"` ja `"merkkijono"` ovat `str`-luokan olioita ja arvo `7` on `int`-luokan olio.

Luokan voi siis ajatella olevan suunnitelma, jonka perusteella olioita luodaan: se kertoo, mitä tietoja oliot talletavat. `Piste`-luokka määrittelee, että pisteellä on x- ja y-koordinaatit; jokainen piste tietää, mitkä sen omat arvot näille attribuuteille ovat.

9.2 Omien luokkien määritteleminen

Käsitteitä ei opeteltu huvin vuoksi, vaan ne ovat tarpeen, kun harjoitellaan omien luokkien luomista. Ei käsitellä vielä metodeita tai periytystä; keski-

tytään yksinkertaisiin luokkiin, joilla on ainoastaan attribuutteja. Aiemmin mainittiin esimerkkinä `Piste`-luokka, jolla oli attribuutteinaan kaksi koordinaattia – katsotaan, miltä se Pythonissa näyttää.

Esimerkki 9.1: Luokka attribuuteilla

```
1 # Määritellään Piste-luokka
2 class Piste:
3         pass # Tyhjä luokka
4
5 # Luodaan olio
6 p = Piste()
7 p.x = -7
8 p.y = 4.5
9 print("Koordinaatit ovat (", p.x, ", ", p.y, ")")
```

Luokan määrittely aloitetaan syntaksilla `class` *`LuokanNimi`*`:`, ja kaikki sen jälkeen tuleva sisennetty koodi liittyy luokkaan. Attribuutteja ei tarvitse erikseen määritellä ennalta, joten `pass`-avainsana kertoo Pythonille, että luokka on tarkoituksella tyhjä.

Kun luokka on määritelty, sen olioita voi luoda syntaksilla *`LuokanNimi()`*. Attribuutteja voi kirjoittamalla *`olio.attribuutti`* käyttää aivan kuten tavallisia muuttujia – esimerkissä asetetaan pisteen x-koordinaatti kirjoittamalla `p.x = -7`, ja saman attribuutin arvon saa yksinkertaisesti p.x. Jos attribuutin unohtaa määritellä ennen sen käyttöä, tulee `AttributeError`-virhe.

Viimeisellä rivillä käytetään `print`-funktion muotoa, joka ei välttämättä muistu mieleen. Jos sille antaa useita arvoja (`print(`*`arvo1, arvo2, ...`*`)`), funktio tulostaa ne välilyönnillä erotettuna. Siispä ohjelman ulostulo on

```
Koordinaatit ovat ( -7 , 4.5 )
```

On helppoa nähdä, että näinkin yksinkertaisista luokista voi olla hyötyä koodin siisteyden kannalta. Jos ohjelma käsittelee useita pisteitä, niiden koordinaatit säilyvät kätevästi `Piste`-olioiden attribuuteissa eivätkä mene sekaisin keskenään. Pisteitä käsittelevien funktioiden ei tarvitse ottaa argumentteinaan koordinaatteja erikseen, vaan kaikkialla voi pyörittää olioita, jotka niitä säilövät.

9.3 Metodit

Metodeitakin aloitteleva Python-ohjelmoija on nähnyt. `merkkijono.upper()`, `lista.sort()` ja `tuple.count(alkio)` ovat kaikki metodikutsuja, joissa kutsutaan olion metodia tietyllä listalla argumentteja (kahdessa niitä on nolla,

viimeisessä ainut argumentti on `alkio`). Metodeita voi ajatella funktioina, jotka liittyvät olioon: funktion

`funktio(olio, arg1, arg2, ...)`

voi halutessaan muuttaa metodiksi

`olio.metodi(arg1, arg2, ...)`

Otetaan käytännön esimerkki. Oletetaan, että `Piste`-luokka on määritelty edellisen esimerkin tapaan tyhjäksi luokaksi. Haluamme kirjoittaa funktion, joka laskee kahden pisteen välisen etäisyyden. Googlettamalla tai käymällä lukion matematiikan oppimäärän saa selville, että tätä varten on olemassa pisteille (x_1, y_1) ja (x_2, y_2) kaava $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Pythoniksi muutettuna se näyttää tältä:

```
1 import math # tarvitaan sqrt-funktioon
2
3 def etaisyys(piste1, piste2):
4         dx = piste1.x - piste2.x
5         dy = piste1.y - piste2.y
6         return math.sqrt(dx**2 + dy**2)
```

Nyt voisimme laskea pisteiden `p1` ja `p2` etäisyyden kirjoittamalla `etaisyys(p1, p2)`, mutta toteutetaan se saman tien metodilla.

Metodin määrittely muistuttaa hyvin paljon funktion määrittelyä, mutta oleellisin ero on se, että metodi kirjoitetaan sisennettynä luokan sisään attribuutin tavoin. Tässä `Piste`-luokan paranneltu määritelmä, jossa on mukana `etaisyys`-metodi.

Esimerkki 9.2: Luokka metodeilla

```
1 class Piste:
2
3         def etaisyys(self, toinen):
4                 dx = self.x - toinen.x
5                 dy = self.y - toinen.y
6                 return math.sqrt(dx**2 + dy**2)
```

Metodin ensimmäinen argumentti on olio, jolla metodia kutsutaan. Sille annetaan tavanomaisesti nimeksi `self`. Muuta kummallista esimerkissä ei ole: kuten funktiotkin, metodit palauttavat paluuarvon `return`-lauseella.

Nyt pisteiden välisiä etäisyyksiä voisi laskea näin:

```
origo = Piste()
origo.x = 0
origo.y = 0
toinen = Piste()
toinen.x = 1
toinen.y = 1
print(origo.etaisyys(toinen))
```

Metodin argumentti `self` saa siis arvon `origo`, joka on olio, jonka metodia kutsutaan, ja argumentti `toinen` on `Piste`-luokan olio `toinen`.

9.4 Konstruktori

Luokille voidaan määritellä tietyn nimisiä metodeita, jotka mahdollistavat erityisiä toimintoja. Yksi tällainen metodi on *konstruktori*, joka suoritetaan aina silloin, kun uusi olio luodaan. Konstruktorimetodin nimi on `__init__`, ja se voi `self`-argumentin lisäksi halutessaan ottaa mielivaltaisen määrän argumentteja, jotka on annettava oliota luodessa. Tässä esimerkki `Piste`-luokan konstruktorista, joka ottaa olioita luodessa koordinaattien arvot.

Esimerkki 9.3: Konstruktori `Piste`-luokalle

```
class Piste:

        def __self__(self, x, y):
                self.x = x
                self.y = y
```

Nyt pisteitä voi kätevästi luoda syntaksilla `Piste(x, y)` – lyhyemmän ja selkeämmän koodin lisäksi enää ei tarvitse huolehtia, että ohjelmoija unohtaa määritellä jonkin oliolle vaadituista attribuuteista. Huomaa, kuinka konstruktorin sisällä `x` viittaa konstruktorin argumenttiin `x` ja `self.x` olion `x`-attribuuttiin.

Esimerkkikonstruktori vain asettaa olion attribuutteihin annetut arvot, mutta se on ihan tavallinen metodi, jossa saa tehdä muutakin. Halutessaan tarkastella koodinsa toimintaa voi esimerkiksi tulostaa konstruktorissa viestin, jolloin huomaa aina, kun uusi olio luodaan.

9.5 Muita erityisiä metodeita

Konstruktorin lisäksi on muitakin samalla tavalla nimettyjä metodeita, jotka määrittelemällä saa käyttöön uusia ominaisuuksia luokalle. Tässä taulukoituna tärkeimpiä.

Metodi	Tarkoitus
`__repr__`	Palauttaa oliota kuvaavan merkkijonon, joka on tavanomaisesti Python-lauseke, jolla sen voisi luoda. Käytetään olion tulostamiseen
`__str__`	Palauttaa oliota kuvaavan käyttäjäystävällisen merkkijonon. Käytetään olion tulostamiseen
`__len__`	Palauttaa olion pituuden. Käytetään, kun kutsutaan funktiota `len(`*`olio`*`)`
`__bool__`	Palauttaa olion totuusarvon. Käytetään, jos olio on `if`- tai `while`-lauseen ehtona

Esimerkiksi metodit `__repr__` ja `__str__` voisi toteuttaa `Piste`-luokalle näin:

```
class Piste:
        # Luokan varsinainen määritelmä

        def __repr__(self):
                return "Piste(" + str(self.x) + ", " +
    ↪  str(self.y) + ")"

        def __str__(self):
                return "(" + str(self.x) + ", " + str(self.y) +
    ↪  ")"
```

Nyt `Piste`-luokan olioita voi tulostaa.

```
print(Piste(5, 6))
```

```
(5, 6)
```

Vain `__repr__` on välttämätöntä määritellä, sillä jos `__str__` puuttuu, sitä käytetään sen sijaan.

9.6 Periytys

Periytys on tapa jakaa koodia usean luokan kesken. Kuvitellaan, että on määritelty luokat `Kissa`, `Koira` ja `Hevonen`. Kaikilla on varmaankin samanlaisia toimintoja – esimerkiksi nimi ja omistaja – mutta myös eroavaisuuksia. Kuinka toteuttaa tilanne siten, ettei koodia tarvitse toistaa luokkien välillä?

Ratkaisu löytyy: periytys. Periytys on suhde luokkien välillä: jos luokka `A` perii luokan `B`, sanotaan, että `A` on `B`:n alaluokka ja `B` `A`:n yläluokka. Luokalla on kaikki sen yläluokkien attribuutit ja metodit. Jos `Kissa` on `Eläin`-luokan alaluokka, kaiken sen, mitä voi `Eläin`-olioille tehdä, toimii myös `Kissa`-olioille.

Tämän lisäksi alaluokat voivat *yliajaa* yläluokkien metodeita. Jos ala- ja yläluokka määrittelevät saman metodin, alaluokan metodi voittaa, kun päätetään, mitä olio käyttää. Selventävän esimerkin aika.

Esimerkki 9.4: Yksinkertainen periytys

```
1  class Elain:
2
3          def __init__(self, nimi, omistaja):
4                  self.nimi = nimi
5                  self.omistaja = omistaja
6
7          def aantele(self):
8                  print("<epämääräisiä ääniä>")
9
10 class Kissa(Elain):
11
12         def aantele(self):
13                 print("Miau!")
14
15 kissa = Kissa("Maru", "maruhanamogu") # Kutsutaan yläluokan
   ↪  konstruktoria
16 print(kissa.nimi) # Yläluokan attribuutti toimii
17 kissa.aantele() # Kutsutaan alaluokan metodia
```

Ohjelma tulostaa

```
Maru
Miau!
```

Tarkastellaanpa esimerkkiä tarkemmin. `Kissa`-eläin määritellään syntaksilla `class Kissa(Elain):`; yläluokka laitetaan suluissa luokan nimen jälkeen. Periä voi myös useampia luokkia erottamalla ne pilkuilla: `class` *`Alaluokka`*`(`*`Yläluokka1`*`,` *`Yläluokka2`*`,` *`Yläluokka3`*`, ...)`

Attribuutti `nimi` määritellään `Elain`-luokassa, mutta sitä voi käyttää automaattisesti alaluokassa. `aantele()`-metodi on määritelty kummassakin, mutta alaluokan toteutus yliajaa yliluokan, joten `kissa` tulostaa metodia kutsuttaessa `Miau!`, ei `<epämääräisiä ääniä>`

9.7 Tehtäviä

9.7.1 Selitä lyhyesti käsite...

(a) ... luokka

(b) ... olio

(c) ... konstruktori

(d) ... attribuutti

9.7.2 Muokkaa esimerkkien `Piste`-luokkaa.

(a) Lisää metodi `keskipiste`, joka laskee pisteen ja argumenttipisteen keskipisteen.

(b) Lisää metodi `siirra`, joka asettaa pisteen koordinaateille uudet arvot.

(c) Lisää metodi `lahinpiste`, joka käy läpi argumenttina annetun listan pisteitä ja palauttaa lähimmän pisteen.

9.7.3 Muokkaa `Elain`-esimerkkiä.

(a) Lisää `Elain`-luokan perivä alaluokka `Koira`, joka yliajaa `aantele`-metodin sopivasti.

(b) Muokkaa luokkahierarkiaa siten, että `omistaja`-attribuutti onkin `Elain`-luokan alaluokalla `Lemmikki`. `Kissa` ja `Koira` ovat `Lemmikki`-luokan alaluokkia.

(c) Kokeile, mitä tapahtuu, jos luokka `OutoElain` perii sekä luokan `Kissa` että luokan `Koira`. Mitä tapahtuu, jos sen `aantele()`-metodia kutsuu?

9.7.4 Tee `Tuote`-luokka, jolla on attribuutit `nimi`, `hinta` ja `alennusprosentti`. Tee sille metodi `todellinenHinta()`, joka laskee todellisen hinnan. Talleta listaan erilaisia `Tuote`-olioita (joillakin täytyy olla sama nimi). Tee ohjelma, joka kysyy käyttäjältä tuotteen nimiä ja tulostaa halvimman nimeen täsmäävän tuotteen tiedot. Vinkki: määrittele tuotteelle metodi `__repr__()`, jotta voit tulostaa sen helposti.

Luku 10

Siistin koodin käytänteitä

Siitä, että osaa ohjelmoida, ei automaattisesti seuraa, että osaisi ohjelmoida hyvin. Selkeän, siistin ja helppolukuisen koodin kirjoittaminen on taito, jota ei opi harjoittelematta; tässä luvussa tutustutaan muutamiin periaatteisiin, joita noudattamalla ohjelmiensa ymmärrettävyyttä voi kohentaa.

Miksi kirjoittaa siistiä koodia? Jos kyse on kymmenen rivin sovelluksesta, joka heitetään roskakoriin välittömästi käytön jälkeen, on totta, että tyylisääntöihin ei kannata haaskata aikaa. Jokainen ohjelmointia laajemmin harrastava törmää kuitenkin joskus seuraaviin ongelmatilanteisiin:

- Ohjelman parissa työskentelee useampi ihminen, jotka eivät saa selvää toistensa koodista
- Kuluu niin kauan aikaa, että ohjelmoija itse unohtaa, mitä on koodia kirjoittaessaan tarkoittanut
- Ohjelma kasvaa niin monimutkaiseksi, että alussa tehdyt huolimattomat ratkaisut estävät tehokkaan laajentamisen

Kaikkeen ei voi varautua, mutta siistin koodin periaatteet auttavat myös kirjoittamaan ohjelmia, joita on helpompaa muokata myöhemmin.

10.1 Oikeiden rakenteiden käyttäminen

Python – kuten mikä tahansa ohjelmointikieli – tarjoaa useita tapoja toteuttaa minkä tahansa ohjelman. Jos haluamme esimerkiksi tulostaa muuttujassa `lista` sijaitsevan listan alkiot, jokainen omalle rivilleen, kumpi tahansa seuraavista esimerkkiohjelmista toimii.

```
# Ohjelma 1
indeksi = 0
while indeksi < len(lista):
        alkio = lista[indeksi]
        print(alkio)

# Ohjelma 2
for alkio in lista:
        print(alkio)
```

Ohjelma 2 on tietenkin siistimpi toteutus. Kun itse miettii, mikä monista vaihtoehdoista kannattaa valita, voi miettiä seuraavia seikkoja:

- Suosi lyhyttä toteutusta. 5 rivin ohjelmaa on todennäköisesti helpompi ymmärtää kuin sellaista, jossa on 10 riviä.
- Suosi idiomaattista toteutusta. `for`-lausetta käytetään tietorakenteiden läpikäymiseen, joten kokenut ohjelmoija ymmärtää heti sen nähdessään, mistä on kyse. Ohjelma 1 voi puolestaan hämätä, koska useimmiten `while`-lausetta käytetään johonkin muuhun.
- Suosi Pythonin valmiita funktioita. Omaa toteutusta ei kannata kirjoittaa, jos Pythonissa on jo vaihtoehto, joka on suurelle osalle koodia lukevista tuttu ennestään.
- Suosi virheitä ehkäisevää toteutusta. Jos kolmas 3 olisikin `while indeksi <= len(lista):`, virhettä ei välttämättä heti huomaisi. Ohjelma 2 ei tarvitse `indeksi`-muuttujaa ollenkaan, mikä tekee mahdottomaksi sen väärinkäytöstä johtuvat virheet. Koodia tarkasti lukevan ei myöskään tarvitse käyttää aivokapasiteettiaan sen pohtimiseen, onko kaikki koodissa varmasti oikein.

Otetaan vielä toinen esimerkki. Vertaillaan äskeisillä kriteereillä kahta tapaa tulostaa tiedoston sisältö.

```
# Ohjelma 1
tiedosto = open("tiedosto.txt", "r")
print(tiedosto.read())
tiedosto.close()

# Ohjelma 2
with open("tiedosto.txt", "r") as tiedosto:
        print(tiedosto.read())
```

Ohjelma 2 on yhden rivin lyhyempi. Se on myös selvän idiomaattinen – `with open(...)` kommunikoi välittömästi, että on kyse tiedostosta, jota

käsitellään lohkon sisällä. Se, ettei `close()`-metodia tarvitse kutsua itse, poistaa sen virheen vaaran, että näin unohtaa tehdä; olisitko huomannut, jos `close()` olisi puuttunut ohjelmasta 1?

Oikeiden rakenteiden käyttämisessä on pohjimmiltaan kyse siitä, kuinka viestiä koodin lukijalle mahdollisimman selvästi, että ohjelma tekee juuri sen, mitä se tekee. Jos on epävarma, voi vaikka kysyä kaverilta, mikä toteutus näyttää selkeimmältä.

10.2 Hyvin nimeäminen

Osaisitko arvata, mitä seuraava funktio tekee?

```
1 def f(a, b, c):
2         d = 0
3         for e in a:
4                 d = d + e
5         return b >= d * c
```

Huolellisella tarkastelulla kyllä selviäisi, `mitä` funktio palauttaa, mutta minkäänlaista kontekstia ei välttämättä saa ilman arvailuja. Miksi tällainen funktio on olemassa? Mitä hyötyä siitä on? Jos se toimii väärin, kuinka korjaan sen?

Vertaa ymmärrettävyyttä seuraavaan funktioon:

```
1 def voiko_ostaa(tuotteiden_hinnat, rahaa, alennus):
2         summa = 0
3         for hinta in tuotteiden_hinnat:
4                 summa = summa + hinta
5         return rahaa >= summa * alennus
```

Pelkkä se, että muuttujat, funktiot ja niiden argumentit nimeää siististi, avustaa ohjelman tulkitsemisessa. Ylhäällä esitellyt funktiot eivät toiminnallisesti eroa mitenkään, mutta on ilmiselvää, että alempi on se, mitä koodinsa luettavuudesta huolehtivan kannattaa tavoitella.

Hyvä nimi – vaikka tämä ilmiselvältä kuulostaisi – kertoo, mistä on kyse. `hinta`-niminen muuttuja kertoo, että kyse on jonkin hinnasta, kun taas `e` ei juuri mitään (ellei kyseessä ole Neperin luku e). Hyvä nyrkkisääntö onkin, että yksikirjaimisia muuttujia kannattaa käyttää vain silloin, kun kyse on matematiikan tai fysiikan vakiintuneesta tunnuksesta. Lisäksi käydessä läpi vaikkapa lukuväliä käytetään perinteisesti muuttujaa `i`.

Helpointa on luetella sääntöjä sille, minkälainen hyvin nimetty funktio tai muuttuja ei ainakaan saa olla. Hyvä nimi ei...

- ... ole tarpeettoman lyhyt (`a`, `ht` tarkoittamaan hajautustaulua tai `li` listaa)

- ... ole toisaalta liian pitkä (`lisää_tuote_tietokantaan_jos_nettiyhteys_toimii` on hyvä esimerkki; käytä kommentteja, jos haluat kertoa funktiosta lisää)
- ... sisällä turhia täytesanoja, kuten `data`, `tieto` tai jopa `muuttuja` (`merkkijono_data_tieto_muuttuja` on pitkä nimi, mutta käytännössä viestii vain, että kyseessä on merkkijono)
- ... valehtele. Jos muuttujan nimi on `luku`, siihen ei missään tapauksessa aseteta myöhemmin merkkijonoa.

10.3 Hyvä kommentoiminen

Kommentteihin tutustuttiin jo kirjan ensimmäisessä luvussa: `#`-merkki aloittaa kommentin, joka jatkuu rivin loppuun saakka. Nyt, kun tietämystä ohjelmoinnista on ehtinyt kertyä enemmän, on hyvä hetki tutustua hyödyllisen kommentoinnin käytäntöihin.

Hyvä kommentti ennen kaikkea `välittää sellaista tietoa, jota koodista itsestään ei voi päätellä`. Miksi tehdään juuri näin? Miksi tämä on tarpeen? Onko mielessä parannushdotus tulevan varalle? Pitääkö tätä funktiota käyttäessään tietää jotakin erityistä? Kaiken tällaisen voi ilmaista kommentilla.

Käytännön esimerkki hyvästä kommentista lienee tarpeen.

```
1 # Muutetaan tuple listaksi, jotta sen arvoja voi muokata
2 nimilista = list(nimituple)
```

Jos koodin lukija ei muista, että tuplejen arvoja ei voi muokata, rivin tarkoitus voi jäädä pimentoon. Täsmentävä kommentti selittää, miksi näin on tehtävä, jotta joku ei esimerkiksi poista riviä.

Kommentteja käytetään usein myös ennen funktion määrittelyä kertomaan lisätietoja sen argumenteista, paluuarvosta tai muista huomioitavista asioista.

```
1 # Palauttaa indeksin, jossa annettu alkio annetussa listassa
  ↪ sijaitsee.
2 # Listan ei tarvitse olla järjestetty, koska funktio kopioi
  ↪ sen itselleen
3 # ja järjestää kopion.
4 def binaarihaku(lista, alkio):
5         ...
```

Huonoja kommentteja on monenlaisia. Jotkut ovat niin tarpeettomia, että vain hämäävät lukijaa, joka jää pohtimaan, miksi kommentti ylipäätään on olemassa:

```
# laskee, mitä 2 + 3 on
tulos = 2 + 3
```

Joskus kommentti voi jopa valehdella, jos koodia on muokattu unohtamatta muuttaa sitä. Tälläkin voi olla hämäävä vaikutus:

```
# Muuttaa kissan nimen. Argumentin on oltava merkkijono!
def muuta_nimi(uusi_nimi):
        nimi = str(uusi_nimi)
        if len(nimi) < 3:
                raise ValueError("Nimi on liian lyhyt")
        ...
```

Voisi arvata, että ohjelmoija lisäsi funktioon sen ominaisuuden, että se muuttaa argumenttinsa merkkijonoksi `str`-funktiolla huomattuaan, että tästä on jotain hyötyä, mutta unohti päivittää kommentin, jonka huomautus on nyt valheellinen.

Koska kommentti vie lukijan huomion, on ylipäätään järkevää harkita, mitkä kommenteista ovat todella tarpeen. Kriteerit saa kuitenkin suhteuttaa kodeyleisöönsä: Python-kurssin aloittelijoille kannattaa selventää sellaistakin, mitä ei Pythonia työkseen ohjelmoivalle selventäisi.

10.4 Tyylisääntöjen noudattaminen

Viralliset ohjeet Python-koodin muotoilemiseen löytyvät seuraavalta sivulta: `https://www.python.org/dev/peps/pep-0008/` Dokumentti on varsin pitkä, joten tässä tiivistettynä se, mitä aloittelijan on hyvä tietää.

- Muuttujat ja funktiot kirjoitetaan pienellä alkukirjaimella: `kissa`, `kasvatalukua`, `lähinpiste`
- Halutessaan sanoja niiden sisällä saa erottaa alaviivalla: `kasvata_lukua`, `lähin_piste`
- `import`-lauseet laitetaan peräkkäin tiedoston alkuun
- Turhia välilyöntejä vältetään: `funktio(arg1, arg2)`, ei `funktio ( arg1, arg2 )`; `lista[indeksi]`, ei `lista [ indeksi ]`
- Luokat kirjoitetaan `IsollaJaYhteen`

Miksi noudattaa tyylisääntöjä? Säännöt itsessään ovat melko mielivaltaisia (toki voi perustellusti olla sitä mieltä, että `tätäonärsyttävääalukea` ja `tätä_on_miellyttävää_lukea`), mutta niiden noudattaminen ei. Niilläkin

on koodin lukemista helpottava vaikutus: kun kaikki noudattaa yhdenmukaista säännöstöä, nopea silmäily vaikkapa kertoo, että `kissa` on muuttuja, mutta `Kissa` luokka.

Pythonin virallisten tyylisääntöjen noudattamista tärkeämpää se, että jatkaa niiden noudattamista, joita muokattavassa koodissa jo suositaan. Jos sinut laitetaan jatkamaan projektia, jossa on 50000 `näinKirjoitettuaMuuttujaa`, kannattaa mieluummin jatkaa perinnettä kuin muuttaa kaikki `virallisten_sääntöjen_mukaisiksi`.

Sanasto

alkio (engl. *element, item, member*) Jonkin tietorakenteen sisältämä arvo. Esimerkiksi listan `["a", "b"]` alkioita ovat `"a"` ja `"b"`. 34, 65, 66

attribuutti (engl. *attribute*) Johonkin olioon liitetty muuttuja. `kissa.nimi` on `kissa`-olion attribuuttia `nimi`. 52, 66

avainsana (engl. *keyword*) Pythonin varaama termi, kuten `if` tai `while`, jota ei saa käyttää muuttujan nimenä. 11

dokumentaatio (engl. *documentation*) Ohjelmiston tai ohjelmointikielen hakuteosta muistuttava asiakirja, joka kertoo yksityiskohtaisesti sen ominaisuuksista. Vaatii yleensä esitietoja. Pythonin dokumentaatio löytyy osoitteesta `https://docs.python.org/3/`. 2

funktio (engl. *function*) Ohjelmoinnissa sellainen arvo, jota voidaan kutsua antamalla sille nolla tai useampi argumenttia. Funktioilla voi olla paluuarvo, sivuvaikutuksia tai ei kumpaakaan. Esimerkiksi `print` on funktio, joka tulostaa sille annetun argumentin. 6, 26, 65, 67

funktiokutsu (engl. *function call*) Eräs lauseke, jossa on tietyn funktion nimi sekä suluin ympäröidyt argumentit. Esimerkiksi `print("Hello World!")` on funktiokutsu. 26

hajautustaulu (engl. *hash table, hash map, dictionary*) Tietorakenne, jossa jokaista avainta vastaa jokin arvo. Avaimet ja arvot voivat olla mitä tahansa tyyppiä. 39

IDLE (Integrated Development and Learning Environment) Helppokäyttöinen ohjelma Python-koodin käsittelemiseen. 4

indeksoitu (engl. *ordered, indexed*) Tietorakenne on indeksoitu, jos sen alkiot ovat jossakin järjestyksessä. Esimerkiksi listat `[4, 5]` ja `[5, 4]` ovat eri listoja, koska niiden alkiot ovat eri järjestyksessä, joten lista on indeksoitu tietorakenne. 66, 67

joukko (engl. *set*) Tietorakenne, jonka alkiot eivät ole järjestyksessä, ja jossa jokaista alkiota voi olla vain yksi kappale. 37

komentotulkki (engl. *shell* tai `interpreter`) Interaktiivinen ohjelma, johon käyttäjä syöttää ohjelmakoodia, joka suoritetaan välittömästi. Python-komentotulkin saa auki `python`-komennolla; myös IDLE:ssä on komentotulkki. 4

konstruktori (engl. *constructor*) Erityinen metodi, joka suoritetaan silloin, kun olio luodaan. 55

koodinvaihtomerkki (engl. *escape character*) Koodinvaihtomerkki on jokin merkki (Pythonissa kenoviiva \), jonka avulla voidaan kirjoittaa merkkejä, jotka muuten tulkittaisiin virheellisesti. Esimerkiksi merkkijonojen sisällä lainausmerkin saa kirjoittamalla `\"`, sillä pelkkä lainausmerkki tulkittaisiin merkkijonon päättymiseksi. 7, 48

lause (engl. *statement*) Sellainen pätkä Python-koodia, joka voi esiintyä itsenäisesti. Esimerkiksi `print("kissa")` tai `x=6` ovat lauseita. 6

lauseke (engl. *expression*) Osa koodia, jolla on jollakin tyypillä kuvattava arvo. Esimerkiksi `2+4`, `"merkkijono"` ja `int("66")` ovat lausekkeita. 65, 66

lista (engl. *list*) indeksoitu, muuttuva tietorakenne, jonka voi määritellä syntaksilla `[1, 2, 3]`. 34, 65, 67

liukuluku (engl. *floating point number*) Pythonin vastine desimaaliluvuille. Liukuluvuilla laskeminen on niiden sisäisestä esityksestä johtuen epätarkkaa. 14

luokka (engl. *class*) Yleisnimitys tyypille, jolla on attribuutteja, metodeja ja periytys. Pythonissa kaikki tyypit ovat luokkia. 52, 66, 67

merkkijono (engl. *string*) Merkeistä koostuva pätkä tekstiä. Pythonissa merkkijonoja voi merkitä asettamalla ne lainausmerkkien sisään: esimerkiksi `"kissa"` on merkkijono. 6, 24, 35

metodi (engl. *method*) Olioon liitetty funktio. `olio.metodi(argumentit)` kutsuu `olio`-olion metodia `metodi`. 53, 66, 67

muuttuja (engl. *variable*) Lauseke, joka viittaa sille aiempin määriteltyyn arvoon. Muuttujien nimillä on joitakin rajoituksia; `x3` ja `var` ovat sallittuja, mutta `muut)` ja `4y` ovat kiellettyjä. 10, 65

olio (engl. *object*) Jonkin luokan yksi esiintymä; esim. `7` on `int`-luokan olio. 52, 65–67

periytys (engl. *inheritance*) Luokan ominaisuus, jolla se voi kopioida toisen luokan tietoja. 52, 57, 66, 67

rekursio (engl. *recursion*) Se, että funktio kutsuu itseään. 31

semantiikka (engl. *semantics*) Termi sille, mikä merkitys koodin eri osilla on. Vertaa syntaksiin. 21, 67

sisennys (engl. *indentation*) Rivin alussa olevien välilyöntien tai tabulaattorien määrä. 16, 21

sivuvaikutus (engl. *side effect*) Sellainen toiminta, jolla funktio vaikuttaa ohjelman tilaan ja tekee jotain muutakin kuin vain laskee paluuarvonsa. 28

syntaksi (engl. *syntax*) Ohjelmointikielen kielioppi eli se, minkälaisia osia koodista on sallittua käyttää missäkin yhteydessä. Vertaa semantiikkaan. 21, 67

tietorakenne (engl. *data structure*) Arvo, joka voi sisältää muita arvoja. Esimerkiksi lista on tietorakenne. 41, 65–67

tuple (engl. *tuple*, suomeksi myös *monikko* ja *tuppeli*) Muuttumaton, indeksoitu tietorakenne. Vertaa listaan. 38, 62

tyyppi (engl. *type*) Jokaisella arvolla on tyyppi, joka kertoo sen ominaisuudet, kuten sen, mitä operaattoreita ja metodeja sillä on. 11, 52, 66

versiohallintaohjelma (engl. *version control system*) Koodin säilyttämiseen ja historiatietojen kirjaamiseen suunniteltu ohjelmisto, joka helpottaa useamman ohjelmoijan yhteistyötä. Suosittuja ovat nykyisin mm. Git ja Mercurial. 8

yliajaminen (engl. *overloading*) Se, että jonkin metodin perivä olio määrittelee sen uudelleen. 57