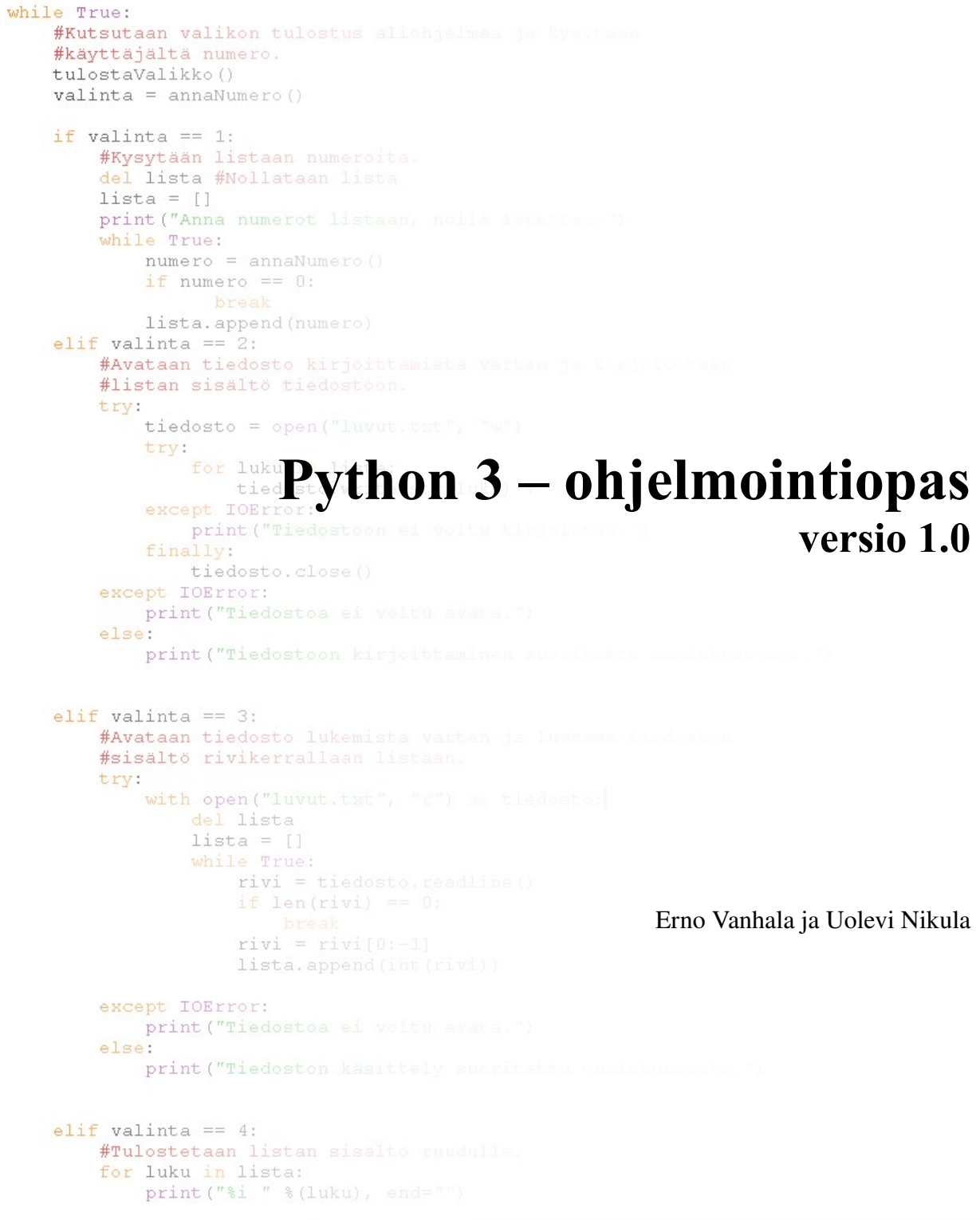

Python 3 – ohjelmointiopas

versio 1.0

Erno Vanhala ja Uolevi Nikula

Käsikirjat 13

Manuals 13

Python 3 – ohjelmointiopas
versio 1.0

Erno Vanhala ja Uolevi Nikula

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan laitos
PL 20
53851 Lappeenrantaan

ISBN 978-952-214-970-1
ISSN 1799-2680

Lappeenranta 2010

Python 3 oppaan kasaus, korjaus ja taitto:

Erno Vanhala

Alkuperäinen käännös ja lokalisointi, täydentävä materiaali sekä taitto:

Jussi Kasurinen

Toteutuksen ohjaus sekä tarkastus:

Uolevi Nikula

Lappeenrannan teknillinen yliopisto, Ohjelmistotekniikan laboratorio.

Lappeenranta x.x.2010

Alkusanat

Tervetuloa Lappeenrannan teknillisen yliopiston Ohjelmoinnin perusteet -kurssin ohjelmointioppaan pariin. Tämä on oppaan neljäs, uusittu ja päivitetty, painos. Alkuperäisen ohjelmointioppaan suunnitteli ja kirjoitti Jussi Kasurinen. Tämä uusittu painos pohjaa vahvasti siihen, mutta kaikki materiaali on käännetty Pythonin versiolle 3 sopivaksi.

Alkuperäisen ohjelmointioppaassa on lähdeaineistona käytetty kolmea verkosta saatavilla olevaa lähdettä, joista ensimmäinen on CH Swaroopin kirjoittama teos ’Byte of Python’ (www.byteofpython.info) ja toinen Python Software Foundationin ylläpitämä Python-dokumenttiarkisto (docs.python.org). Kolmas teos, josta on otettu lähinnä täydentävää materiaalia on nimeltään ’How to Think Like a Computer Scientist: Learning with Python’ (http://www.ibiblio.org/obp/thinkCS/). Kyseisen teoksen ovat kirjoittaneet Allen B. Downey, Jeffrey Elkner sekä Chris Meyers.

Tämän uusimman painoksen apuna on käytetty myös Dive Into Python 3 -oppaan materiaalia (http://diveintopython3.org/). Oppaan on kirjoittanut Mark Pilgrim.

Alkuperäinen opas on vapaa käännös. Yleisesti käännöksen alkuperäinen kieliasu on pyritty mahdollisuuksien mukaan säilyttämään, mutta joitain tekstejä on jouduttu muuntelemaan luettavuuden ja jatkuvuuden parantamiseksi. Myös esimerkit on lokalisoitu englanninkielisestä Linux-shell-ympäristöstä suomenkieliseen IDLE-kehitysympäristöön.

Aiheiden jako kokonaisuuksiin noudattaa ensisijaisesti Lappeenrannan teknillisen yliopiston syksyn 2010 kurssin ”Ohjelmoinnin Perusteet” viikoittaista jakoa. Lisäksi joissain luvuissa tekstin sekaan on lisätty aiheita, jotka eivät suoranaisesti liity viikoittaiseen aiheeseen, mutta ovat hyödyllistä lisätietoa. Lisäksi huomioi, että oppaan lopussa olevissa liitteissä on paljon hyödyllistä lisätietoa esimerkiksi tulkin virheilmoituksista ja sanastosta. Tämän oppaan tueksi on tarjolla myös asennusopas, tyyliopas ja esimerkkirakenteita sisältävä opas. Viimeiset kaksi löytyvät myös tämän oppaan liitteistä.

Kuten kurssillakin, myös teoriaosioissa sekä materiaalissa oletetaan, että käyttäjä tekee tehtäviä IDLE-kehitysympäristön avulla Windows-, Mac- tai Linux-työasemalla, ja käyttää Python-tulkin versiota 3.x.

Tämä opas on neljäs päivitetty ja korjattu versio aiemmin ilmestyneistä ensimmäisistä Python-ohjelmointioppaista ja on tarkoitettu korvaamaan kyseiset teokset, kun ohjelmoidaan Pythonin versiolla 3.

Sisällysluettelo

Luku 0: Miksi ohjelmointi on kivaa ja sen lisäksi tärkeää

Toisen maailmansodan jälkeen tietokoneiden kehitys lähti kasvuun ja sen sijaan, että yksi kone olisi osannut tehdä vain yhden asian, koneita alettiin ohjelmoida tekemään erilaisia asioita. Ensimmäiset ohjelmat olivat mahdollisimman lähellä tietokoneen itsensä ymmärtämää kieltä. Ne kirjoitettiin siis konekielellä. Sen kanssa työskentely on tehokasta, mutta erittäin hidasta, eikä sitä nykypäivänä käytetä kuin hyvin harvoin.

Seuraava askel oli kirjoittaa ohjelmat englantia muistuttavalla kielellä ja kääntää tämä koodi sitten konekielelle. Ohjelmien tuottaminen tällä tavoin oli paljon nopeampaa. Ensimmäisissä ohjelmointikielissä ei ollut paljoa toimintoja, mutta eipä ollut tietokoneissakaan tehoja – saati edes monitoreja. Ohjelmakoodi oli kuitenkin varsin järjestelemätöntä, joten niin kutsuttua spagettikoodia oli helppo kirjoittaa.

Seuraava vaihe oli siirtyä rakenteelliseen ohjelmointiin. Ensin tulivat proseduraalinen ohjelmointi ja sen jälkeen olio-ohjelmointi. Jälkimmäinen on tällä hetkellä vallitseva ohjelmointiparadigma. Tämän lisäksi on olemassa esimerkiksi funktionaalista ohjelmointia, joka voi olla vallitsevassa asemassa 20 vuoden kuluttua tai sitten jokin muu tapa todetaan paremmaksi. Hopealuotia ei ole vielä keksitty ja paljon työtä täytyy edelleen tehdä käsin.

Tulee kuitenkin huomata, että kaikki nämä ohjelmointiparadigmat on kehitetty jo 50- ja 60-luvuilla, niiden nouseminen vallitsevaan asemaan on vain kestänyt vuosikymmeniä. Huomaa myös, että tällä kurssilla käyttämämme Python-ohjelmointikieli tukee kaikkia näitä ohjelmointitapoja.

Oli ohjelmointitapa mikä tahansa, ohjelmoinnin tarkoitus on kuitenkin sama: saada tietokone tekemään asioita, joista on hyötyä käyttäjälle. Kyseessä voi olla tekstin oikoluku, jutteleminen kaverin kanssa Internetissä tai pelien pelaaminen. Jokainen näistä on vaatinut ohjelmointia.

Nykyinen yhteiskunta ei tulisi toimeen ilman tietotekniikkaa, toimivia ohjelmia ja niitä kirjoittavia alan ammattilaisia. Ohjelmoinnin perusajatusten ymmärtäminen kuuluu insinöörin osaamiseen, sillä nykypäivänä kaikki laitteet jääkaapista auton kautta kännyköihin sisältävät miljoonia ja taas miljoonia rivejä koodia. On hyvä tietää edes jotain jääkaapin sielunelämästä.

Ohjelmia voidaan kirjoittaa oikein ja ei-niin-oikein eli väärin. Kannattaa jo alusta lähtien opetella ohjelmoimaan oikein. Siitä ei ole kuin hyötyä elämän varrella (nimim. 10 vuotta väärin ohjelmoinut). Tämän oppaan tarkoitus onkin perehdyttää lukija ohjelmoinnin ihmeelliseen maailmaan ja tarjota oikeita tapoja ratkaista ongelmia ohjelmoimalla.

Muista kuitenkin: ohjelmoimaan oppii vain yhdellä tavalla – ohjelmoimalla.

Oppimisen riemua!

Luku 1: Ensiaskeleet Pythoniin

Ensimmäisessä luvussa on tarkoitus oppia käyttämään Python-ohjelmointiin tarkoitettuja työkaluja. Työkalujen avulla myös luodaan, tallennetaan ja ajetaan ensimmäinen oikea Python-ohjelma.

Python-editorin käynnistäminen

Windows-käyttäjille helpoin tapa aloittaa Pythonin käyttö on ajaa Python-koodi IDLEllä. IDLE on lyhenne sanoista Integrated DeveLopment Environment, ja sen haku- ja asennusohjeet läpikäytiin erillisessä asennusoppaassa. Ohjelma löytyy käynnistä-valikosta polun Käynnistä → Kaikki ohjelmat → Python 3.x → IDLE (Python GUI) (Start -> All Programs -> Python 3.x -> IDLE (Python GUI)) kautta (x:n tilalla Pythonin uusimman version numero). IDLE-ympäristössä on interaktiivinen komentorivitulkki-ikkuna. IDLE on saatavilla myös Linux- ja MacOS-järjestelmille.

Huomaa, että jatkossa esimerkkien merkintä `>>>` tarkoittaa tulkille syötettyä Python-käskyä. Seuraavassa annamme interaktiiviseen ikkunaan komennon **`print("Hello World!").`**

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

Huomionarvoista on se, että Python palauttaa tulostetun rivin välittömästi. Se, mitä itse asiassa kirjoitit, oli yksinkertainen Python-käsky. Pythonin syntaksi käyttääkin `print` –funktiota sille annettujen arvojen tulostamiseen ruudulle. Tässä esimerkissä annoimme sille tekstin ”Hello World”, jonka se tulosti välittömästi ruudulle.

Lähdekooditiedoston käyttäminen

Ohjelmoinnin opiskelussa on olemassa perinne, jonka mukaan ensimmäinen opeteltava asia on ”Hello World” -ohjelman kirjoittaminen ja ajaminen. Ohjelma on yksinkertainen koodinpätkä, joka ajettaessa tulostaa ruudulle tekstin ”Hello World”.

Avaa valitsemasi editori (IDLE käy siis tähän hommaan [File → New Window]) ja kirjoita alla olevan esimerkin mukainen koodi. Tämän jälkeen tallenna koodi tiedostoon helloworld.py. Huomaa, että tiedoston nimessä on, erottimena toimivan pisteen jälkeen, tarkentimena py. Tämän tarkentimen avulla ohjelmat tietävät, että tiedosto sisältää Python-koodia.

Esimerkki 1.1. Lähdekooditiedoston käyttäminen

```
# -*- coding: utf-8 -*-
# Ensimmäinen ohjelma
print("Hello World!")
```

Tämän jälkeen aja ohjelma. Jos käytät IDLEä, onnistuu tämä editointi-ikkunan valikosta Run-> Run Module. Tämä voidaan toteuttaa myös pikavalintanäppäimellä F5.

Tuloste

```
>>>
Hello World!
>>>
```

Jos koodisi tuotti yllä olevan kaltaisen vastauksen, niin onneksi olkoon – teit juuri ensimmäisen kokonaisen Python-ohjelmasi!

Jos taas koodisi aiheutti virheen, tarkasta, että kirjoitit koodisi täsmälleen samoin kuin esimerkissä ja aja koodisi uudestaan. Erityisesti huomioi se, että Python näkee isot ja pienet kirjaimet eri merkkeinä. Tämä tarkoittaa sitä, että esimerkiksi ”Print” ei ole sama asia kuin ”print”. Varmista myös, että et epähuomiossa laittanut välilyöntejä tai muutakaan sisennystä rivien eteen. Tästä puhumme lisää luvussa 2.

Miten se toimii

Katsotaan hieman tarkemmin, mitä koodisi itse asiassa sisältää. Ensimmäiset kaksi riviä ovat *kommenttirivejä*. Niitä sanotaan kommenteiksi, koska ne eivät pääsääntöisesti vaikuta ohjelman suoritukseen, vaan ne ovat muistiinpanoja, jotka on tarkoitettu helpottamaan koodin ymmärtämistä.

Kommenteista

Käytä kommentteja järkevästi. Kirjoita kommenteilla selvitys siitä, mitä ohjelmasi mikäkin vaihe tekee – tästä on hyötyä kun asiasta tietämätön yrittää tulkita kirjoitetun koodin toimintaa. Kannattaa myös muistaa, että ihmisen muisti on rajallinen. Kun kuuden kuukauden päästä yrität lukea koodiasi, niin huomaat, että se ulkopuolinen olet myös sinä itse!

Python ei käytä kommenttirivejä järjestelmän hallintaan kuin ainoastaan erikoistapauksissa. Tässä tapauksessa ensimmäinen rivi määrittelee, mitä merkkitaulukkoa halutaan koodin tulkitsemisessa käyttää. Tämä mahdollistaa esimerkiksi skandinaavisten merkkien käyttämisen teksteissä. Jotkin järjestelmät tunnistavat käytettävän merkistön automaattisesti, mutta esimerkiksi IDLE antaa käyttäjälle mahdollisuuden valita itse, mitä merkistöä haluaa käyttää. IDLE myös huomauttaa asiasta, mikäli se katsoo merkistön määrittelemisen tarpeelliseksi.

Kommenttirivejä seuraa Python-käsky `print`, joka siis tulostaa tekstin ”Hello World”. Varsinaisilla termeillä puhuttaessa `print` on itse asiassa funktio, ja ”Hello World” merkkijono, mutta näistä termeistä sinun ei tässä vaiheessa tarvitse tietää enempää. Puhumme terminologiasta ja niiden sisällöstä jatkossa enemmän.

Yhteenveto

Tässä vaiheessa sinun pitäisi osata kirjoittaa yksinkertainen Python-koodi, käyttää editoria sekä ajaa koodinpätkiä ja tallentaa lähdekooditiedostoja. Nyt kun hallitset työkalujen käytön, niin siirrytään eteenpäin ja jatketaan Python-ohjelmointiin tutustumista!

Luvun asiat kokoava esimerkki

Esimerkki 1.2. Sämpyläohje

```
# -*- coding: utf-8 -*-
# Tiedosto: jauhopeukalo.py

print("Sämpyläohje")
print("===========")
print("Maitorahkaa:    1prk")
print("Vettä:          2,5dl")
print("Hiivaa:         25g")
print("Suolaa:         1,5tl")
print("Ruishiutaleita: 1,5dl")
print("Vehnäjauhoja:   8dl")
print("Voita:          50g")
```

Luku 2: Muuttujat, tiedon vastaanotto, muotoseikat ja laskeminen

Pelkän ”Hello World” ohjelman tekeminen ei ole kovinkaan vaikeaa ja itse ohjelman käytännön hyöty jää myös vähäiseksi. Oletettavasti haluat päästä kysymään käyttäjältä syötteitä, muuntelemaan niitä ja tulostamaan ruudulle erilaisia vastauksia. Onneksi Python mahdollistaa tämän kaiken.

Vakio

Tähän mennessä törmäsimme vasta ”Hello World” tyyppiseen *vakiomerkkijonoon.* Kyseessä on siis jono kirjaimia, jotka eivät muutu vaikka ohjelman ajaisi kuinka monta kertaa. Aivan saman asian ajaa, jos vaihdamme merkkijonon tilalle numeron `5`, `10` tai `655.36`.

Numerot

Muutama sana numeroista. Pythonista löytyy erilaisia numerotyyppejä: kokonaisluvut, liukuluvut (kansanomaisesti desimaaliluvut) sekä kompleksiluvut.

- Esimerkiksi `42`, `54321` tai `-5` ovat kokonaislukuja, koska ne eivät sisällä desimaaliosaa.
- Desimaalilukuja ovat esimerkiksi `3.23` ja `52.3E-4`. Merkki E tarkoittaa kymmenpotenssia. Tässä tapauksessa, `52.3E-4` on siis `52.3 * 10`$^{-4}$. **Huomaa, että ohjelmoitaessa desimaalierottimena käytetään pistettä, eikä suomalaisittain pilkkua.**
- Kompleksilukuja ovat vaikkapa `(-5+4j)` ja `(2.3 - 4.6j)`.

Muuttujat

Pelkkien vakioarvojen käyttäminen muuttuu nopeasti tylsäksi. Tarvitsemme jonkinlaisen keinon tallentaa tietoa sekä tehdä niihin muutoksia. Tämä on syy, miksi ohjelmointikielissä, kuten Pythonissakin, on olemassa *muuttujia*. Muuttujat ovat juuri sitä mitä niiden nimi lupaa, ne ovat eräänlaisia säilytysastioita, joihin voit tallentaa mitä haluat ja muutella tätä tietoa tarpeen mukaan vapaasti. Muuttujat tallentuvat tietokoneesi muistiin käytön ajaksi, ja tarvitset jonkinlaisen tunnisteen niiden käyttämiseen. Tämän vuoksi muuttujille annetaan nimi aina, kun sellainen otetaan käyttöön.

Muuttujien nimeäminen

Muuttujat ovat esimerkki tunnisteista. Tunniste tarkoittaa nimeä, jolla yksilöidään jokin tietty asia. Samanniminen muuttuja ei siis voi kohdistua kahteen sisältöön. Muuttujien nimeäminen on melko vapaata, joskin seuraavat säännöt pätevät muuttujien sekä kaikkeen muuhunkin nimeämiseen Pythonissa:

- Nimen ensimmäinen merkki on oltava kirjain (iso tai pieni) taikka alaviiva '_'.
- Loput merkit voivat olla joko kirjaimia (isoja tai pieniä), alaviivoja tai numeroita (0-9).
- Skandinaaviset merkit (å,ä,ö,Å,Ä,Ö) ja välilyönnit eivät kelpaa muuttujien nimiin (Python 3:n myötä ääkköset periaatteessa käyvät, mutta niitä **EI** tule kuitenkaan käyttää).
- Nimet ovat aakkoskoosta riippuvaisia (eng. case sensitive), eli isot ja pienet kirjaimet ovat tulkille eri merkkejä. Siksi nimet ”omanimi” ja ”omaNimi” **eivät** tarkoita samaa muuttujan nimeä.

Näiden sääntöjen perustella kelvollisia nimiä ovat muun muassa `i`, `_mun_nimi`, `nimi_23` ja `a1b2_c3`. Epäkelpoja nimiä taas ovat esimerkiksi: `2asiaa`, `taa on muuttuja`, `jäljellä` ja `-mun-nimi`.

Muuttujien tietotyypit

Muuttujille voidaan antaa mitä tahansa Pythonin tuntemia tietotyyppejä, kuten merkkijonoja tai numeroita. Kannattaa kuitenkin huomata, että jotkin operaatiot eivät ole mahdollisia keskenään. Esimerkiksi kokonaisluvusta ei voi vähentää merkkijonoa (`32 - "auto"` ei siis toimi).

Tulee myös huomata seuraavat erot:

```
>>> a = 3
>>> b = 5
>>> a + b
8
>>> a = "3"
>>> b = "5"
>>> a + b
'35'
>>>
>>> a = 3
>>> b = 5.2
>>> a - b
-2.2
>>>
```

Eli numeroita voidaan laskea yhteen, mutta merkkijonojen tapauksessa ne liitetään yhteen. Kokonaisluvun ja desimaalinluvuilla laskeminen onnistuu myös. Python ei erottele niitä tällaisessa käytössä mitenkään.

Kuinka kirjoitinkaan ohjelman Pythonilla

Aloitetaan jälleen kerran perusteista; tehdäksesi Pythonilla esimerkin ohjelman, toimi seuraavasti:

1. Avaa mieleisesi koodieditori (IDLE → New window).
2. Kirjoita alla olevan esimerkin mukainen koodi, muista kirjainkoot.
3. Tallenna tiedosto nimellä muuttuja.py. Vaikka editorisi ei tätä automaattisesti tekisikään, on hyvä muistaa Python-koodin pääte **py**, jotta käyttöjärjestelmä – ja jotkin editorit – tunnistavat tiedostosi Python-koodiksi.
4. Käytä IDLEn komentoa 'Run Module' (F5).

Esimerkki 2.1. Muuttujien käyttäminen ja vakioarvot

```
# -*- coding: utf-8 -*-
# Tiedosto: muuttuja.py

luku = 4
print(luku)
luku = luku + 1
luku = luku * 3 + 5
print(luku)

teksti = "Tässä meillä on tekstiä."
print(teksti)
```

Tuloste

```
>>>
4
20
Tässä meillä on tekstiä.
>>>
```

Kuinka se toimii

Ohjelmasi toimii seuraavasti: Ensin me määrittelemme muuttujalle `luku` vakioarvon `4` käyttäen sijoitusoperaattoria (=). Tätä riviä sanotaan käskyksi, koska rivillä on määräys siitä, että jotain pitäisi tehdä. Tässä tapauksessa siis sijoittaa arvo `4` muuttujaan `luku`. Seuraavalla rivillä olevalla toisella käskyllä me tulostamme muuttujan `luku` arvon ruudulle käskyllä `print`. Sijoitusoperaatio toimii loogisesti aina siten, että se arvo, johon sijoitetaan on operaattorin vasemmalla puolella ja se, mitä sijoitetaan oikealla.

Tämän jälkeen me kasvatamme luvun arvoa yhdellä ja tallennamme sen takaisin muuttujaan `luku`. Seuraava rivi tuottaa vielä hieman monimutkaisemman laskennan ja tallentaa tuloksen jälleen muuttujaan `luku`, eli itseensä. Tulostuskäsky `print` antaakin – kuten olettaa saattoi – tällä kertaa arvon 20.

Samalla menetelmällä toiseksi viimeinen rivi tallentaa merkkijonon muuttujaan `teksti` ja viimeinen rivi tulostaa sen ruudulle.

Huomautus muuttujista

Muuttuja otetaan käyttöön ja esitellään samalla kun sille annetaan ensimmäinen arvo. Erillistä esittelyä tai tyypin määrittelyä ei tarvita, eli Pythonille ei tarvitse kertoa tuleeko muuttuja olemaan esimerkiksi kokonaisluku tai merkkijono. Lisäksi muuttujaan, joka sisältää vaikkapa kokonaisluvun voidaan tallentaa tilalle merkkijono. Joissain tapauksissa Pythonin syntaksi tosin vaatii sen, että käytettävä muuttuja on jo aiemmin määritelty joksikin soveltuvaksi arvoksi. Täten emme voi esimerkiksi tulostaa määrittelemättömän muuttujan arvoa.

Kommenttirivit

Kuten aikaisemmin jo mainittiinkin, on Pythonissa lisäksi mahdollisuus kirjoittaa lähdekoodin sekaan vapaamuotoista tekstiä, jolla käyttäjä voi kommentoida tekemäänsä koodia tai kirjoittaa muistiinpanoja, jotta myöhemmin muistaisi, kuinka koodi toimii. Näitä rivejä sanotaan kommenttiriveiksi, ja ne tunnistetaan #-merkistä.

Kun tulkki havaitsee rivillä kommentin alkamista kuvaavan merkin ”#”, lopettaa tulkki kyseisen rivin lukemisen ja siirtyy välittömästi seuraavalle riville. Ainoan poikkeuksen tähän sääntöön tekee ensimmäinen koodirivi, jolla käyttäjä antaa käytettävän aakkoston tiedot. Kannattaa huomata, että #-merkillä voidaan myös rajata kommentti annetun käskyn perään. Tällöin puhutaan koodinsisäisestä kommentoinnista. Lisäksi kommenttimerkkien avulla voimme poistaa suoritettavasta koodista komentoja, joita käytämme esimerkiksi testausvaiheessa välitulosten tarkasteluun. Jos meillä on esimerkiksi seuraavanlainen koodi:

```
# -*- coding: utf-8 -*-

# Tämä on kommenttirivi
# Tämäkin on kommenttirivi
# Kommenttirivejä voi olla vapaa määrä peräkkäin

print("Tulostetaan tämä teksti")
# print("Tämä käsky on kommenttimerkillä poistettu käytöstä.")
print("Tämäkin rivi tulostetaan") # Tämä kommentti jätetään huomioimatta
```

Tulostaisi se seuraavanlaisen tekstin

```
>>>
Tulostetaan tämä teksti
Tämäkin rivi tulostetaan
>>>
```

Käytännössä tulkki siis jättää kommenttirivit huomioimatta. Erityisesti tämä näkyy koodissa toisessa tulostuskäskyssä, jota ei tällä kertaa käydä läpi, koska rivin alkuun on lisätty kommenttimerkki. Vaikka kommentoitu alue sisältäisi toimivaa ja ajettavaa koodia,

ei sitä siitäkään huolimatta suoriteta – se on siis ”kommentoitu pois”.

Loogiset ja fyysiset rivit

Fyysisellä rivillä tarkoitetaan sitä riviä, jonka näet, kun kirjoitat koodia. Looginen rivi taas on se kokonaisuus, minkä Python näkee yhtenä käskynä. Oletusarvoisesti Pythonin tulkki toimiikin siten, että se käsittelee yhtä fyysistä riviä yhtenä loogisena rivinä.

Esimerkiksi käsky `print("Hello World")` on yksi looginen rivi. Koska se myös kirjoitetaan yhdelle riville, on se samalla ainoastaan yksi fyysinen rivi. Yleisesti ottaen Pythonin syntaksi tukee yhden rivin ilmaisutapoja erityisen hyvin, koska se käytännössä pakottaa koodin muotoutumaan helposti luettavaksi sekä selkeästi jaotelluksi.

On mahdollista kirjoittaa useampi looginen rakenne yhdelle riville, mutta **on erittäin suositeltavaa, että kirjoitat koodin sellaiseen muotoon, jossa yksi fyysinen rivi koodia tarkoittaa yhtä loogista käskyä.** Käytä useampaa fyysistä riviä yhtä loogista riviä kohti ainoastaan, mikäli rivistä on tulossa erittäin pitkä. Taka-ajatuksena tässä on se, että koodi pyritään pitämään mahdollisimman helppolukuisena ja yksinkertaisena tulkita tai tarvittaessa korjata.

Alla muutamia esimerkki tilanteista, joissa looginen rivi jakautuu useammalle fyysiselle riville.

```
s = "Tämä on merkkijono. \
Merkkijono jatkuu täällä."
print(s)
```

Tämä tulostaa seuraavan vastauksen:

```
Tämä on merkkijono. Merkkijono jatkuu täällä.
```

Joissain tapauksissa käy myös niin, että et tarvitse kenoviivaa merkitäksesi rivivaihtoa. Näin käy silloin, kun looginen rivi sisältää sulkumerkin, joka loogisesti merkitsee rakenteen määrittelyn olevan kesken. Esimerkiksi listojen tai muiden sarjamuotoisten muuttujien kanssa tämä on hyvinkin tavallista, mutta niistä puhumme enemmän luvussa 8.

Sisennys

Ohjelmia kirjoitettaessa on aikoja jo pähkäilty, kuin koodi saataisiin helppolukuiseksi. Jo mainitut loogiset ja fyysiset rivit ovat yksi asia, jolla koodin lukua helpotetaan: yksi rivi, yksi asia. Toinen asia, jolla koodin luettavuutta ja ylläpidettävyyttä parannetaan, on sisennys.

Pythonissa **välilyönti rivin alussa on merkitsevä merkki**. Tätä sanotaan **sisennykseksi**. Rivin alun tyhjät merkit (välilyönnit ja tabulaattorivälit) määrittelevät loogisen rivin sisennyksen syvyyden, jota taas käytetään, kun määritellään mihin loogiseen joukkoon rivi kuuluu.

Tämä tarkoittaa sitä, että koodirivit, jotka ovat loogisesti samaa ryhmää, on sijoitettava samalle sisennystasolle. Sisennystasoa, joka taas sisältää loogisen Python-käskyn, sanotaan osioksi. Mm. luvuissa 4 ja 5 näemme, kuinka merkitseviä nämä osiot ovat ja kuinka niitä käytetään.

Seuraavassa esimerkissä esittelemme, millaisia ongelmia sisennystason asettaminen väärin useimmiten aiheuttaa:

```
i = 5
  print("Arvo on", i) # Virhe! Huomaa välilyönnit alussa
print("Arvo siis on", i)
```

Jos tallennat tiedoston nimellä whitespace.py ja yrität ajaa sen, saat virheilmoituksen:

```
  File "whitespace.py", line 2
    print("Arvo on", i) # Virhe! Huomaa välilyönnit alussa
    ^
IndentationError: unexpected indent
```

Vaihtoehtoisesti ohjelma saattaa myös antaa “syntax error” -dialogi-ikkunan ja kieltäytyy ajamasta kirjoittamaasi koodia.

Virheen syy on toisen rivin alussa olevassa ylimääräisessä välilyönnissä. *Syntax error* tarkoittaa sitä, että Python-koodissa on jotain, mikä on täysin kieliopin vastaista, eikä tulkki pysty edes sanomaan mitä siitä pitäisi korjata. Tässä tuleva *IndentationError kertoo sisennyksen olevan pielessä.* Käytännössä sinun tulee muistaa tästä osiosta lähinnä se, että **et voi aloittaa satunnaisesta paikasta koodiriviä, vaan sinun on noudatettava sisennyksien ja osioiden kanssa loogista rakennetta**. Rakenteita, joissa luodaan uusia osioita, käsitellään luvusta 4 eteenpäin.

Ohjeita sisennysten hallintaan

Älä käytä välilyöntien ja tabulaattorin sekoitusta sisennysten hallintaan. Tämä voi aiheuttaa ongelmia, mikäli siirryt editoriohjelmasta toiseen, koska ohjelmat eivät välttämättä käsittele tabulaattoria samalla tavoin. Suositeltavaa on, että käytät editoria, joka laskee yhden tabulaattorimerkin neljäksi välilyönniksi – kuten esimerkiksi IDLE tekee - ja käytät sisennyksissä ainoastaan tabulaattorimerkin pituisia välejä tasolta toiselle siirryttäessä.

Tärkeintä sisennysten hallinnassa on se, että valitset itsellesi luontaisen tyylin, joka toimii ja pysyt siinä. Ole johdonmukainen sisennystesi kanssa ja pysyttele vain ja ainoastaan yhdessä tyylissä.

Yhteenveto tähän mennessä opitusta

Nyt olemme käyneet läpi tärkeimmät yksityiskohdat Pythonin kanssa operoinnista ja voimme siirtyä eteenpäin mielenkiintoisempiin aiheisiin. Seuraavaksi tutustumme yleisimpiin operaattoreihin, jotka liittyvät muuttujilla operointiin. Aikaisemmin olet jo tutustunut niistä muutamaan, (+) (*) ja (=) -operaattoriin.

Operaattorit ja operandit

Useimmat kirjoittamasi käskyt sisältävät jonkinlaisen operaattorin. Yksinkertaisimmillaan tämä voi tarkoittaa vaikka riviä `2 + 3`. Tässä tapauksessa + edustaa lauseen operaattoria, ja `2` ja `3` lauseen operandeja.

Voit kokeilla operaattorien toimintaa käytännössä syöttämällä niitä Python interaktiiviseen ikkunaan ja katsomalla, minkälaisia tuloksia saat aikaiseksi. Esimerkiksi yhteen- ja kertolaskuoperaattorit toimisivat näin:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Taulukko 2.1 Laskentaoperaattorit

Operaattori	Nimi	Selite	Esimerkki
=	Sijoitus	Sijoittaa annetun arvon kohdemuuttujalle	`luku = 5` sijoittaa muuttujalle luku arvon 5. Operaattori toimii ainoastaan mikäli kohteena on muuttuja.
+	Yhteen	Laskee yhteen kaksi operandia	`3 + 5` antaa arvon `8`. `'a' + 'b'` antaa arvon `'ab'`.
-	Vähennys	Palauttaa joko negatiivisen arvon tai vähentää kaksi operandia toisistaan	`-5.2` palauttaa negatiivisen numeron. `50 - 24` antaa arvon `26`.
*	Tulo	Palauttaa kahden operandin tulon tai toistaa merkkijonon operandin kertaa	`2 * 3` antaa arvon `6`. `'la' * 3` antaa arvon `'lalala'`.
**	Potenssi	Palauttaa x:n potenssin y:stä.	`3 ** 4` antaa arvon `81` (eli. `3 * 3 * 3 * 3`)
/	Jako	Jakaa x:n y:llä	`4/3` antaa arvon `1.3333333333333333`
//	Tasajako	Palauttaa tiedon kuinka monesti y menee x:ään	`4 // 3` antaa arvon `1`
%	Jakojäännös	Palauttaa x:n jakojäännöksen y:stä.	`8%3` antaa `2`. `-25.5%2.25` antaa `1.5`.

Laskentajärjestyksestä

Oletusarvoisesti suoritusjärjestys toteutuu matemaattisten laskusääntöjen mukaisesti. Kuten normaalisti matematiikassa, voit lisäksi muuttaa järjestystä käyttämällä sulkuja. Suluilla työskennelläänkin täysin samalla tavoin kuin matemaattisesti laskettaessa, sisimmistä ulospäin ja samalla tasolla laskentajärjestyksen mukaisesti. Esimerkiksi, jos haluat, että yhteenlasku toteutetaan ennen kertolaskua, merkitään se yksinkertaisesti `(2 + 3) * 4`.

Lausekkeet

Esimerkki 2.2. Lausekkeiden käyttö

```
# -*- coding: utf-8 -*-
# Tiedosto: lauseke.py

pituus = 5
leveys = 2

pinta = pituus * leveys
print("Nelikulmion pinta-ala on", pinta)
print("ja kehä on", 2 * (pituus + leveys))
```

Tuloste

```
>>>
Nelikulmion pinta-ala on 10
ja kehä on 14
>>>
```

Kuinka se toimii

Kappaleen pituus ja leveys on tallennettu samannimisiin muuttujiin. Me käytämme näitä muuttujia laskeaksemme kappaleen pinta-alan ja kehän pituuden operandien avulla. Ensin suoritamme laskutoimituksen pinta-alalle ja sijoitamme operaation tuloksen muuttujaan `pinta`. Seuraavalla rivillä tulostamme vastauksen tuttuun tapaan. Toisessa tulostuksessa laskemme vastauksen `2 * (pituus + leveys)` suoraan `print`-funktion sisällä.

Huomioi myös, kuinka Python automaattisesti laittaa tulostuksissa välilyönnin tulostettavan merkkijonon `'Pinta-ala on'` ja muuttujan `pinta` väliin. Tämä on yksi Pythonin erityispiirteistä, jotka tähtäävät ohjelmoijan työmäärän vähentämiseen.

Tiedon lukeminen käyttäjältä

Varsin nopeasti huomaat kuitenkin, että etukäteen luodut merkkijonot taikka koodiin määritellyt muuttujanarvot eivät pysty hoitamaan kaikkia tehtäviä kunnolla. Haluamme päästä syöttämään arvoja ajon aikana, sekä antaa käyttäjälle mahdollisuuden vaikuttaa omaan koodiinsa. Seuraavaksi käymmekin läpi keinoja, kuinka Python osaa pyytää käyttäjältä syötteitä ja tallentaa niitä muuttujiin.

Esimerkki 2.2. Arvojen vastaanottaminen käyttäjältä

```
# -*- coding: utf-8 -*-
# Tiedosto: lauseke2.py

x = input("Anna ympyrän säde: ")
sade = int(x)
pii = 3.14

ala = sade * sade * pii
print("Pinta-ala on", ala)
print("Kehä on", 2 * sade * pii)
```

Tuloste

```
>>>
Anna ympyrän säde: 3
Pinta-ala on 28.26
Kehä on 18.84
>>>
```

Kuinka se toimii

Ensimmäisellä rivillä sijoitamme muuttujalle `x` arvoksi `input`-funktion tuloksen. `input`-funktio saa syötteenä merkkijonon, kuten nyt ”Anna kappaleen pituus: ”, ja tulostaa tämän merkkijonon ruudulle toimintaohjeeksi käyttäjälle. Käyttäjä syöttää mieleisensä luvun vastauksena kehotteelle, jonka jälkeen annettu luku tallentuu muuttujaan `x`. Pythonin `input`-funktio ei kuitenkaan tiedä, että käyttäjä syöttää nimenomaan numeron, vaan se ottaa vastaa merkkijonon, tässä tapauksessa merkin ”3”. Laskussa tarvitsemme kuitenkin numeroa, joten muutamme `int`-funktiolla käyttäjän syöttämän merkkijonon numeroksi. Tämä jälkeen voimme laskea ja tulostaa pinta-alan ja kehän pituuden edellisen esimerkin tavoin.

Esimerkki 2.3. input-funktion käyttäminen, osa 2

```
# -*- coding: utf-8 -*-
# Tiedosto: input.py

sana = input("Anna merkkijono: ")
print("Annoit sanan", sana)
```

Tuloste

```
>>>
Anna merkkijono: Kumiankka
Annoit sanan Kumiankka
>>>
```

Kuinka se toimii

Tämä esimerkki ei juurikaan eroa edellisestä. Otimme vain vastaan yhden merkkijonon ja tulostimme sen ruudulle. Koska mitään numeraalista laskentaa ei tarvita (eikä sitä kumiankalla pystyttäisi tekemäänkään), ei myöskään `int`-funktiolle ole käyttöä.

Tyyppimuunnoksista ja merkkijonoilla operoinnista puhumme enemmän luvussa 3. Siihen asti riittää, että tiedät, kuinka `input`-funktiolla voidaan ottaa käyttäjältä syötteinä lukuja ja merkkijonoja.

Pythonin käyttö laskimena

Tulkkia voi käyttää myös kuten yksinkertaista laskinta. Annat tulkille operandit ja operaattorin ja tulkki tulostaa sinulle vastauksen. Voit myös kokeilla IDLEn interaktiivisen ikkunan avulla yksinkertaisia yhdistelmiä ja rakenteita. Esimerkiksi tulkkia voidaan käyttää seuraavilla tavoilla:

```
>>> 2+2
4
>>> # Kommenttirivi
... 2+2
4
>>> 2+2  # Kommenttirivi ei sotke koodia
4
>>> (50-5*6)/4
5
>>> # Jakolasku tuottaa usein desimaaliluvun
... 7/3
2.3333333333333335
>>> 7%3
1
```

Yhtäsuuruusmerkki (‘=’) toimii sijoitusoperaattorina. Sillä voit syöttää vakiotietoja, joilla voit suorittaa laskutehtäviä. Sijoitusoperaattoriin päättyvä lauseke ei tuota välitulosta:

```
>>> leveys = 20
>>> korkeus = 5*9
>>> leveys * korkeus
900
```

Kokonaislukuja ja liukulukuja (eli desimaalilukuja) voidaan käyttää vapaasti ristiin. Python suorittaa automaattisesti muunnokset sopivimpaan yhteiseen muotoon:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Muutama sana muuttujien rooleista

Olet nyt törmännyt sanaan *muuttuja*. Nimensä mukaisesti muuttuja voi sisältää tietoa, joka voi *muuttua*. Muuttujilla on kuitenkin erilaisia käyttötarkoituksia. Tähän mennessä olemme törmänneet *kiintoarvoon*, eli muuttujaan, joka saa kerran sisältönsä ja pitää tämän sisällön sisällään, eikä muuta sitä missään vaiheessa.

```
>>>
nimi = input("Anna nimi: ")
print("Nimesi on", nimi)
>>>
```

Tässä esimerkissä muuttuja `nimi` saa arvokseen käyttäjän syöttämän nimen. Muuttuja `nimi` on *kiintoarvo*, koska sitä ei muuteta ohjelman ajon aikana. Sitä käytetään kyllä

tulostuksessa, mutta arvo pysyy kiinteänä koko ohjelman suorituksen ajan.

Toinen rooli johon törmäsimme on *tilapäissäilö*, jona `x` toimi esimerkissä 2.2. Tilapäissäilön tarkoitus on toimia nimensä mukaisesti tilapäisenä säilönä. Tällaisen muuttujan ”elinaika” on hyvin lyhyt, eikä sitä käytetä (samassa tarkoituksessa) enää myöhemmin ohjelmassa. Luvun kokoavassa esimerkissä 2.4 on esitetty myös tapa, jolla tilapäissäilöä ei tarvitse käyttää ollenkaan.

Tulevissa luvuissa törmäämme vielä muutamaan muuhunkin rooliin. Näistä on huomautus aina, kun uusi rooli esitellään.

Luvun asiat kokoava esimerkki

Esimerkki 2.4. Leivontaohjeen rakennus

```
# -*- coding: utf-8 -*-
# Tiedosto: jauhopeukalo2.py

ohje = input("Mikä ohje tämä on? ")
x = input("Kuinka monta desiä vettä? ")
vesi = int(x)
x = input("Kuinka monta grammaa hiivaa? ")
hiiva = int(x)
# Huomaa, että seuraavassa input() on laitettu suoraan int():n sisään.
jauho = int(input("Kuinka monta desiä jauhoja? "))

print(ohje)
print("===========")
print("Vettä:       ", vesi, "dl")
print("Hiivaa:      ", hiiva, "g")
print("Vehnäjauhoja:", jauho, "dl")
```

Luku 3: Merkkijonot ja niiden kanssa työskentely

Merkkijono on jono peräkkäisiä merkkejä. Merkkijonot voivat olla esimerkiksi sanoja tai lauseita, mutta varsinaisesti merkkijonoksi lasketaan mikä tahansa joukko merkkejä.

Luultavasti tulet käyttämään merkkijonoja usein, joten seuraava osio kannattaa lukea ajatuksella lävitse. Pythonissa merkkijonoja voidaan käyttää seuraavilla tavoilla:

Käyttäen sitaattimerkkiä (')

Voit määritellä merkkijonoja käyttäen sitaatteja; esimerkiksi näin: `'Luota minuun tässä asiassa.'`. Kaikki ei-näkyvät merkit kuten välilyönnit tai sisennykset tallentuvat kuten tulostus näyttää ne, eli omille paikoilleen.

Käyttäen lainausmerkkiä (”)

Lainausmerkki (”) toimii samalla tavoin kuin sitaattimerkki. Tässäkin tapauksessa kahden merkin väliin jäävä osa luetaan merkkijonona, esimerkiksi: `"Elämme kovia aikoja ystävä hyvä"`. Pythonin kieliopin kannalta sitaatti- ja lainausmerkillä ei ole minkäänlaista eroa, joskaan ne eivät toimi keskenään ristiin. Tämä siis tarkoittaa sitä, että esimerkiksi `"Tämä on yritelmä'` ei olisi kelvollinen merkkijono vaikka se teknisesti onkin oikeiden merkkien rajoittama.

Ohjausmerkit

Oletetaan, että haluat käyttää merkkijonoa, joka sisältää sitaattimerkin ('). Kuinka pystyisit käyttämään sitä ilman, että Pythonin tulkki aiheuttaa ongelmia? Esimerkiksi voidaan ottaa vaikka merkkijono `vaa'an alla`. Et voi määritellä merkkijonoa tyyliin `'vaa'an alla'`, koska silloin tulkki ei tiedä mihin sitaattimerkkiin merkkijonon olisi tarkoitus päättyä. Tässä tilanteessa joudut jotenkin kertomaan tulkille, mihin sitaattimerkkiin tulee lopettaa. Tarvitset siis ohjausmerkkiä (\), jonka avulla voit merkata yksinkertaisen sitaattimerkin ohitettavaksi tyyliin \'. Nyt esimerkkirivi `'vaa\'an alla'` toimisi ilman ongelmia.

Toinen vaihtoehto olisi tietenkin käyttää lainausmerkkiä, jolloin esittely `"vaa'an alla"` toimii ongelmitta. Tämä tietenkin toimii myös toisin päin, jolloin tekstiin kuuluvan lainausmerkin voi merkata ohjausmerkillä (\) tai koko rivin määritellä sitaateilla. Samoin itse kenoviivan merkkaamiseen käytetään ohitusmerkkiä, jolloin merkintä tulee näin \\.

Entä jos haluat tulostaa useammalle riville? Voit käyttää rivinvaihtomerkkiä (\n). Rivinvaihtomerkki tulee näkyviin tekstiin normaalisti kauttaviiva-n-yhdistelmänä, mutta tulkissa tulostuu rivinvaihtona. Esimerkiksi `"Tämä tulee ensimmäiselle riville. \n Tämä tulee toiselle riville."` Toinen vastaava hyödyllinen merkki on sisennysmerkki (\t), joka vastaa

tabulaattorimerkkiä ja jolla voimme tasata kappaleiden reunoja. Ohjausmerkeistä on hyvä tietää lisäksi se, että yksittäinen kenoviiva rivin päässä tarkoittaa sitä, että merkkijono jatkuu seuraavalla rivillä. Tämä aiheuttaa sen, että tulkki ei lisää rivin päähän rivinvaihtoa vaan jatkaa tulostusta samalle riville. Esimerkiksi,

```
"Tämä on ensimmäinen rivi joka tulostuu.\
Tämä tulee ensimmäisen rivin jälkeen."
```

On sama kuin `"Tämä on ensimmäinen rivi joka tulostuu. Tämä tulee ensimmäisen rivin jälkeen. "`

Täydellinen lista ohjausmerkeistä löytyy mm. Python Software Foundationin dokumenteista, jotka löytyvät osoitteesta www.python.org.

Merkkijonojen yhdistäminen

Jos laitat kaksi merkkijonoa vierekkäin, Python yhdistää ne automaattisesti. Esimerkiksi merkkijonot `'Vaa\'an' 'alunen'` yhdistyy tulkin tulostuksessa merkkijonoksi `"Vaa'an alunen"`.

Esimerkkejä

Kuten varmaan olet jo huomannut, Python osaa numeroiden lisäksi operoida myös merkkijonoilla, jotka määritellään sitaateilla. Seuraavaksi tutustumme hieman tarkemmin niiden kanssa työskentelemiseen:

```
>>> 'kinkkumunakas'
'kinkkumunakas'
>>> 'vaa\'an'
"vaa'an"
>>> "raa'at"
"raa'at"
>>> '"Kyllä," hän sanoi.'
'"Kyllä," hän sanoi.'
>>> "\"Kyllä,\" hän sanoi."
'"Kyllä," hän sanoi.'
>>> '"Vaa\'an alla," mies sanoi.'
'"Vaa\'an alla," mies sanoi.'
```

Lisää merkkijonoista: yhdistelyt ja leikkaukset

Merkkijonoja voidaan yhdistellä ”+” -operaattorilla ja toistaa ”*”-operaattorilla:

```
>>> sana1 = "Ko"
>>> sana2 = "ralli"
>>> sana1 + sana2
'Koralli'
>>> (sana1 + "-") * 3 + sana1 + sana2
'Ko-Ko-Ko-Koralli'
```

Jos haluamme päästä käsiksi merkkijonon sisällä oleviin merkkeihin, voimme ottaa merkkijonosta leikkauksia. Leikkauksen ala määritellään hakasuluilla ja numerosarjalla, jossa ensimmäinen numero kertoo aloituspaikan, toinen leikkauksen lopetuspaikan ja kolmas siirtymävälin. Kaikissa tilanteissa näitä kaikkia ei ole pakko käyttää. Kannattaa myös muistaa, että merkkijonon numerointi alkaa luvusta 0. Hakasulkujen sisällä numerot erotellaan toisistaan kaksoispisteillä:

```
>>> sana = "Teekkari"
>>> sana[3]
'k'
>>> sana[0:3]
'Tee'
>>> sana[4:8]
'kari'
>>> sana[0:8:2]
'Tekr'
```

Indeksointi alkaa nollasta

Pythonissa siis indeksointi alkaa nollasta (tästä syystä tässäkin kirjassa on Luku 0). Syitä tähän on monia. Esimerkiksi voidaan ajatella, että indeksi kertoo siirtymän sanan alusta ja koska sanan ensimmäinen merkki on jo alussa, on siirtymä 0, joten indeksi on myös tuo sama 0. Syytä on perusteltu myös matemaattisella kauneudella, nolla on pienin ei-negatiivinen kokonaisluku. Toisaalta, on myös ohjelmointikieliä, joissa indeksointi ei ala nollasta...

Leikkaus sisältää hyödyllisiä ominaisuuksia. Lukuarvoja ei aina myöskään ole pakko käyttää; ensimmäisen luvun oletusarvo on 0, ja toisen luvun oletusarvo viimeinen merkki. Siirtymävälin oletusarvo on 1.

```
>>> sana[:2]    # Ensimmäiset kaksi kirjainta
'Te'
>>> sana[2:]    # Kaikki muut kirjaimet paitsi kaksi ensimmäistä
'ekkari'
```

Lisäksi tulee muistaa, että Pythonissa merkkijonojen muuttelussa on jonkin verran rajoituksia, koska merkkijono on vakiotietotyyppi:

```
>>> sana[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support item assignment

>>> sana[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support slice assignment
```

Tämä ei kuitenkaan aiheuta ongelmaa, koska merkkijonon voi määritellä kokonaan uudelleen leikkausten avulla helposti:

```
>>> 'P' + sana[1:]
'Peekkari'
>>> sana[:3] + "hetki"
'Teehetki'
```

Tai vaihtoehtoisesti

```
>>> sana = "kumiankka"
>>> sana = "testi" + sana
>>> sana
'testikumiankka'
>>>
```

Huomaa myös, että leikkaus ”`s[:i] + s[i:]`” on sama kuin `s`.

```
>>> sana = "Teekkari"
>>> sana[:2] + sana[2:]
'Teekkari'
>>> sana[:3] + sana[3:]
'Teekkari'
```

Myös merkkijonoalueen yli meneviä leikkauksia kohdellaan hienovaraisesti. Jos annettu numeroarvo ylittää merkkijonon rajat tai aloituspaikka on lopetuspaikkaa suurempi, tulee vastaukseksi tyhjä jono:

```
>>> sana[1:100]
'eekkari'
>>> sana[10:]
''
>>> sana[2:1]
''
```

Tämä ei kuitenkaan koske tilannetta, jossa pyydetään merkkijonosta yksittäistä merkkiä sen sijaan, että otetaan leikkaus:

```
>>> sana[100]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    sana[111]
IndexError: string index out of range
>>>
```

Leikkauksissa voidaan myös käyttää negatiivisia lukuja. Nämä luvut lasketaan oikealta vasemmalle, eli siis lopusta alkuun päin. Esimerkiksi:

```
>>> sana[-1]     # Viimeinen merkki
'i'
>>> sana[-2]     # Toiseksi viimeinen merkki
'r'

>>> sana[-2:]    # Viimeiset kaksi merkkiä
'ri'
>>> sana[:-2]    # Muut paitsi viimeiset kaksi merkkiä
'Teekka'
```

Lisäksi negatiivisia arvoja voidaan käyttää siirtymävälinä, jos halutaan liikkua merkkijonossa lopusta alkuun päin:

```
>>> sana = "Robottikana"
>>> sana[::-1]    # Sana käännettynä ympäri
'anakittoboR'
>>> sana[::-2]    # Joka toinen kirjain
'aaitbR'
```

Huomioi kuitenkin, että arvo -0 ei viittaa viimeisen merkin taakse, vaan että -0 on sama kuin 0

```
>>> testi = "Kumiankka"
>>> testi[-0]     # (koska -0 on sama kuin 0)
'K'
```

Paras tapa muistaa miten merkkijonon numeroiden leikkaukset lasketaan, on ajatella numeroiden sijaan niiden välejä:

```
 +---+---+---+---+---+
 | A | p | u | V | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Ylempi numerorivi kertoo sijainnin laskettuna normaalisti vasemmalta oikealle. Alempi rivi taas negatiivisilla luvuilla laskettuna oikealta vasemmalle. Huomaa edelleen, että -0 ei ole olemassa.

Jos taas haluat selvittää yleisesti ottaen merkkijonon pituuden, voit käyttää siihen Pythonin sisäänrakennettua funktiota `len`. Funktiolle annetaan syötteenä muuttuja tai merkkijono, ja se palauttaa sen pituuden merkkeinä:

```
>>> s ='Apumiehensijaisentuuraajankorvaajanlomittajanpaikka'
>>> len(s)
51
```

Huomaa kuitenkin, että viimeinen merkki on paikalla `s[50]`, johtuen siitä että funktio `len` palauttaa todellisen pituuden merkkeinä, ja merkkijonon ensimmäisen merkin järjestysnumero on 0.

Tyyppimuunnokset ja pyöristys

Pythonin mukana tulee sisäänrakennettuna funktiot, joiden avulla muuttujan tyyppi voidaan vaihtaa, jos se on yksikäsitteisesti mahdollista toteuttaa. Esimerkiksi `int` muuttaa annetun syötteen kokonaisluvuksi ja `str` merkkijonoksi. Lisäksi liukuluvulle on olemassa oma tyyppimuunnosfunktio:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int() with base 10: 'hello'
```

`Int` voi muuttaa liukulukuja kokonaisluvuiksi, mutta ei osaa pyöristää niitä. Tämä siis tarkoittaa käytännössä sitä, että `int` ainoastaan katkaisee luvun desimaaliosan pois. Lisäksi `int` osaa myös muuttaa soveltuvat merkkijonot (käytännössä numerojonot) kokonaisluvuiksi:

```
>>> int(3.99999)
3
>>> int("-267")
-267
```

`float` muuttaa kokonaislukuja ja numeerisia merkkijonoja liukuluvuiksi:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

`str` muuttaa annettuja syötteitä merkkijonoiksi:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Tyyppimuunnoksiin hyvin läheisesti liittyy myös lukujen pyöristys. Käyttäjälle ei kovinkaan usein kannata printata pitkää desimaalijonoa ruudulle, jos pienempi määrä on informatiivisempi. Esimerkit selventävät asiaa.

```
>>> luku = 234.5647292340234
>>> round(luku)
235
>>> round(luku, 3)
234.565
>>>
```

Esimerkissämme määrittelemme muuttujan `luku` ja sijoitamme siihen paljon desimaaleja sisältävän liukuluvun. Tämän jälkeen annamme sen `round`ille, joka pyöristää siitä kokonaisluvun. Jos annamme `round`ille toisen parametrin (parametreistä ja funktiosta lisää luvussa 6) 3, pyöristää `round` luvun kolmen desimaalin tarkkuudella. Toinen esimerkki selventää lisää.

```
>>> a = 3.14
>>> b = 1043.55
>>> b / a
332.34076433121015
>>> round((b / a), 2)
332.34
>>>
```

Tulee kuitenkin huomata, että `round` ei lisää lukuun (turhia) desimaaleja, vaan pyöristyksessä näytetään vain merkitsevät numerot. Eli `round(3.10000, 3)` on yhtä kuin *3.1*. Tällainen pyöristys on käytössä Pythonin versioilla 3.1.2 ja aiemmilla. Tulevaisuudessa asia voi tietysti muuttua, mikäli muutos koetaan tarpeelliseksi. Mikäli lukuun halutaan tasamäärä numeroita, täytyy käyttää muotoiltua tulostusta, mutta tästä puhumme vasta luvussa 7.

Huomautus tyyppimuunnoksista

Lue tämä kappale huolella! Tyyppimuunnoksiin liittyy kuitenkin yksi tekninen yksityiskohta, joka tulee huomioida niitä käytettäessä:

```
>>> luku = "2323" # Alustetaan luku merkkijonona
>>> int(luku)
2323
>>> luku +1 # Muuttuja ei tyyppimuunnoksesta huolimatta ole numero, \
              koska sitä ei tallennettu mihinkään.
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in -toplevel-
    luku +1
TypeError: Can't convert 'int' object to str implicitly
>>> luku = int(luku) # Tallennetaan muutos, jolloin luku todella on \
                       numero
>>> luku + 1 # Nyt lukuun voidaan lisätä
2324
>>>
```

Tyyppimuunnosfunktiot eivät muuta alkuperäistä annettua arvoa, vaan tuottavat annetusta arvosta uuden, muunnetun tuloksen. Tämä siis tarkoittaa sitä, että mikäli haluat käyttää tyyppimuunnettua arvoa, joudut tallettamaan sen muuttujaan sijoitusoperaattorilla (”=”).

Print-funktion muita ominaisuuksia

Printillä voidaan tulostaa, kuten on tähän mennessä opittu, mutta tämän lisäksi on muutama muukin tapa, joilla tekstiä voidaan tulostaa monipuolisemmin.

Esimerkki 3.1. Printtailua

```
# -*- coding: utf-8 -*-
# Tiedosto: printteri.py

print("Tekstiä")
nimi = "Maija"
print("Nimi on", nimi)
print("Nimi on " + nimi + ", joka onkin hyvä nimi.")
print("Tekstiä on tässä. ", end="")
print("Tämä jatkuu samalle riville.")
```

Tuloste

```
>>>
Tekstiä
Nimi on Maija
Nimi on Maija, joka onkin hyvä nimi.
Tekstiä on tässä. Tämä jatkuu samalle riville.
>>>
```

Kuinka se toimii

Ensimmäinen printti on jo opittu, eikä seuraavakaan tuo mitään uutta. Kolmannessa printissä korvaamme pilkut ”+”-merkeillä, jolloin muuttujan ympärille ei tule automaattista välilyöntiä ja saamme pilkun heti nimen Maija perään, koska käsittelemme nyt merkkijonoja ja niiden ”yhteenlasku” tarkoittaa niiden liittämistä yhteen.

Neljännessä printissä määritämme normaalin tekstin lisäksi avainsanalla `end` `print`-funktiolle, että rivin loppuun tuleva merkki on tyhjä ts. sitä ei siis ole. Oletuksena käytössä oleva rivinvaihto (”\n”) siis poistuu ja seuraava rivi tulostuukin edellisen perään. `end`-avainsanalla voidaan määrittää mikä tahansa merkkijono rivin loppuun, myös nyt käytössä oleva tyhjä merkkijono.

Luvun asiat kokoava esimerkki

Esimerkki 3.2. Jalojuoma

```
# -*- coding: utf-8 -*-
# Tiedosto: jalojuoma.py

sana = input("Anna sana: ")
x = input("Monennenko merkin kohdalta haluat katkaista sanan? ")
luku = int(x) # muutetaan numeroksi

print("Sana:", sana, "on katkaistuna", sana[0:luku]) #leikataan loppu

print("Sinulla on 10 litraa vettä ja kilo hiivaa ja toinen kilo
sokeria.")
x = input("Anna hiivan tehokkuuskerroin? ")
kerroin = float(x)
tulos = 10 * 1 * 1 + kerroin / 1.3
tulos = round(tulos, 2)
# tehdään lasku ja tulostetaan tulos kahden desimaalin tarkkuudella
print("Kertoimella", kerroin, "valmistuu", tulos, "litraa simaa.")
```

Luku 4: Valintarakenteet

Tähän asti olemme tehneet ohjelmia, jotka ovat olleet joukko peräkkäisiä käskyjä, jotka suoritetaan aina samassa järjestyksessä. Entäpä, jos haluaisimme koodin tekevän vertailuja tai laittaa mukaan osioita, jotka ajetaan ainoastaan mikäli niitä tarvitaan? Esimerkiksi, miten toimisimme, jos haluaisimme ohjelman, joka sanoo ”Hyvää huomenta” tai ”Hyvää iltaa” kellonajan mukaisesti?

Kuten varmaan arvaat, tarvitaan tässä vaiheessa koodin tekemiseen ohjausrakenteita. Python käyttää kolmea ohjausrakennetta, `if`, `for` ja `while`, joista tässä luvussa tutustutaan ensimmäiseen, if-rakenteeseen, jonka avulla voimme luoda ”ehdollista” koodia.

If-valintarakenne

`If-else`-rakenne perustuu koodille annettavaan loogiseen väittämään. Tämän väittämän ollessa totta (`True`) ajetaan se koodin osio, joka on liitetty `if`-lauseeseen. Muussa tapauksessa ajetaan `else`-rakenteen osio, tai `else`-rakenteen puuttuessa jatketaan samalla tasolla eteenpäin. Rakenne tuntee myös `elif` (else if)-osiot, joilla useita `if`-lauseita voidaan ketjuttaa peräkkäin siten, että voidaan testata useita eri vaihtoehtoja samassa rakenteessa. `Elif`-osioita voi `if`-rakenteessa olla mielivaltainen määrä. Lisäksi myös `else`-rakenne on vapaaehtoinen, mutta kumpaakaan ei voi olla olemassa ilman `if`-osiota, joita voi olla ainoastaan yksi per rakenne. Kuulostaa vaikealta? Ei hätää, seuraava esimerkki helpottaa ymmärtämistä.

Esimerkki 4.1. Käytetään if -rakennetta

```
# -*- coding: utf-8 -*-
# Tiedosto: if.py

print("Tervetuloa ohjelmaan!")
print() # Tulostetaan tyhjä rivi
x = input("Anna kellonaika: ")
kello = int(x)

if kello < 7:
    print("Zzz...")
elif kello <= 8:
    print("Aika nousta luennolle.")
elif kello <= 12:
    print("Aamuluennot menivät jo, mutta vielä kerkeää iltapäivän
opiskella.")
else:
    print("Taitaa olla parempi pitää rokulipäivä... Zzz...")

print() # Toinen tyhjä rivi
print("Kiitos käynnistä!")
```

Tuloste

```
>>>
Tervetuloa ohjelmaan!

Mitä kello näyttää? 8
Aika nousta luennolle.

Kiitos käynnistä!
>>>
Tervetuloa ohjelmaan!

Mitä kello näyttää? 15
Taitaa olla parempi pitää rokulipäivä... Zzz...

Kiitos käynnistä!
>>>
```

Kuinka se toimii

Ohjelman rakenne on yksinkertainen. Ensin kysymme käyttäjältä, mitä kello näyttää ja tämän jälkeen muutamme käyttäjän syötteen kokonaisluvuksi, jotta voimme ohjelman myöhemmässä vaiheessa työskennellä numeroiden kanssa helposti.

Seuraavassa vaiheessa ohjelma siirtyy `if`-rakenteeseen ja testaa, onko käyttäjän syöttämä kellonaika vähemmän kuin 7. Mikäli on, ohjelma siirtyy seuraavan rivin sisennettyyn osioon ja tulostaa tekstin ”Zzz...”. Jos taasen käyttäjän syöttämä kellonaika on enemmän tai yhtä suuri kuin seitsemän mennään seuraavaan vertailuun, jossa tarkistetaan, onko syöte kahdeksan tai vähemmän. Jos näin on, siirrytään taas sisennettyyn koodiosioon ja tulostetaan teksti ”Aika nousta luennolle.” Huomaa, että tähän kohtaan ei tulla, jos käyttäjä on syöttänyt arvon 6, vaikka tämäkin arvo on alle 8, niin se on käsitelty jo ensimmäisessä vaiheessa. Näiden jälkeen on vielä tarkistus, onko syöte vähemmän tai yhtä suuri kuin 12 ja viimeisimpänä `else`-osio, joka toteutuu, jos mikään edellisistä ei ole vielä toteutunut, eli käyttäjä on syöttänyt suuremman luvun kuin 12. Tällöin siirrytään taas sisennettyyn koodiosaan ja tulostetaan ”Taitaa olla parempi pitää rokulipäivä... Zzz...”. Lopuksi vielä, käyttäjän syötteestä riippumatta, tulostetaan ”Kiitos käynnistä!”.

Kannattaa tietysti myös muistaa, että `if`-rakenteita voi olla myös sisäkkäin, `if`-osio, `elif`-osiot tai `else`-osio voivat kaikki sisältää vapaasti lisää `if`−rakenteita. Myös se, että `elif` ja `else`-rakenteiden käyttö on vapaavalintaista, on hyvä muistaa. Yksinkertaisin `if`−rakenne voidaan toteuttaa vaikka näin:

```
if True:
      print("Kyllä, arvo True on aina totta.")
```

Erityisesti `if`-rakenne vaatii tarkkuutta siinä, että muistat merkitä `if`-, `elif`- ja `else`-rakenteiden perään kaksoispisteen, jotta Python-tulkki tietää odottaa uuden osion alkavan siitä. Lisäksi sisennykset vaativat varsinkin alussa tarkkuutta. Jotta `if`-rakenteen käyttäminen ei jäisi epäselväksi, otamme vielä toisen esimerkin.

Esimerkki 4.2. If-elif-else-rakenne, kertaus

```
# -*- coding: utf-8 -*-
# Tiedosto: if2.py

nimi = input("Anna nimesi: ")

if nimi == "Matti":
    print("Mitäs Masa?")
elif nimi == "Karoliina":
    print("Mitäs Karkille kuuluu?")
else:
    print("En tunnista nimeäsi.")
```

Kuinka se toimii

Tässä esimerkikssä kysymme ensin käyttäjän nimeä ja käytämme tämän jälkeen `if-elif-else`-rakennetta kertomaan tervehdyksen käyttäjälle. Mikäli käyttäjä on antanut nimekseen ”Matti”, saa hän tervehdykseksi tekstin ”Mitäs Masa?”. Jos taasen nimeksi on annettu ”Karoliina”, tulee ruudulle teksti ”Mitäs Karkille kuuluu?”. Mikäli käyttäjä on syöttänyt minkä tahansa muun nimen (esim. *kana*, *Gee7ghfF* tai *32*), tulostuu ruudulle viesti ”En tunnista nimeäsi”.

Tästä esimerkistä viimeistään kannattaa hahmottaa, että `if-elif-else`-rakenteesta vain yksi sisennetty koodilohko suoritetaan. Mikäli nimeksi havaitaan ”*Matti*” ei seuraavia `elif`-osioita tai `else`-osiota suoriteta – koskaan.

Toinen asia, johon kannatta jo tässä vaiheessa kiinnittää huomiota, on yhtäsuuruusvertailuoperaattorin (”==”) ja sijoitusoperaattorin (”=”) ero. **Vertailua ei siis koskaan toteuteta vain yhdellä yhtäsuuruusmerkillä!** Tulkki kyllä huomauttaa, jos tällaista yritetään.

Huomioita if-rakenteesta

Kuten aiemmin mainittiin, ei `if`-rakenteessa ole välttämätöntä käyttää aina muotoa `if-elif-else`. Rakenteellisesti helpoin vaihtoehto on käyttää pelkkää `if`-osiota:

```
palkinto = "kolmipyörä"
if palkinto == "kolmipyörä":
    print("Otit kolmipyörän.")
```

Tulostaisi vastauksen:

```
Otit kolmipyörän.
```

Vastaavasti, jos haluaisimme testata tapahtuiko jotain ja ilmoittaa käyttäjälle myös negatiivisesta testituloksesta, voisimme lisätä rakenteeseen pelkän `else`-osion:

```
palkinto = "kolmipyörä"
if palkinto == "rahapalkinto":
    print("Otit rahat.")
else:
    print("Et ottanut rahapalkintoa.")
```

Tulostaisi vastauksen:

```
Et ottanut rahapalkintoa.
```

Huomaa tässä tapauksessa, että emme edelleenkään varsinaisesti tiedä muuttujan `palkinto` sisältöä, mutta tiedämme sen, että se ei ole ”rahapalkinto”. Kolmas tapa, jolla voimme `if`-rakennetta käyttää, on ilman `else`-osiota:

```
palkinto = "kolmipyörä"
if palkinto == "rahapalkinto":
    print("Otit rahat.")
elif palkinto == "kolmipyörä":
    print("Otit kolmipyörän.")
```

Tulostaisi vastauksen:

```
Otit kolmipyörän.
```

Tässä yhteydessä emme tarvitse `else`-osiota mihinkään - oletetaan vaikka, että ainoat meitä kiinnostavat vaihtoehdot ovat rahat tai kolmipyörä - joten voimme jättää `else`-osion huomioimatta.

if-if-if-else versus if-elif-elif-else

On hyvä ymmärtää `if-if-if-else-` ja `if-elif-elif-else`-rakenteiden erot. Koska `if`-osioita voidaan sijoittaa monta peräkkäin, ei mikään varsinaisesti estä meitä käyttämästä useita peräkkäisiä `if`-osioita korvaamaan `elif`-osiot. Tässä yhteydessä pitää kuitenkin ymmärtää, kuinka `if`-rakenne toimii. Yleisesti haluamme, että `if`-rakenteen paikalla ohjelma tekee oikeanlaiset päätelmät annetuista ehdoista ja jatkaa eteenpäin. Ajatellaan vaikka kahta esimerkkiä

```
luku = 50

if luku < 10:
    tulos = "pienempi kuin 10."
if luku < 100:
    tulos = "pienempi kuin 100."
if luku < 1000:
    tulos = "pienempi kuin 1000."
else:
    tulos = "suurempi tai yhtä suuri kuin 1000."
print(tulos)
```

Sekä

```
if luku < 10:
    tulos = "pienempi kuin 10."
elif luku < 100:
    tulos = "pienempi kuin 100."
elif luku < 1000:
    tulos = "pienempi kuin 1000."
else:
    tulos = "suurempi tai yhtä suuri kuin 1000."
print(tulos)
```

Mitä nämä koodit tulostavat? Vaikka koodit ovat keskenään näennäisesti samat testit toteuttavia, tulostaa ylempi vastauksen “pienempi kuin 1000” kun taas alempi tulostaa “pienempi kuin 100”. Miksi näin tapahtuu?

Kyse on nimenomaan suoritusjärjestyksestä. Ylemmässä esimerkissä koodi suorittaa jokaisen `if`-lauseen huolimatta siitä, onko aiempi `if`-lause ollut tosi. Tämä on `if-elif-else`-rakenteen ja `if-if-else`-rakenteiden ero: `elif`-väitteet tutkitaan ainoastaan, mikäli aiemmat `if`- tai `elif`-väitteet ovat saaneet arvon `False`. Lisäksi rakenteesta poistutaan heti, kun ensimmäinen `elif`-lause saa arvon `True`. Sen sijaan jokainen `if`-väittämä testataan erikseen siitä huolimatta, oliko aiempi samaan rakenteeseen kuuluva `if`-lause ollut totta.

Huomaa, että ohjelma toimii oikein: luku 50 on pienempi kuin 100, ja varmasti myös pienempi kuin 1000. Ongelma on kuitenkin siinä, että ohjelma ei ymmärrä lopettaa testausta ”pienempi kuin 100”-tuloksen jälkeen, koska jokainen `if`-lause testataan aina. Tämä on oikeasti ongelma, koska tulkki ei näe koodissa mitään väärää: Mikäli koodia käytettäisi vaikkapa 10-potenssin tunnistamiseen, olisi vastaus hyödytön vaikkakin teknisesti täysin paikkansapitävä.

Ehtolausekkeet ja loogiset operaattorit

`If`-lause sisältää vertailuosion, johon liittyen tarvitset ehtolauseita pystyäksesi päättämään, mikä osio suoritetaan. Tämä toteutetaan operaattoreilla, jotka toimivat samantapaisesti kuin aiemmin katselemamme laskuoperaattorit.

Taulukko 4.1. Loogiset- eli vertailuoperaattorit

Operaattori	Nimi	Selite	Esimerkki
<	Pienempi kuin	Palauttaa tiedon siitä onko x vähemmän kuin y. Vertailu palauttaa arvon `False` tai `True`.	`5 < 3` palauttaa arvon `False` ja `3 < 5` palauttaa arvon `True`.
>	Suurempi kuin	Palauttaa tiedon onko x enemmän kuin y.	`5 > 3` palauttaa arvon `True`. Jos molemmat operandit ovat numeroita, ne muutetaan ne automaattisesti vertailukelpoisiksi eli samantyyppisiksi. Merkkijonoista verrataan ensimmäisiä (vasemmanpuolimmaisia) kirjaimia merkistön mukaisessa järjestyksessä. Tähän palataan luvussa 12.
<=	Vähemmän, tai yhtä suuri	Palauttaa tiedon onko x pienempi tai yhtä suuri kuin y.	Jos `x = 3` ja `y = 6` niin `x <= y` palauttaa arvon `True`.
>=	Suurempi, tai yhtä suuri	Palauttaa tiedon onko x suurempi tai yhtä suuri kuin y.	Jos `x = 4` ja `y = 3` niin `x >= y` palauttaa arvon `True`.
==	Yhtä suuri kuin	Testaa ovatko operandit yhtä suuria.	Jos `x = 2` ja `y = 2` niin `x == y` palauttaa arvon `True`. Jos `x = 'str'` ja `y = 'stR'` niin `x == y` palauttaa arvon `False`. Jos `x = 'str'` ja `y = 'str'` niin `x == y` palauttaa arvon `True`.
!=	Erisuuri kuin	Testaa ovatko operandit erisuuria.	Jos `x = 2` ja `y = 3` niin `x != y` palauttaa arvon `True`.

Taulukko 4.2 Boolean-operaattorit

Operaattori	Nimi	Selite	Esimerkki
not	Boolean NOT	Jos x on `True`, palautuu arvo `False`. Jos x on `False`, palautuu arvo `True`.	`Jos x = True niin not x` palauttaa arvon `False`.
and	Boolean AND	`x and y` palauttaa arvon `False` jos x on `False`, muulloin se palauttaa y:n totuusarvon	Jos `x = False` ja `y = True` niin `x and y` palauttaa arvon `False` koska x on `False`. Tässä tapauksessa Python ei edes testaa y:n arvoa, koska se tietää varman vastauksen. Tätä sanotaan pikatestaukseksi, ja se vähentää tuloksen laskenta-aikaa.
or	Boolean OR	Jos x on `True`, palautuu arvo `True`, Muussa tapauksessa se palauttaa y:n totuusarvon.	Jos `x = True` ja `y = False` niin `x or y` palauttaa arvon `True`. Yllä olevan kohdan maininta pikatestauksesta pätee myös täällä.

Operaattorien suoritusjärjestys

Jos sinulla on vaikkapa lauseke `2 + 3 * 4`, niin suorittaako Python lisäyksen ennen kertolaskua? Jo peruskoulumatematiikan mukaan tiedämme, että kertolasku tulee suorittaa ensin, mutta kuinka Python tietää siitä? Tämä tarkoittaa sitä, että kertolaskulla on oltava korkeampi sijoitus suoritusjärjestyksessä.

Alla olevassa taulukossa on listattuna Pythonin operaattorit niiden suoritusjärjestyksen mukaisesti korkeimmasta matalimpaan. Tämä tarkoittaa sitä, että lausekkeessa, joka sisältää useita operaattoreita, niiden toteutusjärjestys on listan mukainen.

Lista on oiva apuväline esimerkiksi testauslausekkeita tarkastettaessa, mutta käytännössä on suositeltavampaa käyttää sulkeita laskujärjestyksen varmistamiseen. Esimerkiksi `2 + (3 * 4)` on lukijan kannalta paljon parempi kuin `2 + 3 * 4`. Kuitenkin tulee muistaa, että sulkeiden kanssa kannattaa myös käyttää järkeä ja turhien sulkeiden käyttöä välttää, koska se haittaa laskutoimituksen luettavuutta ja ymmärrettävyyttä. Esimerkiksi laskutoimitus `(2 + ((3 + 4) - 1)+(3))` on hyvä esimerkki siitä, mitä ei pidä tehdä.

Taulukko 4.3. Operaattorien suoritusjärjestys

Operaattori	Selite	
`sisältö, ...`	Merkkijonomuunnos	korkea prioriteetti
{avain:tietue ...}	Sanakirjan tulostaminen	
[sisältö, ...]	Listan tulostaminen	
(sisältö, ...)	Tuplen luominen tai tulostaminen	
f(argumentti ...)	Funktiokutsu	
x[arvo:arvo]	Leikkaus	
x[arvo]	Jäsenyyden haku	
x.attribuutti	Attribuuttiviittaus	
******	**Potenssi**	
+x, -x	Positiivisuus, negatiivisuus	
***, /, %**	**Kertominen, jakaminen ja jakojäännös**	
+, -	**Vähennys ja lisäys**	
<, <=, >, >=, !=, ==	**Vertailut**	
is, is not	Tyyppitesti	
in, not in	Jäsenyystesti	
not x	Boolean NOT	
and	Boolean AND	
or	Boolean OR	matala prioriteetti

Listalla on myös operaatioita joita et vielä tässä vaiheessa tunne. Ne selitetään myöhemmin.

Liitännäisyys

Operaattorit arvioidaan yleisellä tasolla vasemmalta oikealle, mikäli niiden suoritusjärjestys on samaa tasoa. Esimerkiksi 2 + 3 + 4 on käytännössä sama kuin `(2 + 3) + 4`. Kannattaa kuitenkin muistaa, että jotkin operaattorit, kuten sijoitusoperaattori,

toimivat oikealta vasemmalle. Tämä tarkoittaa siis sitä, että lauseke `a = b = c` tulkitaan olevan `a = (b = c)`.

Boolean-arvoista

Python tukee loogisten lausekkeiden kanssa työskennellessä Boolean-arvoja, jotka voivat saada joko arvon tosi (`True`) tai epätosi (`False`).

Loogisissa väittämissä, kuten esimerkiksi ”5 on suurempi kuin 3” tai ”a löytyy sanasta apina”, Python antaa arvon `True`, kun esitetty väittämä pitää paikkansa, ja `False`, kun väittämä ei pidä paikkaansa. Esimerkiksi näin:

```
>>> 5 > 3
True
>>> 3 > 5
False
>>> 'a' in 'apina'
True
>>> True == False
False
>>> True == 1
True
>>> False == 0
True
>>>
```

Boolean–arvojen käyttö on loogisissa lausekkeissa luonnollisempaa kuin pelkkien numeroiden käyttö.

Esimerkki 4.3. Boolean-arvot ja operaattorit

```
# -*- coding: utf-8 -*-
# Tiedosto: boolean.py

sana1 = input("Anna 1. sana: ")
sana2 = input("Anna 2. sana: ")

if sana1 == sana2:
    print("Sanat ovat samat.")

if sana1 < sana2:
    print("Ensimmäinen sana on aakkosissa ensin.")

if sana1 == "pulla" and sana2 == "taikina":
    print("Sanat olivat sopivia.")
```

Tuloste

```
>>>
Anna 1. sana: pulla
Anna 2. sana: taikina
Ensimmäinen sana on aakkosissa ensin.
Sanat olivat sopivia.
>>>
Anna 1. sana: foo
Anna 2. sana: foo
Sanat ovat samat.
>>>
```

Tämä ohjelma esittelee kuinka vertailuoperaattoreita (”==”, ”<”) ja boolean-operaattoria (”and”) voidaan käyttää. Ohjelma kysyy käyttäjältä syötteinä kaksi sanaa ja tarkistaa ovatko sanat samoja, onko ensimmäinen sana pienempi, eli aakkosissa ensin ja vielä, että onko sana1 ”pulla” ja sana2 ”taikina”. Mikäli jokin ehto toteutuu, annetaan käyttäjälle palauteteksti.

Luvun asiat kokoava esimerkki

Esimerkki 4.4. Ehtoilua

```
# -*- coding: utf-8 -*-
# Tiedosto: ehtoilua.py

# Kysytään käyttäjältä kolme syötettä ja muutetaan ne kokonaisluvuiksi
luku1 = int(input("Anna 1. luku: "))
luku2 = int(input("Anna 2. luku: "))
luku3 = int(input("Anna 3. luku: "))

if luku1 == luku2:
    print("Luvut 1 ja 2 ovat samoja.")

if luku1 == luku2 and luku2 == luku3:
    print("Kaikki luvut ovat samoja.")

# Ratkotaan luvuista suurin, tasatilanteessakin tulostetaan jokin
# vaihtoehto.
if luku1 < luku2:
    if luku2 < luku3:
        print("Luku 3 on suurin.")
    else:
        print("Luku 2 on suurin.")
else:
    if luku1 < luku3:
        print("Luku 3 on suurin.")
    else:
        print("Luku 1 on suurin.")
```

Luku 5: Toistorakenteet

Ihmisen elämässä usein laatu korvaa määrän, mutta tietokone osaa tehdä vain rajatun määrän erilaisia toimintoja, joten usein toistojen määrä saattaa olla suurikin. Tässä kappaleessa tapaamme kaksi hieman toisistaan poikkeavaa toistorakennetta: `while` ja `for`.

While-rakenteen käyttäminen

Esimerkki 5.1. Numeronarvauspeli while-rakenteen avulla

```
# -*- coding: utf-8 -*-
# Tiedosto: while.py

oikea_numero = 23
while True: # Jatketaan ikuisesti, breakillä pääsee ulos
    arvaus = int(input("Anna kokonaisluku: "))
    if arvaus == oikea_numero:
        print("Arvasit oikein!")
        print("Peli päättyy tähän")
        break # hypätään ulos while-silmukasta
    elif arvaus < oikea_numero:
        print("Luku on suurempi kuin arvaus") # Toinen osio

    else:
        print("Luku on pienempi kuin arvaus")

print("Tämä tulostuu loppuun, koska se on while-rakennetta seuraava
looginen rivi.")
```

Tuloste

```
>>>
Anna kokonaisluku: 40
Luku on pienempi kuin arvaus
Anna kokonaisluku: 20
Luku on suurempi kuin arvaus
Anna kokonaisluku: 23
Arvasit oikein!
Peli päättyy tähän
Tämä tulostuu loppuun, koska se on while-rakennetta seuraava looginen
rivi.
>>>
```

Kuinka se toimii

`While`-silmukan avulla saimme rakennettua fiksulta näyttävän numeronarvauspelin. Ohjelman aluksi määritellään kiintoarvo `oikea_numero` ja annetaan sille arvoksi 23. Tämän jälkeen aloitamme `while`-silmukan ja määritämme, että sitä suoritetaan niin kauan kuin `True` on totta ja tämähän on aina tosi, joten

Muuttujan rooli: tuoreimman säilyttäjä

Tutustuimme aiemmin kiintoarvoon, eli muuttujaan, jonka arvo ei muutu ohjelman ajon aikana. Edellisessä esimerkissämme muuttuja `arvaus` sai jokaisella kierroksella uuden arvon, se siis säilytti itsessään tuoreimman syötteen ja oli täten rooliltaan *tuoreimman säilyttäjä.*

silmukka pyörii näennäisen loputtomasti.

Silmukan sisällä kysymme käyttäjältä arvauksena kokonaislukua ja vertaamme sitä kiintoarvoon `oikea_numero`. Mikäli käyttäjä arvaa oikein, tulostetaan kehut ja poistutaan koko while-silmukasta `break`-komennolla.

Mikäli käyttäjän arvaus ei osunut oikeaan, verrataan oliko se pienempi kuin oikea numero ja jos oli, niin tulostetaan teksti ”Luku on suurempi kuin arvaus”. Jos ei ollut, tulostetaan vastakkainen teksti ”Luku on pienempi kuin arvaus”. Mikäli käyttäjä ei siis arvannut oikein, tulostetaan vihje ja palataan `while`-silmukan alkuun ja kysytään uutta arvausta.

Lopulta käyttäjän arvattua luvun, ohjelma hyppää ulos `while`-silmukasta ja suorittaa seuraavan rivin, joka on sisennetty samalle tasolle kuin `while`-sana. Tässä tapauksessa tulostetaan siis teksti ”Tämä tulostuu loppuun, koska se on `while`-rakennetta seuraava looginen rivi.”

Huomioita while-rakenteesta

`while`-rakenne on varsin vapaamuotoinen, eikä tulkki esimerkiksi valvo sen edistymistä millään tavoin. Tämä tarkoittaa sitä, että käyttäjän täytyy itse huolehtia siitä, että toistorakenne saavuttaa joskus lopetusehtonsa. Muussa tapauksessa `while`-lause jää ikuiseen toistoon ja ohjelma menee jumiin. `While`-rakennetta voidaan myös käyttää numeroarvojen kanssa työskennellessä:

Esimerkki 5.2. while-rakenteen toiminta

```
# -*- coding: utf-8 -*-
# Tiedosto: while2.py

arvo = 0
while arvo < 4:
   print(arvo)
   arvo = arvo + 1
```

Tuloste

```
>>>
0
1
2
3
>>>
```

Huomaa, että tässä tapauksessa teemme testauksen lähes samalla periaatteella kuin aiemmin. Edellisessä esimerkissä joka kierroksella tarkistettiin, onko `True` totta ja olihan se. Nyt tarkastamme, onko muuttujan `arvo` sisältämä numeroarvo pienempi kuin 4. Kierroksella, jolla tulostamme 3, kasvatamme muuttujan arvoa yhdellä, joten toistoehto ei enää toteudu.

For-rakenne

`For`-rakenne on toinen Pythonin kahdesta tavasta toteuttaa toistorakenteita. `For`-lause eroaa `while`-rakenteesta kahdella merkittävällä tavalla. Ensinnäkin `for`-lauseen kierrosmäärä on pystyttävä esittämään vakioituna arvona. Toisekseen `for`-lausetta voi käyttää monialkioisten rakenteiden, kuten listojen alkioiden läpikäymiseen. Tällä kertaa keskitymme kuitenkin `for`-lauseen käyttämiseen sen perinteisemmässä muodossa eli silmukoiden tekemisessä. Listoista ja niiden läpikäymisestä puhumme myöhemmin.

Esimerkki 5.3. for-rakenteen toiminta

```
# -*- coding: utf-8 -*-
# Tiedosto: for.py

for i in range(1, 5):
      print(i)

print("Silmukka on päättynyt.")
```

Tuloste

```
>>>
1
2
3
4
Silmukka on päättynyt.
>>>
```

Kuinka se toimii

Tämä ohjelma tulostaa joukon numeroita. Joukko luodaan sisäänrakennetun funktion `range` avulla. Käsky `range(1,5)` luo meille lukujonon yhdestä viiteen poislukien ylärajan (1 <= i < 5) [1,2,3,4], jossa siis on neljä jäsentä. `For`-rakenne käy läpi nämä neljä jäsentä – toistorakenne tapahtuu neljä kertaa - jonka jälkeen ohjelma jatkaa normaalia kulkuaan seuraavalta loogiselta riviltä, joka tulostaa kommentin silmukan päättymisestä.

Tässä vaiheessa on hyvä huomata se, että esimerkiksi aiemmin käydyn `while`-rakenteen `True`-kytkin ei tässä tapauksessa toimisi, koska se ei yksiselitteisesti kerro sitä, kuinka monesti `for`-rakenne ajetaan läpi. `range`-funktio sen sijaan tekee tämän luomalla neljän numeron ryhmän, jonka `for`-lause läpikäy kohta kerrallaan, tässä tapauksessa tulostaen aina jäsenen numeron. `for`-lause osaakin käydä tällä tavoin läpi kaikkia Pythonin sarjarakenteisia muotoja, mutta tällä erää riittää, kun tiedät kuinka `for`-lause yleisesti ottaen toimii.

Muuttujan rooli: askeltaja

Esimerkin 5.2 muuttuja `arvo` ja 5.3 `for`-silmukassa oleva muuttuja `i` saavat silmukan jokaisella kierroksella ennalta määrätyn uuden arvon, ne siis askeltavat läpi niille määrätyn joukon. Tästä syystä muuttujien rooli onkin *askeltaja.*

Break-käsky

Ensimmäisestä `while`-esimerkissämme 5.1 törmäsimmekin jo `break`-käskyyn pikaisesti. `break`–käskyä käytetään ohjaamaan toistorakenteen toimintaa. Sen avulla voimme keskeyttää toistorakenteen suorittamisen, jos päädymme tilaan, jonka jälkeen toistojen tekeminen olisi tarpeetonta. Tämä tarkoittaa esimerkiksi tilannetta, jossa etsimme numeroarvoa suuresta joukosta. Kun löydämme sopivan numeron, voimme lopettaa toiston välittömästi läpikäymättä joukon loppuosaa. `break`-käskyn jälkeen ohjelma jatkaa toistorakenteen jälkeiseltä seuraavalta loogiselta riviltä, vaikka toistoehto ei olisikaan saavuttanut arvoa `False`.

Tämä tarkoittaa myös sitä, että `break` –käskyn tapahtuessa myös toistorakenteen `else`-osio ohitetaan.

Esimerkki 5.4. Toistorakenne break-käskyllä höystettynä

```
# -*- coding: utf-8 -*-
# Tiedosto: break.py

for i in range(1,11):
    if i == 7:
        print("Onnennumero", i, "löytyi!")
        break
    print("Tutkitaan numeroa", i)
else:
    print("Tätä te ette koskaan tule näkemään.")

print("Loppu!")
```

Tuloste

```
>>>
Tutkitaan numeroa 1
Tutkitaan numeroa 2
Tutkitaan numeroa 3
Tutkitaan numeroa 4
Tutkitaan numeroa 5
Tutkitaan numeroa 6
Onnennumero 7 löytyi!
Loppu!
>>>
```

Kuinka se toimii

Ohjelman toiminta on varsin yksinkertainen. `for`-rakenne asetetaan käymään lukuja lävitse yhdestä yhteentoista. Silmukan sisällä tarkistetaan onko käsiteltävä luku 7, ja jos on, tulostetaan onnennumeron löytyminen ja hypätään ulos silmukasta. Viimeiset kolme numeroa jäävät tutkimatta, samoin `else`-osion tulostusta ei nähdä koskaan. Pythonin syntaksi olettaa, että toistorakenteen katkaiseminen `break`-käskyllä tarkoittaa, että toistorakenne on täyttänyt sille annetun tehtävän, eikä täten ”muussa tapauksessa” –osiota tule ajaa.

Muista myös, että `break`-lause toimii myös `while`-rakenteen kanssa.

Continue-käsky

`continue` on toiminnaltaan pitkälti `break`-käskyn kaltainen, mutta toimii hieman eri tarkoituksessa. Kun `break`-käsky lopettaa koko toistorakenteen suorittamisen, `continue` ainoastaan määrää, että toistorakenteen loppuosa voidaan jättää käymättä läpi ja että siirrytään välittömästi seuraavalle kierrokselle.

Esimerkki 5.5. Toistorakenne continue-käskyllä höystettynä

```
# -*- coding: utf-8 -*-
# Tiedoston: continue.py

while True:
      merkkijono = input("Syötä tekstiä: ")
      if merkkijono == "lopeta":
            break
      if len(merkkijono) < 6:
            continue
      print("Syöte on yli 5 merkkiä")
```

Tuloste

```
>>>
Syötä tekstiä: testi
Syötä tekstiä: uudelleenyritys
Syöte on yli 5 merkkiä
Syötä tekstiä: no niin!
Syöte on yli 5 merkkiä
Syötä tekstiä: lopeta
>>>
```

Kuinka se toimii

Tämä ohjelma ottaa vastaan käyttäjältä merkkirivejä. Ohjelma tarkastaa, onko merkkirivi yli 5 merkkiä pitkä ja mikäli tämä ehto täyttyy, suorittaa se jatkotoimenpiteitä. Jos merkkijono jää alamittaiseksi, ohjelma hyppää `continue`-käskyn avulla uudelle kierrokselle. Kun käyttäjä syöttää lopetuskäskyn ”lopeta”, ohjelma katkaisee toistorakenteen `break`-käskyllä.

`Continue` toimii myös `for`-rakenteen kanssa.

Range-funktiosta

Aiemmin `for`-rakenteen yhteydessä mainittiin funktio `range`. Kyseinen funktio on sisäänrakennettu Pythoniin ja juuri se mahdollistaa `for`-lauseen käyttämisen normaalin toistorakenteen tavoin myös silloin, kun tieto ei ole tallennettuna sarjamuotoiseen muuttujaan. `range`-funktion käyttö on varsin yksinkertaista:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Annetun alueen viimeinen arvo ei koskaan kuulu joukkoon. Kuten ylläolevasta esimerkistä huomasit, on generoitavalla lukujonolla 10 arvoa, kuten pyydettiinkin, mutta viimeinen

arvo on 9. Lisäksi `range`-funktiota käytettäessä voidaan käyttää leikkausmaisia arvomäärittelyjä, kun määritellään aloituslukua, lopetuspaikkaa ja siirtymäväliä. `range`n oletusarvot ovatkin aloituspaikalle 0 sekä askelvälille 1. Lopetuspaikka täytyy aina määritellä käsin, koska sille ei oletusarvoa ole annettu.

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Jos haluat käsitellä merkkijonoa, voi `range`- ja `len`-funktiot myös yhdistää:

```
>>> sana = "ankka"
>>> for i in range(len(sana)):
      print(sana[i], end=" ")

a n k k a
>>>
```

Else-osio toistorakenteessa

`Else`-osiota käytetään yleisesti toistorakenteessa merkitsemään sitä, mitä tehdään, jos toistorakenteen läpikäyminen ei ole tuottanut toivottua tulosta. Tämä tarkoittaa siis sitä, että `else`-rakenne käydään läpi aina, kun toistorakenne loppuu toistoehdon täyttymiseen. Jos toistorakenne lopetetaan `break`-käskyllä, ei `else`-osiota suoriteta. Tästä tarkentavana esimerkkinä voidaan ottaa ohjelma, joka laskee alkulukuja:

Esimerkki 5.6. Else-osio toistorakenteessa, alkuluvut

```
# -*- coding: utf-8 -*-
# Tiedosto: alkuluku.py

for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, "on yhtä kuin", x, "*", n/x)
            break
    else:
        # Kierros päättyi siten, että ohjelma ei löytänyt sopivaa paria
        print(n, "on alkuluku")
```

Tuloste

```
>>>
2 on alkuluku
3 on alkuluku
4 on yhtä kuin 2 * 2
5 on alkuluku
6 on yhtä kuin 2 * 3
7 on alkuluku
8 on yhtä kuin 2 * 4
9 on yhtä kuin 3 * 3
>>>
```

Luvun asiat kokoava esimerkki

Esimerkki 5.7. Silmukointia

```
# -*- coding: utf-8 -*-
# Tiedosto: silmukointia.py

while True:
    kerroin = int(input("Anna toistojen lukumäärä (0=lopetus): "))
    if kerroin == 0:
        break
    sana = input("Anna toistettava sana: ")
    for i in range(0,kerroin):
        print(sana)

print("Ohjelman suoritus lopetettu.")
```

Luku 6: Pääohjelma, aliohjelmat ja funktiot

Ajatellaanpa hetki suomen kielen kielioppia. Meillä on käytössä lauseita ja virkkeitä, joista jälkimmäiset sisältävät erilaisia lausetyyppejä: päälauseita ja sivulauseita. Kaikki muistavat peruskoulusta, että sivulauseet ovat alisteisia päälauseelle – ne eivät siis toimi ilman päälausetta. Nyt mieleesi herännee kysymys, että mitä tekemistä tällä on Pythonin kanssa? Python ei tässä mielessä eroa mitenkään suomen kielestä. Python-ohjelmassakin (virke) voi olla pääohjelma (päälause) ja aliohjelmia (sivulauseita).

Tähän mennessä olemme kirjoittaneet ohjelmia, joissa on vain päätason koodia, eli pelkkä pääohjelma. Tässä luvussa perehdymme pääohjelman hyödyksi rakennettaviin aliohjelmiin, joita kutsutaan funktioiksi.

Funktiot ovat ”koodinpätkiä”, joita voidaan käyttää uudestaan moneen kertaan. Niillä on mahdollista nimetä koodiin osio, jolle annetaan syötteet, joiden avulla tämä osio – funktio – suorittaa sille määritellyt toimenpiteet ja palauttaa laskemansa vastauksen. Funktion käyttöön ottamista sanotaan funktion kutsumiseksi. Arvoa, jonka funktio antaa vastaukseksi sanotaan paluuarvoksi. Esimerkiksi `len` on itse asiassa funktio, jota kutsutaan laskettavaksi haluttavalla merkkijonolla tai muuttujalla, ja palautunut numeroarvo on funktion `len` paluuarvo. Python ei ole funktiorakenteen kanssa niin tarkka kuin esimerkiksi C-kieli, mutta on hyvän ohjelmointitavan mukaista, että kaikki kirjoitetut käskyt ja loogiset lausekkeet on sijoitettu funktioiden sisään.

Funktion tekeminen alkaa määrittelemällä sen alkamispaikka `def` –avainsanalla, joka tulee englanninkielen sanasta define. Tämän jälkeen seuraa funktion nimi, jolla funktio tästä eteenpäin tunnetaan, sulut, joihin määritellään mahdolliset syötteet sekä kaksoispiste, joka määrittelee funktio-osion alkaneeksi. Parhaiten asian ymmärtää alla olevasta esimerkistä; alkuun funktiot voivat vaikuttaa hämmentäviltä, mutta itse asiassa ne eivät ole kovinkaan monimutkainen asia.

Funktio vai aliohjelma?

Tässä ollaan perimmäisten kysymysten äärellä. Kumpi oli ensin muna vai kana? Joissain ohjelmointikielissä käytetään erillisiä avainsanoja luomaan aliohjelmia: proseduureja ja funktiota. Näiden erona pidetään sitä, ettei proseduuri palauta mitään, kun taas funktio palauttaa. Pythonin tapauksessa aliohjelma on funktio, jos se palauttaa `return`illa jotain. Näiden kahden termin käyttö on kuitenkin harmaantunut vuosien saatossa ja termiä funktio käytetään niin proseduureista kuin funktioistakin. Loppujen lopuksi Pythonkin palauttaa aina jotain. Jos ohjelmoija ei laita funktiota palauttamaan mitään, lisää tulkki automaattisesti `return None` osan funktioon.

Funktion määritteleminen

Esimerkki 6.1. Funktion määritteleminen ja kutsuminen

```
# -*- coding: utf-8 -*-
# Tiedosto: funktio1.py

# Aliohjelma alkaa
def sano_terve(): # Funktion määrittely alkaa tästä
    print("Terve vaan!")
    print("Tämä tulostus tulee funktion sisältä!")
# Aliohjelma loppuu

# Pääohjelma alkaa
sano_terve() # Funktiota kutsutaan sen omalla nimellä
print("Funktio suoritettu.")
# Pääohjelma loppuu
```

Tulostus

```
>>>
Terve vaan!
Tämä tulostus tulee funktion sisältä!
Funktio suoritettu.
>>>
```

Kuinka se toimii

Edellä loimme funktion nimeltä `sanoTerve` käyttäen ohjeita, jotka aiemmin kävimme lävitse. Tämän funktion kutsussa ei ole muuttujia, joten sitä voidaan kutsua suoraan omalla nimellään. Ohjelman aluksi määritämme funktion ja aloitamme ohjelman suorituksen kutsumalla funktiota sen nimellä. Tämän jälkeen ohjelman suoritus siirtyy funktion sisään. Tällä kertaa siellä ei ole muuta kuin kaksi tulostuslausetta. **Kun funktio saadaan suoritettua, palataan siihen kohtaan ohjelmaa, josta funktiokutsu lähti**, eli tässä tapauksessa suoritus jatkuu pääohjelman tulostuksella ”Funktio suoritettu”.

Pää- ja aliohjelma

Tällä kurssilla pääohjelmalla tarkoitetaan koodia, jota ei ole sijoitettu mihinkään aliohjelmaan.

Pythonissa on myös mahdollista määritellä erillinen pääohjelma käynnistymään koodia ajettaessa:

```
if __name__ == "__main__":
    main()
```

Funktion tärkein etu on sen uudelleenkäytettävyysarvo. Nyt kun funktio on luotu, on se saatavilla uudelleenkäyttöä varten vain kutsumalla sitä uudelleen (olettaen että se on aiemmin kertaalleen jo tulkattu). Lisätään pääohjelmaan rivit

```
print("Toistetaan")
sano_terve()
print("Toistetaan")
sano_terve()
```

Tämän jälkeen tulostus näyttää tällaiselta:

```
>>>
Terve vaan!
Tämä tulostus tulee funktion sisältä!
Funktio suoritettu.
Toistetaan
Terve vaan!
Tämä tulostus tulee funktion sisältä!
Uudelleen
Terve vaan!
Tämä tulostus tulee funktion sisältä!
>>>
```

Funktio, joka on esitelty lähdekoodin alussa, voidaan kutsua aina uudelleen ja uudelleen pääkoodin sisällä niin useasti kuin vain haluamme. Tästä johtuen omatekemät funktiot esitelläänkin ennen pääohjelmaa, koska Python-tulkki lukee asiat samalla tavoin kuin me ihmiset. Tulkin pitää etukäteen lukea tieto funktion olemassaolosta ennen kuin sitä voidaan käyttää; ethän sinäkään voi tietää, mitä tämän oppaan liitteissä lukee, ennen kuin olet ne kertaalleen lukenut.

Funktioita voi myös kutsua Pythonin komenrivitotulkissa. Kirjoittamalla funktion nimen IDLEn interaktiiviseen ikkunaan voimme välittömästi ajaa funktion uudelleen ilman, että joudumme palaamaan takaisin itse lähdekooditiedostoon. Erityisen hyödyllistä tämä on silloin, kun pääohjelma on oma funktionsa, jolloin ohjelma voidaan käynnistää kutsumalla kyseistä funktiota.

Funktiokutsu ja parametrien välitys

Funktiokutsu voi sisältää parametreja, jotka käytännössä ovat arvoja, joiden avulla funktio osaa työskennellä. Nämä parametrit käyttäytyvät funktion sisällä aivan kuin ne olisivat normaaleja muuttujia sillä erolla, että ne määritellään funktiokutsussa eikä funktion sisällä.

Parametrin määrittely tapahtuu funktion määrittelyssä olevien sulkujen sisään. Jos halutaan antaa useampia parametreja, tulee ne erotella toisistaan pilkuilla. Funktiokutsussa vastaavasti parametreille annettavat arvot syötetään samassa järjestyksessä. Kannattaa myös huomioida että funktiokutsun arvoja sanotaan sekä parametreiksi että argumenteiksi. Selvyyden vuoksi puhumme jatkossa pelkästään parametreista.

Esimerkki 6.2. Funktio, jolle annetaan parametreja

```
# -*- coding: utf-8 -*-
# Tiedosto: funktio2.py

def sano_terve(nimi, osasto, vuosikurssi):
    print("Terve vaan " + nimi + "!")
    print("Sanoit olevasi osastolla " + osasto + ".")
    print("Ja että meneillään on " + vuosikurssi + ". vuosi.")

sano_terve("Brian", "Tite", "4" ) # Annetaan kutsussa parametreja
```

Tuloste

```
>>>
Terve vaan Brian!
Sanoit olevasi osastolla Tite.
Ja että meneillään on 4. vuosi.
>>>
```

Kuinka se toimii

Kuten huomaat, ei uusi koodi poikkea paljoa ensimmäisestä esimerkistä. Tällä kertaa funktiokutsu sisältää kolme muuttujaa: nimi, osasto ja vuosikurssi. Kuten huomaat, voit käyttää funktion saamia parametreja sen sisällä aivan kuten normaaleja muuttujia. Myös suoraan muuttujilla kutsuminen onnistuu helposti:

```
>>> a = "Late"
>>> b = "Kote"
>>> c = "2"
>>> sano_terve(a, b, c)
Terve vaan Late!
Sanoit olevasi osastolla Kote.
Ja että meneillään on 2. vuosi.
>>>
```

Tässä tapauksessa tulkki yksinkertaisesti siirtää muuttujien a, b ja c arvot funktiokutsun parametreiksi. a:n arvo siirtyy muuttujaan nimi, b:n arvo muuttujaan osasto ja c:n arvo muuttujaan vuosikurssi.

Nimiavaruudet

Nimiavaruus on ensimmäinen asia, johon luultavasti tulet törmäämään ainakin kerran aloittaessasi ohjelmoimaan funktioita apuna käyttäen. Jos luot funktion sisälle muuttujan `luku`, niin tämä muuttuja on käytettävissä ainoastaan sen nimenomaisen funktion sisällä, jossa se luotiin. Itse asiassa, voit luoda jokaiseen funktioon muuttujan `luku`, koska jokainen funktio toimii omassa nimiavaruudessaan. Tämä tarkoittaa siis sitä, että funktioiden välillä muuttujilla ei ole minkäänlaista yhteyttä. Tämän vuoksi funktionsisäisiä muuttujia sanotaan paikallisiksi tai lokaaleiksi muuttujiksi. Ne näkyvät ainoastaan oman funktionsa sisällä.

Esimerkki 6.3. Paikallisten muuttujien toimivuus

```
# -*- coding: utf-8 -*-
# Tiedosto: funktio_lokaali.py

def funktio(x):
    print("x on funktioon sisään tullessaan", x)
    x = 2
    print("x muutettu funktion sisällä arvoon", x)

x = 50
print("x on ennen funktiota", x)
funktio(x)
print("x on funktion kutsumisen jälkeen edelleen", x)
```

Tuloste

```
>>>
x on ennen funktiota 50
x on funktioon sisään tullessaan 50
x muutettu funktion sisällä arvoon 2
x on funktion kutsumisen jälkeen edelleen 50
>>>
```

Kuinka se toimii

Funktion sisällä me näemme, kuinka funktio saa kutsussaan parametrina muuttujan x arvon 50. Tämän jälkeen me muutamme funktion sisällä olevan muuttujan x arvoksi 2. Tämä ei kuitenkaan vaikuta päätason muuttujan x arvoon, koska funktion muuttuja x ja päätason muuttuja x ovat ainoastaan samannimisiä, mutta eri avaruudessa olevia muuttujia. Funktiokutsussa parametrina välitetään ainoastaan muuttujan arvo. Ajattele asiaa vaikka siten, että funktiokutsussa sinä lähetät muuttujan sisällöstä kopion, etkä alkuperäistä sisältöä. Ja koska funktio tekee muutoksensa kopioon, eivät ne näy kun myöhemmin tarkastelet alkuperäistä muuttujaa x.

Yhteiset muuttujat

Joskus jostain syystä kuitenkin haluat päästä muokkaamaan ylemmän tason muuttujaa. Tai sitten saatamme haluta, että muuttujan arvo on todellakin kaikkien funktioiden muokattavissa. Silloin ratkaisu on se, että käytät yhteisiä eli globaaleja muuttujia. Pythonissa on olemassa varattu sana `global`, joka tarkoittaa sitä, että käyttäjä haluaa nimenomaisesti käyttää ylemmän tason – normaalisti päätason - muuttujia paikallisten muuttujien sijaan.

Tätä etenemistapaa ei kuitenkaan voida suositella käytettäväksi vakavamielisessä ohjelmoinnissa. Yleisesti ottaen yhteisten muuttujien käyttäminen ja nimiavaruuksien sotkeminen keskenään tekee koodista monimutkaisempaa sekä virhealttiimpaa. Lisätään vielä se fakta, että kasvava lähdekoodin määrä tarkoittaa normaalisti myös suurempaa kompleksisuutta, joten globaaleista muuttujista on lähinnä vain harmia. Yleisesti ongelmat johtuvat siitä, että muuttujien roolit eivät pysy samoina funktiosta toiseen, tai että rooli kyllä säilytetään, mutta ohjelma toimii siten, että pääohjelmatasolla jotain menee pieleen. Tämä tarkoittaa normaalisti tilanteita, joissa funktiot vahingossa ylikirjoittavat tietoa, mikä taas aiheutuu siitä, että käytettävä yhteinen muuttuja on toiselle funktiolle suorituskriittinen muuttuja ja toiselle taas pelkkä varastomuuttuja, jonka sisällöstä ei niin ole väliä.

Esimerkki 6.4. Yhteisen muuttujan käyttäminen

```
# -*- coding: utf-8 -*-
# Tiedosto: funktio_globaali.py

def funktio():
    global x

    print("x on", x)
    x = 2
    print("Yhteinen x muutettiin funktiossa arvoksi", x)

x = 50
print("Ennen funktiota x oli", x)
funktio()
print("Funktion jälkeen x on", x)
```

Tuloste

```
>>>
Ennen funktiota x oli 50
x on 50
Yhteinen x muutettiin funktiossa arvoksi 2
Funktion jälkeen x on 2
>>>
```

Kuinka se toimii

`global` –sanaa käytetään funktion sisällä muuttujan x kanssa kertomaan tulkille, että funktio käyttää ylemmän tason muuttujaa. Tämän ansiosta päätason muuttujaa x ei tarvitse välittää parametrina, ja tämän ansiosta funktion sisällä tehdyt muutokset jäävät voimaan myös funktion suorituksen jälkeen. Luonnollisesti parametrin välittäminen ja paluuarvon käyttäminen olisi kuitenkin turvallisempaa ja yhtä yksinkertaista. Tästä syystä perehdymmekin funktion paluuarvoon seuraavaksi.

Paluuarvo

Paluuarvo on funktion osa, joka suorittaa funktiosta poistumisen ja mahdollisen muuttuja-arvon palauttamisen. Pythonissa paluuarvoa varten on varattu sana `return`, johon päätyminen keskeyttää funktion suorittamisen samalla tavoin kuin `break` keskeyttää toistorakenteen. `return`-käskyn kanssa voidaan myös esitellä muuttuja, joka palautetaan funktiokutsun tehneelle osiolle. Tämäkin asia on helpoin ymmärtää esimerkin avulla.

Esimerkki 6.5. Paluu-arvo käytännössä

```
# -*- coding: utf-8 -*-
# Tiedosto: func_return.py

def maksimi(x, y):
    if x > y:
        isompi = x
    else:
        isompi = y
    return isompi

luku1 = 100
luku2 = 50
suurempi = maksimi(luku1, luku2)
print("Suurempi arvo on", suurempi)
```

Tuloste

```
>>>
Suurempi arvo on 100
>>>
```

Kuinka se toimii

`maksimi`-funktio ottaa vastaan funktiokutsussaan kaksi parametria ja vertailee niitä keskenään yksinkertaisella `if-else`-rakenteella. Tämän jälkeen funktio palauttaa funktiokutsun tehneelle lauseelle suuremman arvon `return` –käskyllä.

Huomaa, että tässä tapauksessa funktion ja päätason ei tarvitse sotkea keskenään nimiavaruuksia johtuen siitä, että päätason muuttuja suurempi saa sijoituksena funktion `maksimi` paluuarvon. Tämä mahdollistaa tehokkaan koodin tekemisen ilman, että joutuisimme käyttämään yhteisiä muuttujia.

Jos `return`-käskylle ei anneta mitään palautusarvoa, on Pythonin vakio silloin return `None`, eli `return`-käsky lopettaa funktion ajamisen, mutta ei palauta mitään.

Esimerkki 6.6. Paluu-arvo käytännössä, osa 2

```
# -*- coding: utf-8 -*-
# Tiedosto: kyltti.py

def tee_kyltti(teksti):
    plakaatti = "*" * (len(teksti) + 4) + "\n"
    plakaatti = plakaatti + "* " + teksti + " *\n"
    plakaatti = plakaatti + "*" * (len(teksti) + 4) + "\n"
    return plakaatti

syote = input("Anna syöte: ")
taulu = tee_kyltti(syote)
print(taulu)
```

Tuloste

```
>>>
Anna syöte: Matti
*********
* Matti *
*********

>>>
Anna syöte: Python-koodailu on kivaa!
****************************
* Python-koodailu on kivaa! *
****************************

>>>
```

Kuinka se toimii

Esimerkkikoodi on lyhyehkö, mutta se saa paljon aikaan. Ohjelman suoritus alkaa siitä, että käyttäjältä kysytään syöte. Tämän jälkeen syöte annetaan `tee_kyltti`-funktion parametriksi. Funktio alkaa rakentamaan tulostaan `plakaatti` nimiseen muuttujaan. Ensimmäiseksi siihen laitetaan ”*”-merkkiä käyttäjän syötteen pituuden verran + 4 kappaletta ja rivinvaihto. Seuraavalla rivillä plakaattiin lisätään tähti ja välilyönti sekä käyttäjän antama tekstin ja vielä loppuun välilyönti, tähti ja rivinvaihto. Funktion kolmas rivi lisää plakaatti-muuttujaan samanlaisen tähtirivin kuin ensimmäinen rivi. Neljäs rivi palauttaa valmiin plakaatin.

Funktion palauttama sisältö otetaan talteen `taulu`-muuttujaan, joka tulostetaan ruudulle. Ohjelma siis rakentaa syötteen ympärille tähtikehykset.

Voimme luonnollisesti käyttää funktiota myös valmiilla merkkijonoilla ja rakentaa nopeasti useita tauluja seuraavasti:

```
print(tee_kyltti("Matti"))
print(tee_kyltti("<3"))
print(tee_kyltti("Mervi"))
```

Tulostaa seuraavaa:

```
>>>
*********
* Matti *
*********

******
* <3 *
******

*********
* Mervi *
*********

>>>
```

Funktioiden dokumentaatiorivit ja help-funktio

Luodaan ohjelma, joka laskee Fibonaccin lukusarjan lukuja:

```
def fib(maara):
     """Tulostaa Fibonaccin lukusarjan maara ensimmäistä \n \
      jäsentä. Suositus maara < 400."""
     kulkija = 0
     tuorein = 1
     for i in range(0,maara):
         print(tuorein, end = " ")
         kulkija = tuorein
         tuorein = kulkija + tuorein

>>> fib(11)
1 1 2 3 5 8 13 21 34 55 89
>>>
```

Huomaat varmaan, että funktion ensimmäiselle riville on ilmestynyt kolmen sitaattimerkin notaatiolla tehty merkkirivi. Tämä ei aiheuta tulkissa virhettä siksi, koska se on dokumentaatiorivi, englanniksi *docstring*. Jos merkkirivi aloittaa funktion, ymmärtää Pythonin tulkki sen olevan dokumentaatiorivi, eli eräänlainen ohjerivi jossa kerrotaan funktion toiminnasta, annetaan ohjeita sen käytöstä tai mahdollisesti käydään läpi joitain perusasioita paluuarvoista tai parametreista.

Dokumentaatiorivi toimii siten, että kun IDLEssä kirjoitat funktiokutsua kyseisestä rivistä, näet automaattisesti aukenevassa apuruudussa funktiokutsun mallin niin kuin kirjoitit sen itse määrittelyyn, sekä sen alapuolella kirjoittamasi dokumentaatiorivin. Rivin käyttö ei ole pakollista, mutta se helpottaa tuntemattomien funktioiden käyttöä ja on hyvän ohjelmointitavan mukaista.

Dokumentaatiorivi näkyy myös IDLEn interaktiivsessa ikkunassa, jos käytät Pythonin sisäänrakennettua help-funktiota. Esimerkiksi yllä olevan funktion ajaminen help-funktiosta läpi näyttäisi tältä:

```
>>> help(fib)
Help on function fib in module __main__:

fib(maara)
    Tulostaa Fibonaccin lukusarjan maara ensimmäistä
    jäsentä. Suositus maara < 400.

>>>
```

Tulkki siis esittää kootusti annetun funktion – tai kirjastomoduulin – dokumentaatiorivit, joissa on ohjeena, mitä milläkin funktiolla voi tehdä. Jos taas kirjoitat pelkän ”help()” tulkkiin, käynnistyy Pythonin sisäänrakennettu apuohjelma. Sieltä voit tulkin kautta selata ohjetietoja eri käskyistä ja moduuleista. Helppi lopetetaan jättämällä rivi tyhjäksi, kirjoittamalla ”quit” ja painamalla enter.

Pythonin valmiiksi tarjoamia funktioita

Tähän mennessä olemmekin tutustuneet jo jokuseen Pythonin valmiina tarjoamaan funktioon, mutta niitä on vielä huomattavasti enemmän kuin `print`, `input`, `len`, `int`, `float`, `str`, `round` ja `range`. Seuraavassa esitellään muutama varsin käytännöllinen funktio.

abs(*x*)

`abs` palauttaa parametrinaan saaneen luvun itseisarvon (absolute value).

```
>>> abs(-42)
42
```

max(*x*)

`max` palauttaa parametrinaan saadun joukon suurimman alkion. `max` voi ottaa parametrinaan myös listan, johon tutustutaan luvussa 8.

```
>>> max(4, 3, 2, 65, 234, 2, 23423, 43)
23423
```

min(*x*)

`min` toimii kuten `max`, mutta palauttaa pienimmän alkion.

```
>>> min(4, 3, 2, 65, 234, 2, 23423, 43)
2
```

round(*x*[, *n*])

`round`in tehtävä on pyöristää lukuja. Parametri `x` on luku jota pyöristetään ja vapaaehtoinen parametri `n` kertoo, kuinka monen desimaalin tarkkuudella pyöristys tehdään. Jos `n` jätetään pois, tehdään pyöristys kokonaisluvuksi eli nollan desimaalin tarkkuuteen.

```
>>> round(3.14159265, 3)
3.142
```

sum(*iterable*[, *start*])

`sum` palauttaa parametrinaan saamansa listan alkioiden summan (toimii vain numeroilla). Valinnainen `start` parametri kertoo, mistä alkiosta laskeminen lähtee liikenteeseen. Oletuksena on alku eli alkio numero 0.

```
>>>lista= [1,34,65,45,7,456, 4]
>>>sum(lista)
612
```

type(*object***)**

`type` kertoo, mitä tyyppiä parametrina annettu muuttuja on.

```
>>> type("tekstiä")
<class 'str'>
```

Luvun asiat kokoava esimerkki

Esimerkki 6.7. Funktio

```
# -*- coding: utf-8 -*-
# Tiedosto: funkkis.py

def funktio(teksti, kerroin):
    for i in range(0, kerroin):
        print(teksti)

merkkijono = input("Anna tekstiä: ")
numero = int(input("Anna numero: "))

funktio(merkkijono, numero)
funktio("LUT", 3)
```

Luku 7: Tiedostojen käsittely ja jäsenfunktiot

Usein kohtaamme tilanteen, jossa haluaisimme tallentaa muuttujien arvot koneelle tai laatia asetustiedoston. Tällöin voisimme lukea muuttujien arvot suoraan koneelta ilman, että joudumme joka kerta aloittamaan ohjelman kirjoittamalla muuttujien tiedot koneelle. Tällaisia tilanteita varten Python luonnollisesti tukee tiedostoihin kirjoittamista sekä niistä lukemista, joista puhumme tässä luvussa.

Python käsittelee tiedostoja hyvin maanläheisellä ja yksinkertaisella tavalla. Avattu tiedosto on käytännössä Pythonin tulkille ainoastaan pitkä merkkijono, jota käyttäjä voi muutella mielensä mukaisesti. Tähän liittyy kuitenkin tärkeä varoitus: Python-tulkki ei erota järjestelmäkriittisiä tiedostoja tavallisista tekstitiedostoista! Availe ja muuttele ainoastaan niitä tiedostoja, joista voit olla varma, että niitä voi ja saa muutella.

Tiedostojen kanssa työskentely

Käsiteltäessä tiedostoja olisi meidän hyvä ensin tietää, mistä niitä löydämme. Jos teemme töitä IDLEn editorilla tai käytämme interaktiivista ikkunaa, on oletuskansio se, mihin lähdekoodi on tallennettu. Nyt, kun me tiedämme mihin tekemämme muutokset tallentuvat, voimme avata tiedoston, kirjoittaa sinne jotain sekä lukea aikaansaannoksemme:

Esimerkki 7.1. Tiedoston avaaminen, kirjoittaminen ja lukeminen

```
# -*- coding: utf-8 -*-
# Tiedosto: tiedtesti.py

# Huomaa, että käytämme tässä kolmea lainausmerkkiä määrittämään
# tekstinpätkän, jossa on useita rivejä. Toimii kuten docstring.
teksti = '''\
Balin palapelitehdas
pisti pillit pussiin pian
pelipalojen palattua piloille pelattuina:
      pelipalojen puusta puuttui pinnoite!
'''

tiedosto = open("uutinen.txt", "w", encoding="utf-8") # avataan
kirjoitusta varten
tiedosto.write(teksti) # kirjoitetaan teksti tiedostoon
tiedosto.close() # suljetaan tiedosto

tiedosto = open("uutinen.txt", "r", encoding="utf-8") # avataan tiedosto
lukemista varten
while True:
    rivi = tiedosto.readline() # Luetaan tiedostosta rivi
    if len(rivi) == 0: # Jos rivin pituus on 0, ollaan lopussa
      break
    print(rivi, end = "")
tiedosto.close() # suljetaan tiedosto
```

Tuloste

```
>>>
Balin palapelitehdas
pisti pillit pussiin pian
pelipalojen palattua piloille pelattuina:
      pelipalojen puusta puuttui pinnoite!
>>>
```

Kuinka se toimii

Tällä kertaa esimerkki sisältää niin paljon uutta asiaa, että tämä tehtävä käydään läpi rivi riviltä. Ensimmäiset kaksi riviä ovat normaaleja kommenttirivejä, samoin ensimmäinen looginen rivi on vanhastaan tuttu; sillä ainoastaan tallennetaan testiteksti muuttujaan, joskin nyt käytämme kolmen heittomerkin syntaksia. Tämän jälkeen pääsemme ensimmäiselle tiedostoja käsittelevälle riville: `tiedosto = open("uutinen.txt", "w", encoding="utf-8")`. Ensinnäkin, avaamme tiedoston `uutinen.txt` tiedostokahvaan `tiedosto`, joka saa arvona tiedostojen avaamisessa käytetyn funktion `open` palautusarvon. `open`-funktiota kutsutaan antamalla sille kolme parametriä, joista ensimmäinen on avattavan tiedoston nimi, tässä tapauksessa merkkijono `uutinen.txt`. Toinen parametri on tila, johon tiedosto halutaan avata, tässä tapauksessa ‘`w`’, eli kirjoitustila (eng. write). Kirjoitusmoodi ei erikseen tarkasta, onko aiempaa tiedostoa nimeltä “uutinen.txt” olemassa; jos tiedosto löytyy, se korvataan uudella tyhjällä tiedostolla, ja jos taas ei ole, sellainen luodaan automaattisesti. Koska ajamme lähdekoodin suorittamalla tiedoston `tiedtesti.py`, on oletuskansio automaattisesti sama kuin se, mihin ko. tiedosto on tallennettu. Kolmas parametri kertoo käytettävän koodauksen, jolla tiedostoon kirjoitetaan merkkejä. Tässä esimerkissä käytämme yleismaailmallista UTF-8-koodausta.

Seuraavalla rivillä, `tiedosto.write(teksti)`, kirjoitamme tiedostoon muuttujan `teksti` sisällön. `write`-funktio toimii kuten muutkin vastaavat funktiot; sille voidaan antaa parametrina joko sitaatein merkittyjä merkkijonoja tai muuttujan arvoja, jotka se tämän jälkeen kirjoittaa muuttujan `tiedosto` kohdetiedostoon. Tässä kohtaa kannattaa kuitenkin huomioida kolme asiaa:

- Ensinnäkin, tiedostokahva `tiedosto` ei varsinaisesti vielä laske meitä käsiksi tiedoston sisältöön, vaan on ainoastaan ”kulkuyhteys” tiedoston sisälle. Tiedostoa itsessään manipuloidaan kahvan kautta, ja siksi funktiot kuten `write` tai `close` merkataan pistenotaation avulla, koska ne ovat tiedostokahvan jäsenfunktioita. Tässä vaiheessa tärkeintä on kuitenkin vain muistaa lisätä piste muuttujannimen ja funktion väliin.

- Toiseksi kannattaa muistaa, että olemme avanneet tiedoston nimenomaisesti kirjoittamista varten. Nyt voimme kirjoittaa vapaasti, mitä haluamme, mutta jos tila olisi valittu toisin – kuten esimerkiksi lukutila – kirjoitusyritys aiheuttaisi virheilmoituksen.

- Kolmas asia on se, että **Python voi kirjoittaa tiedostoihin ainoastaan merkkijonoja.** Tämä tulee ottaa kirjaimellisesti; jos haluat tallentaa lukuarvoja tiedostoihin, joudut ennen kirjoitusta muuttamaan ne merkkijonoiksi. Tämä luonnollisesti onnistuu tyyppimuunnosfunktio `str`:llä.

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

Ohjelma jatkaa suoritustaan avaamalla tiedoston uudelleen, tällä kertaa lukutilaan moodilla '`r`' (eng. read). Jos tiedoston avaamisessa ei anneta erikseen tilaa, johon tiedosto avataan, olettaa Python sen olevan lukutila '`r`'. Lukutila on paljon virheherkempi kuin kirjoitustila, koska se palauttaa tulkille virheen, mikäli kohdetiedostoa ei ole olemassa. Tämä on varsin loogista, jos ajattelemme asiaa tarkemmin: jos tiedostoa ei ole olemassa, ei meillä varmaankaan ole myös mitään luettavaa. Koska juuri loimme tiedoston, voimme olla varmoja, että se on olemassa ja ohjelma jatkaakin itse lukuvaiheeseen.

Luemme tiedoston rivi riviltä funktiolla `readline`, joka palauttaa tiedostosta merkkijonon, joka edustaa yhtä tiedoston fyysistä riviä. Tässä tapauksessa sillä tarkoitetaan riviä, joka alkaa tiedoston kirjanmerkin kohdalta (eli arvosta, joka kertoo missä kohdin tiedostoa olemme menossa) ja päättyy joko rivinvaihtomerkkiin tai tiedoston loppuun. Jos kirjanmerkki on valmiiksi tiedoston lopussa, palauttaa funktio tyhjän merkkijonon, jonka pituus siis on 0, ja tällöin tiedämme, että voimme lopettaa lukemisen. Sama toistorakenne myös vastaa tiedoston sisällön tulostamisesta.

Lopuksi vielä suljemme tiedoston viimeisen kerran, jonka jälkeen ohjelman suoritus on päättynyt, ja voimme tulostuksesta todeta, että kirjoittaminen ja lukeminen onnistui juuri niin kuin pitikin. Voit myös etsiä tiedoston koneeltasi ja todeta, että kovalevyllä oleva tiedosto ”uutinen.txt” sisältää juurikin saman tekstin kuin mikä juuri tulostui ruudulle.

Avaustiloista

Kuten edellisestä esimerkistä huomasit, toimii Pythonin tiedostojen avaustilojen (tai moodien) avulla. Pythoniin on sisäänrakennettuna useita erilaisia tiloja, joista yleisimmät, kirjoittamisen ja lukemisen, oletkin jo nähnyt.

Muista tiloista voidaan mainita mm. lisäystila '`a`' (eng. append), joka toimii samantapaisesti kuin kirjoitustila '`w`'. Niillä on kuitenkin yksi tärkeä ero; kun '`w`' poistaa aiemman tiedoston ja luo tilalle samannimisen mutta tyhjän tiedoston, mahdollistaa '`a`' sisällön lisäämisen tiedostoon ilman, että aiempaa tiedostoa tuhotaan. Tilaan '`a`' avattaessa tiedoston kirjanmerkki siirtyy automaattisesti tiedoston loppuun ja jatkaa kirjoittamista aiemman tekstin perään.

Lisäksi Pythonista löytyy myös luku- ja kirjoitustila `r+`, johon avattaessa tiedostoon voidaan sekä kirjoittaa että lukea tietoa. Tämä kanssa tulee kuitenkin huomioida lukupaikan sijainti, joka on ratkaiseva silloin, kun halutaan olla varma siitä, mitä tiedostoon lopulta päätyy. Yleisesti ottaen onkin kannattavaa ennemmin suunnitella ohjelma niin, että tiedostoon voidaan kerralla ainoastaan joko kirjoittaa, tai sieltä voidaan lukea.

Pythonista löytyy myös muita tiloja mm. binäärisen datan käsittelylle, ja niihin palataan luvussa 12. Voit myös halutessasi lukea lisätietoa avaustiloista vaikkapa Python Software Foundationin dokumenteista.

Kirjanmerkistä

Huomasit varmaan, että yllä olevissa kappaleissa puhuttiin mystisestä kirjanmerkistä. Käytännössä kirjanmerkki on ainoastaan tieto siitä, missä kohdin tiedostoa olemme etenemässä.

Jos luemme rivin verran tekstiä funktiolla readline, kirjanmerkki siirtyy yhden rivin eteenpäin seuraavan fyysisen rivin alkuun. Tämän kirjanmerkin ansiosta voimme lukea tietoa rivi kerrallaan siten, että saamme aina uuden rivin. Jos taas kirjoitamme tiedostoon, aloitetaan tiedostoon kirjoittaminen siitä kohtaa, mihin kirjanmerkki on sijoittunut. Esimerkiksi lisäystila `'a'` siirtää kirjanmerkin automaattisesti tiedoston loppuun. Kirjanmerkkiä voi myös siirtää käsin funktiolla `fseek`, joka esitellään paremmin myöhemmin tässä luvussa.

Erilaisia merkkien koodaustapoja

Esimerkissä 7.1 käytimme UTF-8 merkistökoodausta, kun kirjoitimme tekstiä tiedostoon ja luimme vastaavan tekstin takaisin käyttöön. Mikäli emme olisi laittaneet `encoding`-parametria ollenkaan, olisi Python käyttänyt käyttöjärjestelmän tarjoamaan oletusarvoa. Tästä olisi seurannut se ongelma, ettei tallennettu tiedosto olisi enää ollut välttämättä luettavissa toisilla käyttöjärjestelmillä, jotka käyttäisivät oletuksena jotain muuta merkistökoodausta kuin mitä me käytämme. Nyt, kun valitsimme käyttöön universaalin koodauksen, pystyy ohjelmalla tehtyjä tiedostoja käyttämään niin Japanissa, Ukrainassa kuin Suomessakin.

Muita työkaluja tiedostonkäsittelyyn

Tiedoston lukemiseen on olemassa useita erilaisia lähestymistapoja. Yksinkertaisin ja samalla avoimin niistä on `read(`*`koko`*`)`, joka lukee tiedostoa ja palauttaa sen sisällön merkkijonona. Parametri *koko* voidaan syöttää, jos halutaan tietynkokoisia viipaleita – koko annetaan integer-lukuna, joka tarkoittaa merkkien määrää - mutta mikäli sitä ei anneta, palautetaan tiedoston sisältö alusta loppuun asti. Jos tiedosto sattui olemaan suurempi kuin koneen keskusmuisti, se on käyttäjän ongelma. `read` onkin juuri tämän takia epäluotettava funktio, jos käsitellään tuntematonta datajoukkoa tai suuria määriä tietueita. Jos kirjanmerkki on valmiiksi tiedoston lopussa tai tiedosto on tyhjä, palauttaa `read` tyhjän merkkijonon.

```
>>> f.read()
'Tiedosto oli tässä.\n'
>>> f.read()
''
```

`readline` on edellä olleesta esimerkistä tuttu funktio. Se palauttaa tiedostosta yhden fyysisen rivin, joka siis alkaa rivinvaihdon jälkeisestä merkistä – tai tiedoston alusta – ja jatkuu rivinvaihtomerkkiin – tai tiedoston loppuun. Funktiota käytettäessä kannattaa muistaa, että tiedostosta luettavaan riviin jää viimeiseksi merkiksi rivinvaihtomerkki aina paitsi silloin, jos luetaan tiedoston viimeinen rivi ja sen perässä ei rivinvaihtomerkkiä ole. Tiedoston lopun saavutettuaan `readline` palauttaa tyhjän merkkijonon.

```
>>> f.readline()
'Tämä on tiedoston ensimmäinen rivi.\n'
>>> f.readline()
'Tämä on tiedoston toinen rivi.\n'
>>> f.readline()
''
```

`readlines` on variaatio `readline`-funktiosta. Se palauttaa koko tiedoston rivit kirjanmerkin sijainnista tiedoston loppuun yhteen listaan tallennettuna, mikä käytännössä tarkoittaa sitä, että funktio kerran ajettaessa palauttaa koko tiedoston sisällön. Tälle funktiolle voidaan antaa parametri *sizehint*, joka määrää kuinka monta merkkiä tiedostosta ainakin luetaan, sekä sen päälle riittävästi merkkejä, jotta fyysinen rivi saadaan täyteen. Jos haluamme lukea kokonaisia tiedostoja muistiin, on tämä lähestymistapa huomattavasti `read`-funktiota parempi, koska notaation ansiosta pääsemme suoraan käsiksi haluttuihin riveihin.

```
>>> f.readlines()
['Tämä on tiedoston ensimmäinen rivi.\n', 'Tämä on tiedoston toinen
rivi.\n']
```

Vaihtoehtoinen tapa lähestyä tiedoston sisällön tulostamista on myös esimerkissä esitelty toistorakenne. Tämä tapa säästää muistia ja on yksinkertainen sekä nopea. Jos oletetaan, että sinulla on olemassa tiedostokahva `tiedosto`, voit tulostaa sen sisällön helposti komennolla:

```
>>> for rivi in tiedosto:
        print(rivi)

Tämä on tiedoston ensimmäinen rivi.

Tämä on tiedoston toinen rivi.
```

Tällä tavoin et kuitenkaan voi säädellä tiedoston sisällön tulostusta kuin rivi kerrallaan, eikä tiedoston sisältö tallennu minnekään jatkokäyttöä varten.

tiedosto.write(*merkkijono*) kirjoittaa merkkijonon sisällön tiedostoon, joka on avattu kahvaan `tiedosto`.

```
>>> tiedosto.write("Tämä on testi\n")
```

Kuten esimerkissäkin mainittiin, Python ei osaa kirjoittaa tiedostoon kuin vain ja ainoastaan merkkijonoja. Kaikki muut tietotyypit on ensin muunnettava merkkijonoiksi.

```
>>> arvo = 42
>>> sailio = str(arvo)
>>> f.write(sailio)
```

`tell`–funktio palauttaa integer-arvon, joka kertoo kuinka monta merkkiä tiedoston alusta on tultu, eli siis kuinka monta merkkiä tiedostosta on luettu. Tämä tarkoittaa siis kirjanmerkin senhetkistä sijaintia. Vastaavasti funktio `seek(`*`mihin`*`, `*`mista_laskettuna`*`)` siirtää kirjanmerkkiä haluttuun paikkaan. Uusi paikka lasketaan lisäämällä parametrin *`mihin`* arvo kiintopistettä kuvaavaan parametriin *`mista_laskettuna`*. Arvo 0 tarkoittaa tiedoston alkua, 1 tarkoittaa nykyistä sijaintia ja 2 tiedoston loppua, oletusarvoisesti *`mista_laskettuna`* on 0. Huomaa, että *`mihin`*-parametrille voi myös antaa negatiivisen arvon.

```
>>> f = open("/tmp/workfile", "r+", encoding="utf-8")
>>> f.write("0123456789abcdef")
>>> f.seek(5)     # Mene tiedoston 6. tavuun
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Mene tiedoston 3. viimeiseen tavuun
>>> f.read(1)
'd'
```

Ja kun olet valmis, `close`-funktio sulkee tiedoston ja vapauttaa tiedosto-operaatioita varten varatut resurssit takaisin käyttöjärjestelmän käyttöön.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Tiedostojen käsittelyyn on myös muita työkaluja, mutta niiden käyttö on nyt esiteltyihin nähden marginaalista. Jos haluaisit tutustua niihin tarkemmin, löytyy niistäkin kattava esittely Python Software Foundationin dokumentaatioista luvussa joka käsittelee I/O-operaatioita.

Useamman tiedoston avaaminen samanaikaiseen käyttöön

Usein tiedostoja käsiteltäessä tulee tarve samanaikaisesti lukea jostain tiedostosta ja kirjoittaa toiseen. Tämä ei kuitenkaan Pythonilla ole mikään ongelma, sillä avonaisten tiedostojen lukumäärää ei ole mitenkään rajoitettu, vaan tiedostoja voidaan ottaa käyttöön niin paljon kuin tarvitaan. Esimerkki selventää asiaa.

Esimerkki 7.2. Tiedoston avaaminen, kirjoittaminen ja lukeminen

```
# -*- coding: utf-8 -*-
# Tiedosto: tiedostot.py

tiedosto_luku = open("merkkijonoja.txt", "r", encoding="utf-8")
tiedosto_kirjoitus = open("lyhyet.txt", "w", encoding="utf-8")

while True:
    rivi = tiedosto_luku.readline()
    if len(rivi) == 0:
        break
    if len(rivi) < 10:
        tiedosto_kirjoitus.write(rivi)

tiedosto_luku.close()
tiedosto_kirjoitus.close()
```

Tuloste

Ohjelma ei tulosta mitään.

Kuinka se toimii

Ohjelman tarkoituksena on lukea yhdestä tiedostosta rivejä ja kirjoittaa alle 10 merkkiä sisältävät rivit toiseen tiedostoon. Aluksi avataan kaksi tiedostokahvaa: toiseen luettava tiedosto ja toiseen kirjoitettava tiedosto. Tämän jälkeen `while`-silmukassa luetaan rivi avatusta tiedostosta, tarkistetaan ollaanko tiedoston lopussa (==tyhjä rivi) ja sen jälkeen kirjoitetaan rivi toiseen tiedostoon, mikäli sen pituus on alle 10 merkkiä. Lopuksi molemmat tiedostokahvat suljetaan.

Merkkijonojen metodit

Tässä kappaleessa käymme läpi hieman pidemmälle meneviä tapoja työskennellä merkkijonoilla. Python-ohjelmointikielen perusoperaatioiden – leikkaukset, liitokset – lisäksi voimme käyttää joukkoa kehittyneempiä metodeja, joilla merkkijonoille voidaan tehdä monia muuten vaikeita operaatioita. Tässä lista metodeista, jotka Python-tulkki tunnistaa:

capitalize()

Palauttaa merkkijonosta kopion, jossa ensimmäinen merkki on muutettu isoksi kirjaimeksi.

endswith(*testi*[, *start*[, *end*]]**)**

Palauttaa arvon `True`, jos merkkijono päättyy merkkeihin *testi*, muutoin palauttaa arvon `False`. Lisäkytkimet start ja end, joilla voidaan määrätä testattava alue.

startswith(*prefix*[, *start*[, *end*]]**)**

Sama kun endswith() mutta testaa merkkijonon alkua.

expandtabs([*tabsize*]**)**

Palauttaa merkkijonosta kopion, jossa kaikki sisennysmerkit on korvattu tabsize –määrällä välilyöntejä. Tabsize on oletusarvoisesti 8. (Huom! IDLEllä 4)

find(*sub*[, *start*[, *end*]]**)**

Palauttaa ensimmäisen merkin sijainnin, jos annettu merkkijono *sub* löytyy testijonosta. Lisäkytkimet start ja end mahdollistavat hakualueen rajaamisen. Palauttaa arvon -1, jos merkkijonoa ei löydy.

index(*sub*[, start[, end]]**)**

Kuin find(), mutta palauttaa virheen ValueError, jos merkkijonoa *sub* ei löydy.

isalnum()

Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat joko kirjaimia tai numeroita (alphanumeerisia) ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.

isalpha()

Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat kirjaimia ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.

isdigit()

Palauttaa arvon `True`, jos kaikki testattavan merkkijonon merkit ovat numeroita ja merkkijonon pituus on 1 tai enemmän. Muussa tapauksessa palauttaa arvon `False`.

islower()

Palautttaa arvon `True`, jos kaikki merkkijonon merkit ovat pieniä kirjaimia ja merkkijonossa on ainakin yksi kirjain. Muutoin `False`.

isupper()

Sama kuin islower(), mutta isoille kirjaimille.

isspace()

Palautttaa arvon `True`, jos kaikki merkkijonon merkit ovat välilyöntejä ja merkkijonossa on ainakin yksi merkki. Muutoin `False`.

lower()

Palauttaa merkkijonon kopion, jossa kaikki kirjaimet on muutettu pieniksi kirjaimiksi.

lstrip([chars]**)**

Palauttaa merkkijonon, josta on vasemmasta reunasta poistettu kaikki määritellyt merkit. Jos poistettavia merkkejä ei määritellä, poistetaan pelkästään välilyönnit ja sisennykset. Tasaus on aina vasemmassa reunassa, ensimmäinen ei-poistettava merkki määrää tasauksen paikan:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

Oikean reunan tasaamiseen käytetään vastaavaa funktiota **rstrip()**.

strip([*chars*]**)**

Palauttaa merkkijonosta kopion, josta on poistettu merkkijonon alusta ja lopusta kaikki määritellyt merkit. Käytännössä yhtäaikainen lstrip() ja rstrip(). Jos poistettavia merkkejä ei määritellä, poistetaan välilyönnit ja sisennykset.

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmow.')
'example'
```

replace(*old*, *new*[, *count*]**)**

Korvaa merkkijonosta merkkiryhmän *old* jäsenet merkkiryhmällä *new*. Voidaan myös antaa lisätietona parametri *count*, joka määrää kuinka monta korvausta maksimissaan tehdään.

split([sep[, *maxsplit*]]**)**

Halkaisee merkkijonon erotinmerkin mukaisesti listaksi. Voi saada myös parametrin *maxsplit*, joka kertoo miten monesta kohtaa merkkijono korkeintaan halkaistaan (eli listassa on silloin *maxsplit*+1 alkioa).

```
'1,,2'.split(',') ----> ['1', '', '2']
'1:2:3'.split(':') ----> ['1', '2', '3']
'1, 2, 3'.split(', ') ----> ['1', '2', '3']")
```

Jos erotinmerkkiä ei anneta, käytetään oletuserottimena välilyöntiä.

upper()

Muuttaa kaikki merkkijonon kirjaimet isoiksi kirjaimiksi.

Esimerkki 7.3. Merkkijonojen kanssa kikkailua

```
# -*- coding: utf-8 -*-
# Tiedosto: mjonot.py

mjono1 = "Tässä on meille tekstiä.  "
mjono2 = input("Anna merkkijono: ")

print(mjono1, mjono2)
print(mjono1.strip(), mjono2)
print(mjono1.upper(), mjono2.upper())
if mjono2.isalpha():
    print("mjono2 käsittää vain kirjaimia.")
print("mjono1 jaettuna välilyöntien kohdalta listaksi:", mjono1.split())
```

Tuloste

```
>>>
Anna merkkijono: FooBar
Tässä on meille tekstiä.   FooBar
Tässä on meille tekstiä. FooBar
TÄSSÄ ON MEILLE TEKSTIÄ.   FOOBAR
mjono2 käsittää vain kirjaimia.
mjono1 jaettuna välilyöntien kohdalta listaksi: ['Tässä', 'on',
'meille', 'tekstiä.']
>>>
```

Esimerkin tarkoitus on demota, kuinka merkkijonojen metodeja voidaan käyttää eri tavoilla. Tapoja ja metodeja on vielä paljon enemmän, joten ongelma kuin ongelma on ratkaistavissa. Lisää informaatiota löytyy Pythonin dokumentaatiosta kohdasta String Methods.

Muotoiltu tulostus

Python-ohjelmointikielessä on myös lisätoimintoja, joiden avulla pystymme tarkentamaan sitä, kuinka ohjelma tulostaa merkkijonoja. Tätä sanotaan formatoiduksi eli muotoilluksi tulostukseksi ja tässä käytämme hyväksi merkkijonoille tarjolla olevaa jäsenfunktiota `format`.

Joissain tapauksissa saatamme haluta, että tulostettava merkkijono noudattaa jotain tiettyä kaavaa huolimatta siitä, millainen varsinainen tulos on. Esimerkiksi ”nelinumeroinen luku” tai ”kaksi-desimaalinen liukuluku” ovat hyviä esimerkkejä tästä. Teknisesti tämä toteutetaan siten, että `print`-funktiolle annettavaan tulostussyötteeseen sijoitetaan muotoilumerkit {}-merkeillä erotettuna, ja syötteen perään sijoitetaan `.format(muuttujalista)`. Esimerkiksi:

```
sana = "maitomies"
print(sana, "ajaa maitoautoa.")
```

Muuttuu nyt muotoon:

```
print("{0} ajaa maitoautoa.".format(sana))
```

Siitä huolimatta molempien tuloste on identtinen:

```
maitomies ajaa maitoautoa.
```

Tulosteeseen sijoitettu merkki {0} tarkoittaa, että tämän merkin paikalle tullaan sijoittamaan muuttujalistan *ensimmäinen* arvo. Entäpä, jos haluamme mukaan toisenkin arvon?

```
sana = "maitomies"
luku = 13

print("{0} käy reitillään läpi {1} taloa.".format(sana, luku))
```

Tällä saisimme vastaukseksi tulosteen:

```
maitomies käy reitillään läpi 13 taloa.
```

Muotoillun tulostuksen tärkeimpiä etuja on sen kyky käsitellä numeroarvoja. Jos haluamme käyttää liukulukuja, voimme määritellä muotoilumerkeillä kuinka monta desimaalia haluamme säilyttää mukana:

```
luku = 13.2733385
print("{0:.0f} {0:.1f} {0:.2f} {0:.5f} ".format(luku))
```

Tämä antaa tuloksena seuraavanlaisen tulosteen:

```
13 13.3 13.27 13.27334
```

Voimme siis säädellä vapaasti, kuinka monta desimaalia luvusta näytetään. Kannattaa lisäksi huomata, että kyseinen menetelmä osaa lisäksi pyöristää luvut oikein.

Vastaavasti, jos lisäämme pisteen eteen luvun, kerromme kuinka monta merkkiä tilaa luvulle on vähintään varattava:

```
luku = 13.2733385
print("{0:.0f} {0:1.1f} {0:7.2f} {0:20.5f} ".format(luku))
```

Tulostaa:

```
13 13.3   13.27              13.27334
```

Kuten huomaamme, tämä ei koske tilannetta, jossa luku on pidempi kuin vähintään varattava tila. Kuten esimerkistä huomaamme, voimme käyttää yhtä aikaa numeroa sekä pisteen edessä että takana.

Muotoillun tulostuksen voima tulee esiin, kun haluamme esittää numeroita käyttäjän haluamassa muodossa.

```
luku = 1442.2465274
tarkkuus = 5
print("{0:.{1}f} ".format(float(luku), tarkkuus))
```

Tässä esimerkissä tulostamme `luku`-muuttujan `tarkkuus`-muuttujan osoittamalla tarkkuudella, eli viidellä desimaalilla. {}-merkkien sisällä annetaan muotoilun

tarkkuudeksi {1}, eli muuttujalistan toinen alkio, `tarkkuus`. Huomaa myös, että `luku` muunnetaan `float`iksi, joten vaikka se olisi alun perin ollut kokonaisluku, niin muotoiltu tulostus onnistuu nyt silläkin.

Luvun asiat kokoava esimerkki

Esimerkki 7.4. Tiedostojen käsittelyä

```
# -*- coding: utf-8 -*-
# Tiedosto: potenssit.py

tiedosto_luku = open("numeroita.txt", "r", encoding="utf-8")
tiedosto_kirjoitus = open("potensseja.txt", "w", encoding="utf-8")
while True:
    rivi = tiedosto_luku.readline()
    if len(rivi) == 0:
        break
    luku = int(rivi[:-1])
    potenssi = luku ** 2
    tiedosto_kirjoitus.write(str(potenssi) + "\n")

tiedosto_luku.close()
tiedosto_kirjoitus.close()
```

Luku 8: Rakenteiset tietotyypit lista ja kumppanit

Tietorakenteet ovat yksinkertaisesti ilmaistuna juurikin itseään tarkoittava asia – eli ne ovat rakenteita, jotka sisältävät tietoa. Yleisesti niiden käytettävyys perustuu siihen, että ne sisältävät tietoa, joka on jollain tavalla toisiinsa liittyvää tai osa samaa kokonaisuutta.

Dynaamisen muistinhallintansa ansiosta Python sisältää erityisen helppokäyttöisiä ja tehokkaita tietorakenteita: listan, tuplen, luokan sekä sanakirjan. Tässä kappaleessa keskitymme erityisesti listaan, mutta käymme läpi myös luokan perusajatukset. Lopuksi esittelemme lyhyesti myös joitakin epätavallisempia tietorakenteita.

Lista

Lista on Pythonin perustietorakenne, joka sisältää joukon alkioita. Tähän mennessä olemme tutustuneet muuttujiin, jotka ovat pystyneet pitämään sisällään vain yhden alkion kerrallaan, esimerkiksi numeron 42 tai sanan ”suklaakakku”. Lista pystyy pitämään sisällään useita numeroita tai merkkijonoja ja jopa niiden sekoituksia. Pythonin lista vastaa reaalimaailman vastaavaa. Jos kirjoitat paperille listaa asioista, esim. ostoslistaa, niin todennäköisesti listaat asiat allekkain. Pythonin listassa eri alkiot erotellaan hieman erilailla, mutta idea on kuitenkin sama. Lopputuloksena meillä on alkioista koostuva kokonaisuus, jota kutsutaan listaksi.

Listan määrittelyssä listan alku- ja loppukohta merkitään hakasuluilla. Tämä antaa tulkille ilmoituksen siitä, että haluat määritellä listan, ja että annat sille pilkulla eroteltuja alkioita. Kun olet luonut listan, voidaan siihen tämän jälkeen lisätä alkioita, poistaa alkioita, muuttaa järjestystä sekä hakea alkioita. Tämä siis tarkoittaa, että listaa voidaan muokata vapaasti toisin kuin esimerkiksi merkkijonoja. Lisäksi listan käsittelyyn on olemassa muutamia tehokkaita aputoimintoja – jäsenfunktioita, toiselta nimeltään metodeja – joita käymme kohta lävitse.

Tämä varmasti kuulosti varsin sekavalle, mutta ei hätää, esimerkki selkeyttää taas asiaa kummasti.

Esimerkki 8.1. Listan käyttäminen

```
# -*- coding: utf-8 -*-
# Tiedosto: lista.py

# Ostoslistan määrittely
ostoslista = ["vesi", "sokeri", "sitruuna", "hiiva"]

print("Tarvitsen vielä", len(ostoslista), "tuotetta.")

print("Nämä tuotteet ovat:", end = " ")
for tuote in ostoslista:
    print(tuote, end = " ")

print("\nTarvitsen myös ämpärin.")
ostoslista.append("ämpäri")
print("Nyt ostoslista näyttää tältä", ostoslista)

print("Järjestellään ostoslista")
ostoslista.sort()
print("Järjestelty ostoslista on tämän näköinen:", ostoslista)

print("Ensimmäinen ostettava tuote on", ostoslista[0])
ostettu = ostoslista[0]
del ostoslista[0]
print("Ostin tuotteen", ostettu)
print("Nyt ostoslistalla on jäljellä", ostoslista)
```

Tuloste

```
>>>
Tarvitsen vielä 4 tuotetta.
Nämä tuotteet ovat: vesi sokeri sitruuna hiiva
Tarvitsen myös ämpärin.
Nyt ostoslista näyttää tältä ['vesi', 'sokeri', 'sitruuna', 'hiiva',
'ämpäri']
Järjestellään ostoslista
Järjestelty ostoslista on tämän näköinen: ['hiiva', 'sitruuna',
'sokeri', 'vesi', 'ämpäri']
Ensimmäinen ostettava tuote on hiiva
Ostin tuotteen hiiva
Nyt ostoslistalla on jäljellä ['sitruuna', 'sokeri', 'vesi', 'ämpäri']
>>>
```

Kuinka se toimii

Muuttuja `ostoslista` on lista tuotteista, joita halutaan hankkia. Lista sisältääkin tietueinaan merkkijonoja, joihin on tallennettu ostettavien tuotteiden nimet. Periaatteessa tämä ei perustu mihinkään rajoitteeseen; listalle voi tallentaa alkioihin millaista tietoa tahansa, mukaan lukien esimerkiksi numerot tai vaikkapa toiset listat.

Muuttujan rooli: säiliö

`ostoslista`-muuttujamme pitää sisällään listaa kaupasta ostettavista tuotteista. Sen roolina on siis toimia *säiliönä.*

Huomioi myös, että käytimme hieman tavallisuudesta poikkeavaa `for...in`-toistorakennetta listan läpikäymiseen. Tämä siis tarkoittaa sitä, että lista on eräänlainen sarja (sequence), joten sitä voidaan käyttää yksinään `for`-lauseen määrittelemisessä. Jos annamme `range`-funktion tilalle listan, `for`-lause käy läpi kaikki

listan alkiot.

Seuraavaksi lisäämme listalle siitä alun perin pois jääneen alkion jäsenfunktiolla `append`. Huomioi metodikutsun pistenotaatio: `ostoslista.append("ämpäri")`. Kaikki listojen metodit merkitään piste-erottimella samoin kuin teimme tiedosto-operaatioiden kanssa. Tapahtuneet muutokset tarkastimme antamalla `print`-funktiolle listan. Huomaa myös, että `print`-funktion kanssa lista tulostuu sarjamuodossaan, jolloin sulut, sitaattimerkit sekä pilkut jäävät alkioiden väliin.

Tämän jälkeen suoritamme listan järjestelemisen jäsenfunktiolla `sort`, joka asettelee listan alkiot arvojärjestyksen mukaisesti pienimmästä suurimpaan. Tässä tapauksessa listan järjestys näyttää päällisin puolin olevan suoraan aakkosjärjestyksen mukainen, mutta tässä asiassa on joitakin poikkeuksia. Puhumme poikkeuksista enemmän hieman edempänä. Kannattaa myös huomata, että tässä tapauksessa järjesteltyä listaa ei tarvitse erikseen tallentaa uuteen muuttujaan. Kaikki metodit vaikuttavat suoraan siihen listaan, jolla niitä suoritetaan, toisin kuin esimerkiksi tyyppimuunnosten yhteydessä.

Seuraavaksi sijoitamme yhden alkion arvon muuttujalle. Tämä tapahtuu samanlaisella notaatiolla kuin esimerkiksi yksittäisen merkin ottaminen merkkijonosta. Lisäksi samoin kuin merkkijonojen kanssa, ensimmäisen alkion järjestysnumero on 0. Listan kanssa operoidessa leikkausten suorittaminen onnistuu täsmälleen samoin kuin merkkijonoilla: kun merkkijonoissa leikkaamme merkkejä, listoissa leikkaamme alkioita. Lisäksi alkionsisäisen leikkaukset suoritetaan ensin valitsemalla alkio, ja tämän jälkeen sille tehtävä leikkaus. Ensimmäisen alkion viisi ensimmäistä merkkiä olisi siis `ostoslista[0][0:5]`.

Lopuksi poistamme listalta yhden alkion. Tämä tapahtuu `del` -käskyllä. Yksinkertaisesti annamme `del`-käskylle arvoksi sen listan alkion, jonka haluamme poistaa, jolloin listan arvoksi jää uusi lista ilman ko. arvoa. Lisäksi lista muuttuu siten, että poistetun arvon `ostoslista[0]` paikalle ei jää tyhjää alkiota, vaan se yksinkertaisesti täytetään siirtämällä kaikkia seuraavia alkioita yksi paikka taaksepäin.

Esimerkki 8.2. Listan leikkaukset

```
# -*- coding: utf-8 -*-
# Tiedosto: lista_l.py

# Ostoslistan määrittely ja alustus
lista = ["omena", "mango", "banaani", "persikka"] # huomioi hakasulut []

print("alkio 0 on", lista[0])
print("alkio 1 on", lista[1])
print("alkio 2 on", lista[2])
print("alkio 3 on", lista[3])
print("alkio -1 on", lista[-1])
print("alkio -2 on", lista[-2])
# Muistathan, ettei leikkauksen päätepiste kuulu mukaan
print("alkiot 1-3 ovat", lista[1:3])
print("alkiot 2 eteenpäin ovat", lista[2:])
print("alkiot 1 -> -1 ovat", lista[1:-1])
print("kaikki alkiot yhdessä:", lista[:])
```

Tuloste

```
>>>
alkio 0 on omena
alkio 1 on mango
alkio 2 on banaani
alkio 3 on persikka
alkio -1 on persikka
alkio -2 on banaani
alkiot 1-3 ovat ['mango', 'banaani']
alkiot 2 eteenpäin ovat ['banaani', 'persikka']
alkiot 1 -> -1 ovat ['mango', 'banaani']
kaikki alkiot yhdessä: ['omena', 'mango', 'banaani', 'persikka']
>>>
```

Lisäksi kannattaa huomioida, että `del`-käsky osaa poistaa alkioita kokonaisina leikkauksina:

```
>>> lista = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del lista[0]
>>> lista
[1, 66.25, 333, 333, 1234.5]
>>> del lista[2:4]
>>> lista
[1, 66.25, 1234.5]
```

`del` voi myös poistaa koko listan yhdellä kertaa:

```
>>> del lista
>>> lista

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    lista
NameError: name 'lista' is not defined
```

Yleisimpiä listan jäsenfunktioita

Listan manipulointiin on siis olemassa erilaisia tapoja. Seuraavassa listassa on yleisiä metodipohjaisia tapoja muutella listan sisältöä:

append(*x*)

Lisää alkio x listan loppuun.

extend(*L*)

Lisää listaan kaikki annetun listan L alkiot. Eroaa append:ista siten, että `testi.append(L)` lisäisi listan `testi` viimeiseksi alkioksi listan `L`, kun taas `testi.extend(L)` lisää listan `L` alkiot listan `testi` loppuun.

insert(*i, x*)

Lisää alkion x listalle kohtaan i. Listan alkuun lisääminen siis tapahtuisi käskyllä a.`insert(0, x)`, kun taas loppuun lisääminen – samoin kuin append tekee - tapahtuisi käskyllä `a.insert(len(a), x)`.

remove(*x*)

Poistaa listalta ensimmäisen tietueen, jonka arvo on x, eli siis jossa `x == lista[i] == True`. Palauttaa virheen, mikäli tämän arvoista tietuetta ei ole olemassa.

pop(i)

Poistaa listalta tietueen kohdasta i ja palauttaa sen arvon. Mikäli i on määrittelemätön, poistaa se viimeisen listalla olevan alkion.

index(*x*)

Palauttaa listalta numeroarvon i, joka kertoo millä kohdalla listaa on tietue, jolla on arvo x. Palauttaa virheen mikäli lista ei sisällä tietuetta, jonka arvo on x.

count(*x*)

Palauttaa numeroarvon i, joka kertoo kuinka monta kertaa x esiintyy listalla.

sort()

Järjestää listan alkiot arvojärjestykseen.

reverse()

Kääntää listan alkiot ympäri, eli ensimmäinen viimeiseksi jne.

Esimerkkejä jäsenfunktioiden käyttämisestä:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0

>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]

>>> a.index(333)
1
```

```
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]

>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]

>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Listan käyttö funktiokutsuissa

Tässä vaiheessa Python-opiskelua on hyvä huomata, että tietyissä tilanteissa lista käyttäytyy hieman eri lailla kuin esimerkiksi merkkijono. Siinä, missä merkkijonoa ei pysty muuttamaan, listaa pystyy. Kun funktiolle annetaan parametrinä merkkijono, niin todellisuudessa annetaan vain kopio, mutta listasta siirtyy viittaus alkuperäiseen listaan. Esimerkki selventää:

Esimerkki 8.3. Lista vs. merkkijono

```
# -*- coding: utf-8 -*-
# Tiedosto: mutable.py

def muuta(merkkijono, lista):
    merkkijono = merkkijono + "def"
    lista.append("alkio3")
    print("2:", merkkijono, lista)

m = "abc"
l = ["alkio1", "alkio2"]
print("1:", m, l)
muuta(m, l)
print("3:", m, l)
```

Tuloste

```
>>>
1: abc ['alkio1', 'alkio2']
2: abcdef ['alkio1', 'alkio2', 'alkio3']
3: abc ['alkio1', 'alkio2', 'alkio3']
>>>
```

Kuinka se toimii

Ohjelman aluksi luodaan merkkijono `m` ja lista `l`. Molempiin alustetaan dataa ja tulostetaan molempien sisältö ensimmäisen kerran. Tämän jälkeen kutsutaan `muuta`-funktiota vastikään luoduilla muuttujilla. Funktion sisällä merkkijonoon lisätään merkkejä ja listaan lisätään yksi alkio. Tämän jälkeen merkkijono ja lista tulostetaan, molemmat ovat saaneet selkeästi lisäyksensä.

Arvoparametri ja muttujaparametri

Kun funktiolle annetaan parametrina esimerkiksi integer-luku, saa se käyttöönsä siitä kopion eli arvon, ei alkuperäistä muuttujaa. Kun taas funktiolle annetaan parametrina lista saa se käyttöönsä alkuperäisen listan, eli kokonaisen muuttujan arvoineen. Muuttujaparametri on siis voimakkaampi tapa siirtää dataa kuin arvoparametri.

Mielenkiintoinen osuus alkaa, kun ohjelman suoritus palaa takaisin päätasolle funktiokutsun jälkeen. Huomaa, että funktio ei palauttanut mitään `return`illa. Kolmas tulostus puskee ruudulle taas merkkijonon ja listan sisällön. Merkkijonon sisältö on sama kuin se oli ennen funktiokutsua (funktio ei siis palauttanut tekemiään muutoksia), mutta listassa muutokset näkyvät. Miksi näin?

Syy tähän on seuraava. Kun funktiota kutsutaan merkkijono (tai luku) -parametrilla, niin todellisuudessa funktio saa sisäänsä kopion alkuperäisestä muuttujasta. Tässä tapauksessa merkkijonosta ”abc” annetaan kopio funktiolle – alkuperäinen pysyy muuttumattomana. Listasta ei kuitenkaan anneta kopiota vaan viittaus alkuperäiseen listaan, jolloin funktiossa tapahtuvat muutokset heijastuvat koko ajan alkuperäiseen listaan.

Listan antaminen viittauksena saattaa olla ongelma tietyissä tapauksissa. Mikäli funktion ei ole tarkoitus muuttaa listan sisältöä, vaan esimerkiksi vain tulostaa se aakkosjärjestyksessä, ei `sort`-funktiota voida käyttää suoraan, koska se tekisi muutokset alkuperäiseen listaan. Tähän on olemassa kaksi ratkaisua. Funktiolle voidaan antaa tietoa listan sijaan tuplena, jota ei voida muokata, jolloin ongelmaa ei pääse syntymään. Toinen vaihtoehto on antaa funktiolle kopio listasta.

Esimerkki 8.4. Lista vs. Merkkijono, osa II

```
# -*- coding: utf-8 -*-
# Tiedosto: mutable2.py

def muuta(merkkijono, lista):
    merkkijono = merkkijono + "def"
    lista.append("alkio3")
    print("2:", merkkijono, lista)

m = "abc"
l = ["alkio1", "alkio2"]
print("1:", m, l)
muuta(m, l[:])
print("3:", m, l)
```

Tuloste

```
>>>
1: abc ['alkio1', 'alkio2']
2: abcdef ['alkio1', 'alkio2', 'alkio3']
3: abc ['alkio1', 'alkio2']
>>>
```

Kuinka se toimii

Tällä kertaa annamme päätasolla funktiolle listasta kopion tekemällä siitä leikkauksen, joka sisältää listan kokonaisuudessaan. Funktiokutsun jälkeen alkuperäinen lista on edelleen koskematon ja voimme jatkaa sen kanssa työskentelyä.

Luokka-rakenne

Kaikissa tilanteissa aiemmin esittelemämme tietorakenteet kuten lista tai tuple eivät riitä. Joskus saatamme haluta rakentaa uusia tietotyyppejä, joiden alkioille on annettu oma nimi.

Esimerkiksi, jos haluaisimme tallentaa henkilötietoja, olisi varmaan helpoin tapa tallentaa ne tietotyyppiin, joka on suunniteltu ihmisten henkilötietojen tallentamiseen?

Tähän voimme käyttää rakenteisia tietotyyppejä. Rakenteisilla tietotyypeillä tarkoitetaan tallennusmuotoa, johon käyttäjä itse määrittelee jokaisen alkion nimen, määrän sekä laadun. Esimerkiksi ihmisen tapauksessa voisimme määritellä, että jokaisella ihminen-tietotyypin jäsenellä on etunimi, sukunimi, ikä sekä ammatti. Python-ohjelmointikielessä rakenteisien tietotyyppien määrittelyyn käytetään luokkarakennetta ja sen tunnisteena on avainsana `class`. Seuraavassa esimerkissä tutustumme hieman tarkemmin siihen, kuinka tämä rakenne toimii.

Esimerkki 8.5. Luokka rakenteisena tietotyyppinä

```
# -*- coding: utf-8 -*-
# Tiedosto: luokka.py

# Määritellään ihminen;
# Ihmisellä on etu- ja sukunimi, ikä sekä ammatti.

class Ihminen:
   etunimi = ""
   sukunimi = ""
   ika = 0
   ammatti = ""

# Luodaan uusi ihminen nimeltään mies ja
# annetaan hänelle henkilötiedot

mies = Ihminen()
mies.etunimi = "Kalevi"
mies.sukunimi = "Karvajalka"
mies.ika = 42
mies.ammatti = "pommikoneen rahastaja"

# Tulostetaan miehen tiedot

print(mies.etunimi, mies.sukunimi,"on", mies.ika)
print("vuotta vanha",mies.ammatti)
```

Tuloste

```
>>>
Kalevi Karvajalka on 42
vuotta vanha pommikoneen rahastaja
>>>
```

Kuinka se toimii

Kuten huomaat, koodi aloitetaan käyttämämme tietomuodon määrittelyllä, joka muodoltaan muistuttaa tavallisen funktion rakennetta. Komennolla `class Ihminen:` kerromme Python-tulkille, että aiomme seuraavaksi määritellä uuden tietorakenteen, jonka nimeksi tulee `Ihminen`. Tämän jälkeen voimme määritellä `Ihminen`-tietorakenteeseen kuuluvat jäsenmuuttujat, joita tällä kertaa olemme määritelleet neljä kappaletta: `sukunimi`, `etunimi`, `ammatti` sekä `ika`. Huomioi, että jäsenmuuttujien tyypeillä tai alkuarvoilla ei varsinaisesti ole merkitystä, pääasia on, että jokainen jatkossa käytettävä jäsenmuuttuja on nimetty tässä vaiheessa. Jos määrittelyn yhteydessä olisimme antaneet

jäsenmuuttujalle arvoja, kuten `etunimi = "Niilo"`, käytettäisi näitä oletusarvoina siihen asti, kun määrittelemme niille uuden arvon.

Seuraavassa osiossa otamme luomamme rakenteen käyttöön. Ensiksi määrittelemme, että muuttuja `mies` on tyypiltään luomamme rakenteen `Ihminen` kaltainen. Tämä toteutetaan komennolla `mies = Ihminen()`. Nyt olemme luoneet muuttujan `mies`, jolla on jäsenmuuttujina arvot `sukunimi`, `etunimi`, `ammatti` sekä `ika`. Seuraavilla neljällä rivillä määrittelemme jokaiselle jäsenmuuttujalle arvon. Voimme käyttää jäsenmuuttujia normaalien muuttujien tavoin. Ainoa muistettava asia on, että käytämme pistenotaatiota kertomaan, minkä rakenteen jäsenmuuttujaa muutamme. Jos meillä esimerkiksi olisi kaksi muuttujaa, `mies` ja `nainen`, niin pistenotaation avulla kerromme kumman rakenteen ikää haluamme muuttaa. Periaate on aivan sama kuin esimerkiksi tiedostokahvoja käytettäessä. Asiaa selkeyttääksemme otamme vielä toisen esimerkin.

Esimerkki 8.6. Luokka rakenteisena tietomuotona, esimerkki 2

```
# -*- coding: utf-8 -*-
# Tiedosto: luokka2.py

# Määritellään koira, annetaan sille
# oletusarvot

class Koira:
   rotu = "Sekarotuinen"
   nimi = "Hurtta"

# Luodaan kaksi uutta koiraa
koira_1 = Koira()
koira_2 = Koira()

# Annetaan toiselle koiralle omat tiedot
koira_1.nimi = "Jäyhä"
koira_1.rotu = "Jököttäjä"
print(koira_1.nimi, koira_1.rotu)

# Koiralla 2 on edelleen oletusarvot
print(koira_2.nimi,koira_2.rotu)
```

Tuloste

```
>>>
Jäyhä Jököttäjä
Hurtta Sekarotuinen
>>>
```

Kuinka se toimii

Tällä kertaa määrittelimme rakenteisen luokan `Koira`, josta teimme kaksi muuttujaa nimeltään `koira_1` ja `koira_2`. Tämän jälkeen määrittelimme koiralle 1 jäsenmuuttujiin omat arvot, jotka varmensimme tulostamalla tiedot jälkikäteen. Lopuksi vielä totesimme, että koska emme olleet määritelleet koiralle 2 omia arvoja, oli sillä edelleen oletusarvot, jotka se sai rakenteen määrittelyn yhteydessä. Tärkeää tässä on muistaa se, että rakenteisen tietomuodon jäsenmuuttujissa ei ole mitään erikoista; ne toimivat aivan kuten normaalit muuttujat, ja niitä voidaan käyttää esimerkiksi loogisissa väittämissä – `if-elif-else` – tai toistorakenteissa – `while`, `for` – ehtoina ilman mitään eroavaisuuksia tavallisiin muuttujiin.

Luokan jäsenfunktioista

Koska luokkien yhteydessä puhutaan jäsenmuuttujista, liittyvätkö aiemmin oppaassa käsittelemämme jäsenfunktiot luokkiin jollain tapaa? Periaatteessa vastaus tähän on kyllä. Yleisesti rakenteisissa tietotyypeissä voi olla jäsenmuuttujien lisäksi myös jäsenfunktioita, joiden avulla rakenne joko muokkaa omia tietojaan tai käsittelee saamiansa parametreja. Emme kuitenkaan voi käsitellä jäsenfunktioiden rakennetta ja toimintaa kovinkaan syvällisesti ilman, että joudumme puhumaan olio-ohjelmoinnista, joten tässä oppaassa puhumme jäsenfunktioista hyvin yleisellä tasolla.

Luokkarakenteeseen voidaan siis liittää jäsenfunktioita, joiden avulla ne voivat käsitellä saamiaan parametreja tai omia tietojaan. Esimerkiksi, kun luemme tiedostosta rivin käytämme jäsenfunktiota `readline`.

Kuten sanottu, jäsenfunktiot ja niiden määrittely sisältää pitkälti olio-ohjelmointiin liittyviä aiheita ja asioita, joten niistä emme puhu tässä oppaassa tämän enempää. Jos haluat lukea aiheesta lisää, on esimerkiksi Python Software Foundationin tutoriaalin osassa 9 paljon lisätietoa aiheesta. Python on itse asiassa täysverinen olio-ohjelmointikieli, vaikka tällä kurssilla sitä käytetään ensisijaisesti proseduraaliseen ohjelmointiin.

Muita Python-kielen rakenteita

Lisäksi Python-ohjelmointikielessä on muutama eksoottisempi tietorakenne, joita näkee tavallisesti käytettävän kehittyneemmissä esimerkeissä. Näitä rakenteita emme varsinaisesti tarvitse alkeita opetellessamme, mutta on hyvä esimerkiksi tietää tuplen olemassaolosta, koska huolimattomasti koodatessa on tavallista, että teemme muuttujasta vahingossa listan sijaan tuplen. Myös sanakirja on hyvä tuntea, koska sitä käytetään tavallisesti luokkarakenteen korvaajana.

Tuple

Tuplet ovat rakenteeltaan samanlaisia kuin listat, joskin niissä on yksi merkittävä ero: ne ovat alkioiltaan kiinteitä. Tämä siis tarkoittaa sitä, että tupleilla ei ole metodeja eikä muitakaan tapoja manipuloida niitä. Eli tuplen sisältö onkin muuntumaton sen jälkeen, kun ne on alustettu. Tupleja käytetään usein juuri silloin, kun halutaan varmistaa, että esimerkiksi funktiolle annettu sarjamainen parametri ei tule muuttumaan missään vaiheessa.

Esimerkki 8.8. Tuplen käyttäminen

```
# -*- coding: utf-8 -*-
# Tiedosto: tuple.py

zoo = ("susi", "elefantti", "pingviini") # huomioi kaarisulut vrt. lista
print("Yhteensä eläimiä on", len(zoo))

uusi_zoo = ("apina", "kirahvi", zoo)
print("Uudessa eläintarhassa on", len(uusi_zoo),"eläintä.")
print("Uuden eläintarhan kaikki eläimet:", uusi_zoo)
print("Vanhasta eläintarhasta tuotuja ovat:", uusi_zoo[2])
print("Uuden eläintarhan viimeinen eläin on", uusi_zoo[2][2])
```

Tuloste

```
>>>
Yhteensä eläimiä on 3
Uudessa eläintarhassa on 3 eläintä.
Uuden eläintarhan kaikki eläimet: ('apina', 'kirahvi', ('susi',
'elefantti', 'pingviini'))
Vanhasta eläintarhasta tuotuja ovat: ('susi', 'elefantti', 'pingviini')
Uuden eläintarhan viimeinen eläin on pingviini
>>>
```

Kuinka se toimii

Muuttuja `zoo` on tuple ja se sisältää 3 alkiota. Kun otetaan funktiolla `len` `zoo`:n pituus, saamme arvon 3, jolloin voimme siis päätellä, että myös tuple on sarjamuuttuja ja sitä voidaan käyttää `for`-lauseessa. Erikoista tässä tehtävässä on kuitenkin vasta muuttujan `uusi_zoo` –määrittely. Kun katsot tarkkaan, huomaat että se sisältää muuttujan `zoo`, joka siis tarkoittaa sitä, että tuple on ottanut tuplen itseensä alkioksi. Tämä on mahdollista myös listojen kanssa ja lisääkin kirjaimellisesti ulottuvuuksia listojen hyödyntämiseen. Alkioksi määritellyn listan yksittäiseen alkioon päästään käsiksi, kun valitaan pääalkion indeksi ja tämän jälkeen jäsenalkion indeksi, eli vaikkapa `uusi_zoo[2][2]`, joka siis on ”pingviini”.

Huomioita tuplesta

Kuten aiemmin on mainittu, ei tuple salli alkioidensa manipulointia eikä se sisällä metodeja:

```
>>> uusi_zoo.append("karhu")

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    uusi_zoo.append("karhu")
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

Lisäksi tuple on harmillisen helppo sotkea määrittelyvaiheessa listan kanssa. Listan määrittelyssä käytetään hakasulkuja [], kun taas tuplen yhteydessä käytetään ’tavallisia’ kaarisulkuja (). Muuten niiden määrittelyn syntaksi on täysin identtinen:

```
>>> lista_zoo = ["karhu", "pingviini", "norsu"] # Tämä on lista
>>> tuple_zoo = ("karhu", "pingviini", "norsu") # Tämä on tuple
```

Sanakirja (eng. Dictionary)

Sanakirja on hieman poikkeava sarjamuuttuja verrattuna kahteen muuhun malliin, listaan ja tupleen. Sen toimintalogiikka muistuttaa enemmän puhelinluetteloa – tai yllättäen sanakirjaa - kuin kauppalistaa. Sanakirja tunnetaan myös monella muulla nimellä kuten hakurakenne, assosiaatiotaulu, luettelo ja englanniksi associative array. Sanakirja sisältää kaksi päätyyppiä, avaimet (key) ja arvot (value). Sanakirjan tapauksessa päätyypeistä avaimen tulee olla ainutlaatuinen, eli sanakirja ei voi sisältää kahta samaa avainta.

Lisäksi avainten rajoitteena on se, että niinä voidaan käyttää ainoastaan vakioarvoisia tietotyyppejä kuten merkkijonoja. Tämä käytännössä tarkoittaa siis sitä, että avaimina voidaan käyttää ainoastaan yksinkertaisia tietotyyppejä.

Sanakirjan määrittely on kuitenkin suhteellisen samanlainen muihin sarjamuuttujiin verrattuna:

```
mallisanakirja = {avain1 : arvo1, avain2 : arvo2 }
```

Huomaa kaksoispiste avainten ja arvojen välissä. Lisäksi tietueet – avain-arvo-parit – erotellaan toisistaan pilkuilla. Lisäksi koko määritelty sanakirja, eli kaikki parit, merkitään aaltosulkeiden {} väliin.

Avaimet eivät ole missään tietyssä järjestyksessä sanakirjan sisällä: jos haluat käyttää jotain tiettyä järjestystä, on sinun huolehdittava siitä jo alustusvaiheessa sekä ylläpidettävä tätä järjestystä käsin.

Esimerkki 8.9. Sanakirjan käyttäminen

```
# -*- coding: utf-8 -*-
# Tiedosto: sanakirja.py

# Sanakirjan indeksi voidaan esittää näin; sulkujen sisällä sitaattien
# välissä välilyönti on merkitsemätön merkki.

sanakirja = {"167-671" : "bigtime@beagle.biz",
             "176-761" : "burger@beagle.biz",
             "716-167" : "bouncer@beagle.biz",
             "176-167" : "babyface@beagle.biz"
            }

print("167-671 sähköposti on", sanakirja["167-671"])

# Lisätään tietue
sanakirja["617-716"] = "baggy@beagle.biz"
# Poistetaan tietue
del sanakirja["176-167"]

print("\nSanakirjassa on", len(sanakirja),"merkintää.\n")

for avain, arvo in sanakirja.items():
      print("Veljeksen {0} sähköposti on {1}".format(avain, arvo))

if "176-761" in sanakirja: # tai sanakirja.has_key("176-761")
      print("\n176-761:n osoite on {0}".format(sanakirja["176-761"]))
```

Tuloste

```
>>>
167-671 sähköposti on bigtime@beagle.biz

Sanakirjassa on 4 merkintää.

Veljeksen 617-716 sähköposti on baggy@beagle.biz
Veljeksen 167-671 sähköposti on bigtime@beagle.biz
Veljeksen 716-167 sähköposti on bouncer@beagle.biz
Veljeksen 176-761 sähköposti on burger@beagle.biz

176-761:n osoite on burger@beagle.biz
>>>
```

Kuinka se toimii

Sanakirja luodaan edellä olevalla syntaksilla. Haemme sanakirjasta avaimella '167-671' sille kuuluvan arvon. Tämän jälkeen lisäämme sanakirjaan uuden avaimen '617-716' ja annamme alustuksessa sille arvon. Seuraavalla rivillä poistamme avaimen '176-167' ja toteamme `len`-funktiolla jälleen kerran, että myös sanakirja on sarjallinen muuttuja. Täten sitäkin voidaan käyttää `for`-lauseen yhteydessä.

Lopuksi `for`-lauseella tulostetaan koko sanakirjan sisältö ja tämän jälkeen tehdään yksinkertainen joukkoonkuuluvuustesti.

Kuinka järjestää sanakirja tai tuple

Listan järjestäminen onnistuu helposti `sort`-jäsenfunktiolla, mutta tuplen ja sanakirjan järjestäminen on hieman työläämpää niiden erilaisesta koostumuksesta. Mutta ei hätää, Pythonista löytyy `sorted`-funktio, joka hoitaa homman molemmille.

Esimerkki 8.10. Tuplen järjesteäminen

```
# -*- coding: utf-8 -*-
# Tiedosto: sorted.py

tuple = (1,10,2,7,12)
print(sorted(tuple))

opiskelijat = (
        ('Matikainen', 'A.', 15),
        ('Asikainen', 'K.', 12),
        ('Viippola', 'S.', 13),
        )

lajiteltu = (sorted(opiskelijat, key=lambda opiskelijat:
opiskelijat[0]))
print(lajiteltu)
print(sorted(opiskelijat, key=lambda opiskelijat: opiskelijat[2],
reverse=True))
```

Tuloste

```
>>>
[1, 2, 7, 10, 12]
[('Asikainen', 'K.', 12), ('Matikainen', 'A.', 15), ('Viippola', 'S.',
13)]
[('Matikainen', 'A.', 15), ('Viippola', 'S.', 13), ('Asikainen', 'K.',
12)]
>>>
```

Kuinka se toimii

Esimerkissä luodaan ensin tuple ja tämän jälkeen tulostetaan se `sorted`-funktion käsitelyn jälkeen. `sorted` toimii siis vähän kuten `len`, eli se palauttaa jotain. Tässä tapauksessa järjestyksessä olevat numerot. Tulee kuitenkin huomata, että `sorted` ei tee muutoksia alkuperäiseen muuttujaan. Jos muutokset halutaan saada lajiteltavaan muuttujaan, täytyy lajittelun tulos tallentaa ”itsensä päälle”

```
tietorakenne = sorted(tietorakenne)
# Vertaa listaan
lista.sort()
```

Esimerkin toisessa osassa luomme käyttöön opiskelijat tuplen, jossa on opiskelijan sukunimi, etunimen ensimmäinen kirjain ja ikä. Seuraavaksi suoritamme lajittelun opiskelijat tuplen alkio ensimmäisen (nollannen!) kentän mukaan. Tulos tallennetaan `lajiteltu`-muuttujaan ja tämän jälkeen se tulostetaan ruudulle. Ohjelman viimeisellä rivillä tulostetaan sama opiskelijat tuple lajiteltuna iän mukaan. Tällä kertaa käänteisesti, eli laskevasti.

Mutta mitä ihmettä tämä `lambda` tarkoittaa? Tämä on Pyhonin tapa ilmaista *anonyymi funktio*, mutta tästä sinun ei tarvitse tietää vielä yhtään mitään. Asiaan palataan myöhemmillä ohjelmointikursseilla. Riittää, että hahmotat kuinka opiskelijat lajitellaan tietyn kentän mukaan.

Huomioita sarjallisten muuttujien vertailusta

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

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

Alkioita keskenään verrattaessa se alkio, jonka ensimmäinen eroava merkki on merkistötaulukon arvoltaan suurempi, on myös lopullisesti suurempi, vaikka alkio olisi lyhyempi. Tämän takia esimerkiksi isot kirjaimet ovat aina pieniä kirjaimia arvollisesti

pienempiä. Alla olevasta taulukosta näet joitain esimerkkejä näiden sääntöjen käytännön tulkinnoista; kaikki vertailut tuottavat tuloksen `True`:

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

Eri tietotyyppien vertailu keskenään ei ole sallittua. Listaa ei siis voida verrata kokonaislukuun, eikä merkkijonoa liukulukuun. Tämä ei kuitenkaan päde numeroarvoihin: kokonaisluku (integer) 3 on samanarvoinen kuin liukuluku (float) 3.00, eli numeroita verrataan keskenään nimenomaisesti numeroarvoina tallennustyypistä huolimatta. Huomaa kuitenkin, että kokonaisluku ja liukuluku voivat olla yhtä suuret ainoastaan, kun liukuluvun desimaaliosa on 0.

Luvun asiat kokoava esimerkki

Esimerkki 8.11. Listailua

```
# -*- coding: utf-8 -*-
# Tiedosto: listailua.py

lista = []

while True:
    luku = int(input("Anna pos. kokonaisluku (neg. lopettaa): "))
    if luku < 0:
        break
    lista.append(luku)

print("Listassa on seuraavat luvut:", lista)
lista.sort()
print("Järjestettynä lista on:", lista)
lista.pop(0)
lista.pop()
print("Ensimmäinen ja viimeinen luku poistettuna:", lista)
```

Luku 9: Kirjastot ja moduulit

Funktioista puhuttaessa näimme, kuinka pystymme käyttämään uudelleen aikaisemmin muualla määriteltyä koodissa. Entäpä, jos haluaisimme käyttää aikaisemmin tekemäämme funktiota jossain täysin toisessa lähdekoodissa? Pythonissa tämä voidaan toteuttaa käyttämällä moduuleja. Moduuli on periaatteessa lähdekooditiedosto, joka sisältää funktioita, joita voidaan tuoda käytettäväksi sisällyttämisen (eng. import) avulla. Olemme itse asiassa jo aiemmin tehneet moduuleja, sillä kaikki Python-ohjelmat, joissa käytetään funktioita, voivat myös toimia toisissa ohjelmissa moduuleina. Lisäksi Pythonin mukana tulee perusfunktioiden lisäksi suuri joukko moduuleja, jotka mahdollistavat erilaiset toiminnot. Usein ohjelmointiympäristön mukana toimitettavista lisätoimintoja tarjoavista moduuleista käytetäänkin nimitystä kirjastomoduuli (erityisesti Python) tai funktiokirjasto (C, C++).

Kuten sanottu, käyttääksemme kirjastomoduuleita – tai itse tekemiämme moduuleja – joudumme ensin sisällyttämään ne koodiin `import`-komennolla. Sisällyttämisen jälkeen moduulin funktioita voidaan käyttää kuten normaaleja lähdekoodin sisäisiä funktioita. Aloitetaan moduuleihin tutustuminen esimerkin pohjalta:

Esimerkki 9.1. math-moduuli

```
# -*- coding: utf-8 -*-
# Tiedosto: math_moduuli.py

import math

print("Tehdään muutama korkeampaa matematiikkaa vaativa operaatio:")
print("e toiseen potenssiin on", math.exp(2))
print("Neliöjuuri 10:stä on", math.sqrt(10))
print("sin(2) radiaaneina on", math.sin(2))
print("Piin likiarvo on", math.pi)
```

Tuloste

```
>>>
Tehdään muutama korkeampaa matematiikkaa vaativa operaatio:
e toiseen on 7.38905609893
Neliöjuuri 10:stä on 3.16227766017
sin(2) radiaaneina on 0.909297426826
Piin likiarvo on 3.14159265359
>>>
```

Kuinka se toimii

Ensimmäiseksi ohjelmassa sisällytämme `math`-moduulin käyttämällä `import`-käskyä. Käytännössä tämä ainoastaan ilmoittaa tulkille, että haluamme käyttää tämän nimistä moduulia. `math`-moduuli sisältää matemaattisia funktiota, kuten nimestä voidaan jo päätellä.

Tulkki metsästää `import`-komennolla annettua kirjastoa (math.py) ensin samasta kansiosta lähdekooditiedoston kanssa ja sen jälkeen tulkille erikseen määritellyistä kansioista. Näitä ovat yleensä kansiot, joissa sijaitsee Pythonin kirjastoja.

Itse ohjelmassa käytämme `math`-kirjaston tarjoamia funktiota `exp`, `sqrt` ja `sin` sekä

vakiota `pi`. Funktioihin ja vakioon viitataan niin, että ensin kirjoitetaan moduulin nimi, tämä jälkeen piste ja viimeisenä itse funktio tai vakio. Teknillisesti tässä työskennellään siis `math`-nimiavaruuden kanssa.

Esikäännetyt pyc-tiedostot

Moduulin sisällyttäminen on Python-tulkille suhteellisen raskas ja aikaa vievä toimenpide, joten Python osaa muutaman tempun, jolla se pystyy nopeuttamaan toimintaansa. Yksi tällainen temppu on luoda esikäännettyjä tiedostoja, jotka tunnistat tiedostopäätteestä pyc. Tämä pyc-tiedosto nopeuttaa Pythonin toimintaa huomattavasti, koska tämänkaltainen tiedosto on jo valmiiksi käännetty tulkille muotoon, jossa se voidaan nopeasti sisällyttää toiseen ohjelmaan. Jos tiedostoa ei ole olemassa, joutuu tulkki suorittamaan ensin esikäännöksen ja tämän jälkeen tulkitsemaan sen osaksi koodia. Periaatteessa pyc-tiedostoista riittää tietää, että ne ovat tulkin luomia tiedostoja ja nopeuttavat ohjelman toimintaa. Niitä ei kuitenkaan voi muokata käsin, eikä niitä voi tulkita kuin konekielen ohjelmina. Niiden poistaminen on sallittua; jos tiedostoa ei ole olemassa, tekee tulkki uuden käännöksen, mikäli alkuperäinen lähdekooditiedosto on saatavilla.

from...import -sisällytyskäsky

Joissain tapauksissa saatamme törmätä tilanteeseen, jossa haluamme saada funktiot ja muuttujat suoraan käytettäviksemme ilman pistenotaatiota. Eli siis siten, että pääsisimme esimerkiksi `math`-kirjaston funktioihin käsiksi ilman `math`-etuliitettä käyttämällä pelkkää funktionnimeä `sin`.

Tämä onnistuu notaatiolla ”`from x import y`”, jossa x korvataan halutun moduulin nimellä ja y funktiolla tai sen muuttujan nimellä, joka halutaan ottaa suoraan käyttöön. Esimerkiksi, jos haluaisimme ottaa math-moduulin osat suoraan käyttöömme, voisimme toteuttaa sen käskyllä ”`from math import *`”, jossa ”*” `import`-osan jälkeen tarkoittaa, että haluamme tuoda kaikki funktiot ja muuttujat käyttöömme. Toisaalta, jos haluaisimme tyytyä pelkkään siniin, niin voisimme tehdä sen käskyllä `from math import sin`

Tämä sisällytystapa sisältää kuitenkin yhden vakavan riskin: ylikirjoittumisvaaran. Jos kaksi moduulia sisältää esimerkiksi samannimisen funktion, ylikirjoittuu aiemmin sisällytetyn moduulin funktio eikä siihen päästä enää käsiksi. Oletetaan, että meillä on kaksi moduulia nimeltään ”nastola” ja ”hattula”, ja molemmilla on funktio nimeltä ”kerronimi”, joka tulostaa moduulin nimen. Komennot

```
import nastola
import hattula
nastola.kerronimi()
hattula.kerronimi()
```

Palauttaisivat tulosteen

```
>>>
"nastola"
"hattula"
>>>
```

Kun taas sisällytystapa

```
from nastola import *
from hattula import *
kerronimi()
kerronimi()
```

Palauttaisivat tulosteen

```
>>>
"hattula"
"hattula"
>>>
```

Tässä tapauksessa emme pääsisi käsiksi `nastola`-moduulin funktioon `kerronimi` millään muulla tavalla kuin sisällyttämällä funktion uudelleen. Yleisesti ottaen kannattaakin pyrkiä välttämään `from...import`-syntaksia, koska se ei useimmiten hyödytä koodia niin paljoa, että sekoittumis- ja ylikirjoitusriskin ottaminen nimiavaruuksien kanssa olisi perusteltua. Tähän sääntöön on kuitenkin olemassa muutama yleisesti hyväksytty poikkeus, kuten esimerkiksi luvussa 13 mainittu käyttöliittymien tekemiseen tarkoitettu Tkinter-moduuli, mutta omien moduulien kanssa työskennellessä `from...import` kannattaa jättää väliin.

Omien moduulien tekeminen ja käyttäminen

Kuten aiemmin mainittiin, on omien moduulien tekeminen helppoa. Jopa niin helppoa, että olet tietämättäsi tehnyt niitä jo aiemminkin. Jokaista Python-lähdekoodia, joka sisältää funktioita voidaan käyttää myös moduulina; ainoa vaatimus oikeastaan onkin tiedostonpääte py, mutta koska jo IDLE suosittaa niiden käyttämistä, ei asia aiheuttane ongelmia. Aloitetaan yksinkertaisella esimerkillä:

Esimerkki 9.2 Oma tiedosto moduulina

```
# -*- coding: utf-8 -*-
# Tiedosto: omamoduli.py

def terve():
    print("Tämä tulostus tulee moduulin munmoduli funktiosta 'terve'.")
    print("Voit käyttää myös muita funktiota.")

def summaa(luku1, luku2):
    tulos = luku1 + luku2
    print("Laskin yhteen luvut", luku1, "ja", luku2)
    return tulos

versio = 1.0
sana = "Runebergintorttu"
```

Yllä oleva lähdekooditiedosto toimii esimerkin moduulina. Se on tallennettu alla olevan ajettavan koodin kanssa samaan kansioon ja se sisältää kaksi funktiota, `terve` ja `summaa`. Lisäksi moduulille on määritelty vakiomuuttujat `versio` ja `sana`, joista ensimmäinen kertoo `munmoduli`-moduulin versionumeron ja toinen sisältää vain satunnaisen merkkijonon. Seuraavaksi teemme ohjelman, jolla kokeilemme moduulin toimintaa:

```
# -*- coding: utf-8 -*-
# Tiedosto: omamoduli_ajo.py

import omamoduli

omamoduli.terve()
eka = 5
toka = 6
yhdessa = omamoduli.summaa(eka,toka)
print("eka ja toka ovat yhteensä", yhdessa)

print("munmodulin versionumero on", omamoduli.versio)
print("Päivän erikoinen taas on:", omamoduli.sana)
```

Tuloste

```
>>>
Tämä tulostus tulee moduulin munmoduli funktiosta 'terve'.
Voit käyttää myös muita funktiota.
Laskin yhteen luvut 5 ja 6
eka ja toka on yhteensä 11
munmodulin versionumero on 1.0
Päivän erikoinen taas on: Runebergintorttu
>>>
```

Kuinka se toimii

Huomaa, että nyt sisällytimme moduulin “omamoduli” `import`-käskyllä, joten joudumme käyttämään pistenotaatioa. Luonnollisesti `from…import` –rakenteella olisimme päässeet siitä eroon, mutta kuten aiemmin mainittiin, olisi se voinut myöhemmin aiheuttaa ongelmia nimiavaruuksien kanssa.

Nyt olemme katsoneet jonkin verran sisällyttämistä, joten voimme siirtyä eteenpäin tutustumaan varsinaisiin kirjastomoduuleihin. Jatkossa tulemme tutustumaan lähes

luvuittain uusiin kirjastomoduuleihin, joten on tärkeää, että tästä luvusta opit ainakin valmiiden moduulien sisällyttämisen lähdekoodiisi.

Kirjastomoduuleja

math-moduuli

Jo esitellyn `math`–moduulin avulla pääsemme käsiksi erilaisiin matemaattisiin apufunktioihin, joilla voimme laskea mm. trigonometrisia arvoja taikka logaritmeja:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757

>>> math.log(1024, 2)
10.0
```

Tarkempi esittely moduulista löytyy mm. Python Software Foundationin Reference Librarysta, jossa läpikäydään muutenkin kaikki kirjastomoduulit sekä niiden sisältämät funktiot ja hyödylliset muuttuja-arvot, kuten esimerkiksi piin arvo `math.pi`.

random-moduuli

Toinen mielenkiintoinen kirjastomoduuli on satunnaislukujen käytön mahdollistava moduuli `random`. Sen avulla voimme ottaa käyttöön mm. satunnaiset valinnat sekä suorittaa arvontoja koneen sisällä:

```
>>> import random

>>> # Satunnainen valinta
>>> random.choice(['apple', 'pear', 'banana'])
'apple'

>>> # Satunnainen joukko
>>> random.sample(xrange(100), 10)
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]

>>> # Satunnainen liukuluku väliltä 0 <= x < 1.
>>> random.random()
0.17970987693706186

>>> # Satunnainen kokonaisluku väliltä a <= x <= b
>>> random.randint(0, 12)
4
```

datetime-moduuli

Python sisältää yksinkertaisen kirjastomoduulin nimeltä `datetime`, jolla voimme noutaa järjestelmästä kellonaikoja sekä päivämääriä. Moduuli tuntee myös englanninkieliset nimet kuukausille ja viikonpäiville, sekä mahdollistaa monipuoliset ja nopeat laskutoimitukset kalenteripäivämäärillä. Osa moduulin toiminnoista ottaa myös huomioon aikavyöhykkeet:

```
>>> # luodaan päiväys
>>> import datetime

>>> # Tämä päivä kokonaisuudessaan, eli aika sekunteja myöten
>>> nyt = datetime.datetime.now()
>>> nyt
datetime.datetime(2010, 4, 21, 12, 43, 25, 987000)

>>> # Tulostetaan päivämäärä ja aika havainnollisemmassa muodossa
>>> nyt.strftime("Tänään on %d.%m.%Y ja kello näyttää olevan %H:%M.")
'Tänään on 21.04.2010 ja kello näyttää olevan 12:43.'

>>> # Päiviä voi vähentää toisistaan kuten lukuja
>>> syntymapaiva = datetime.date(1989, 5, 30)
>>> # Nyt otamme vain päivän käyttöön ilman kellonaikaa
>>> tama_paiva = datetime.date.today()
>>> ika = tama_paiva - syntymapaiva
>>> ika.days
7631
>>>
```

Näissä esimerkeissä on paljon asiaa, mutta tällä kurssilla riittää, että osaat muodostaa ajan nykyisestä päivästä ja jostain jo olleesta sekä tulostaa ajan ruudulle. Määritteet %d, %m, %Y, %H ja %M sekä monet muut löytyvät Pythonin dokumentaatiosta kohdasta ”Basic date and time types”.

time-moduuli

Joskus haluamme tietää, kuinka kauan jonkin ohjelman suorittaminen kestää tai haluamme antaa jonkin tietyn aikarajan sille, kuinka kauan jokin asia saa kestää. `datetime`-moduuli on tehokas päivämäärien ja kellonaikojen kanssa, mutta siitä puuttuu kuitenkin kunnolliset työkalut suoritusaikojen mittaamiseen. Tämä taas voidaan toteuttaa helpolla `time`-moduulilla:

```
>>> import time
>>> time.clock()
3.0730162632401604e-006
>>> time.clock()    # Odotetaan hetki
9.224702580914613
>>> time.clock()    # Odotetaan hetki
19.14744667269164
```

Esimerkki ei sano paljoakaan käyttäjälle siitä, mitä oikeastaan tapahtuu, mutta kun kokeilet asiaa omalla koneella, huomaat miten `clock`-funktio toimii.

Windows-järjestelmässä `clock` käyttää Windowsin rajapinnassa olevaa palvelua ja funktio palauttaakin liukulukuna tiedon siitä, kuinka kauan ensimmäisestä ajetusta `clock`-kutsusta on mennyt aikaa sekunteina. Täten suorittamalla funktiokutsun kahden mittauskutsun välissä saamme tiedon siitä, kuinka kauan funktiolla meni aikaa kutsun lähettämisestä siihen, että ohjelma siirtyi sitä seuraavalle loogiselle riville. Mittauksen tarkkuus on normaalisti mikrosekuntien luokkaa.

Unix-järjestelmissä ajanotto toimii hieman eri tavalla, mutta antaa samankaltaisen vastauksen. Unixissa aikaa ei oteta mistään järjestelmärajapinnan apufunktiosta, vaan se

otetaan suoraan prosessin käyttämältä prosessoriajalta. Määritelmä ”prosessoriaika” on jo itsessäänkin hieman epäselvä ajanottomenetelmä, mutta tätä funktiota voi käyttää laskenta-aikojen optimoimiseen kummalla tahansa alustalla. Funktiota käyttäessä täytyy vain muistaa, että saadut ajat ovat suhteellisia, eikä mitattu tulos ole vertailukelpoinen kuin samassa ajoympäristössä samalla työasemalla.

`time`-moduuli sisältää myös paljon muita toimintoja, jotka eivät kuitenkaan ole tätä kurssia ajatellen kovinkaan relevantteja. Niistä voit lukea tarkemmin Python Software Foundationin referenssikirjastosta kohdasta ”Basic date and time types”.

urllib-moduuli

`urllib` on moduulina varsin erikoinen. Se ei ole pelkästään koneensisäinen moduuli, vaan itse asiassa mahdollistaa Internet-protokollien kautta tiedon hakemisen Internetistä. Tämä tarkoittaa siis sitä, että moduuli osaa hakea verkkoresurssien, kuten www-sivujen tai Internetin uutissyötteiden, tietoja omien funktioidensa avulla. Yksinkertaisimmillaan tämä voidaan toteuttaa seuraavilla komennoilla:

```
import urllib.request
sivu = urllib.request.urlopen("http://www.it.lut.fi")
sisalto = sivu.read() # luetaan sivun sisältö
sivu.close()
tekstina = sisalto.decode("utf-8") # parsitaan merkistö oikeanlaiseksi
print(tekstina)
```

Tämä tulostaa tietotekniikan osaston verkkosivujen etusivun html-lähdekoodin interaktiiviseen ikkunaan. Tuloste olisi hieman pitkähkö, joten tavallisuudesta poiketen jätämme sen tässä näyttämättä ja jokainen saa itse kokeilla, miten nuo rivit toimivat.

Tämä funktio tietenkin tarvitsee oikeuden käyttää verkkoyhteyttä, joka taas saattaa joissain tapauksissa aiheuttaa hälytyksen palomuurissa. Joten, jos näin käy, tiedät varautua asiaan ja sallia uuden yhteyden. Luonnollisesti, jos estämme ohjelmalta pääsyn verkkoon, ei funktiokaan saa mitään haettua.

fractions-moduuli

Pythonin matemaattisen taidot eivät rajoitu pelkästään kokonais- ja desimaalilukujen käsittelyyn, vaan mukaan voidaan ottaa vielä murtoluvutkin. Eksakti matemaattinen numeroiden pyörittely vaatii usein murtolukuja ja tällöin Pythonista voi olla paljon apua:

```
>>> import fractions
>>> x = fractions.Fraction(1, 3)
>>> x
Fraction(1, 3)
>>> x * 4
Fraction(4, 3)
>>> x * 3
Fraction(1, 1)
```

Määrittelemme ensin `x`:n arvoksi murtoluvun 1/3. Sen jälkeen kysymme `x`:n arvoa ja

saamme – yllätys yllätys – vastaukseksi murtoluvun 1/3. `x * 4` on 4/3 ja `x * 3` on 3/3, jonka Python automaattisesti supistaa arvoon 1/1.

Esittelemme lisää kirjastoja tulevissa luvuissa sitä mukaa, kun niitä tarvitsemme. Jos taas haluat välittömästi tutustua uusiin moduuleihin ja katsella, millaisia moduuleja on olemassa, löytyy niistä kuvaukset Python Software Foundationin referenssikirjastosta, jonka lyhyt käyttöohje löytyy liitteestä 1.

Luku 10: Virheenkäsittelyä

Ohjelman suoritus ei aina etene suunnitellusti, vaan jotain virheellistä ja ennalta odottamatonta saattaa tapahtua ohjelmasi ajon aikana. Ajatellaan vaikka tilannetta, jossa yrität lukea tiedostoa, jota ei ole olemassa, tai muuttaa merkkijonoa kokonaisluvuksi. Tässä tilanteessa ohjelmasi aiheuttaa poikkeuksen, joka johtaa useimmiten siihen, että tulkki keskeyttää käynnissä olleen prosessin ja tulostaa virheilmoituksen. Tässä luvussa käymme läpi toimenpiteitä, joilla voimme ottaa kiinni näitä poikkeuksia ja virhetiloja sekä toipua niistä ilman, että tulkki keskeyttää ohjelman suorittamisen virheilmoitukseen.

Virheistä yleisesti

Kaikki me olemme tähän mennessä nähneet virheitä. Jos ei muuten, niin ainakin silloin, kun teemme niitä tarkoituksellisesti osoittaaksemme esimerkin paikkansapitävyyden. Helpoin tapa aiheuttaa virhe onkin esimerkiksi kirjoittaa `print` isolla alkukirjaimella. Tämä aiheuttaa ohjelmassa virheen, joka taas tuottaa tulkilta `NameError`-poikkeuksen.

```
>>> Print("Moi maailma")
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    Print("Moi maailma")
NameError: name 'Print' is not defined
>>> print("Moi maailma")
Moi maailma
```

Tulkki siis tietää mitä, tapahtui ja missä kohdin virhe on, mutta ei osannut tehdä virheelle mitään muuta kuin antaa huomautuksen ja lopettaa. Entä, jos jatkossa rakentaisimmekin tulkille toipumissuunnitelman, joka kertoisi sille mitä tehdään, jos virhe tapahtuu?

try…except-rakenne

`try...except` on olemassa virheiden kiinniottamista ja niistä toipumista varten. Otetaan esimerkki, jossa koetamme lukea käyttäjältä lukuarvon, mutta saammekin merkkijonon:

```
>>> luku = int(input("Anna luku: "))
Anna luku: hyppyrotta
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    luku = int(input("Anna luku: "))
ValueError: invalid literal for int() with base 10: 'hyppyrotta'
>>>
```

Python palauttaa virheen nimeltä `ValueError`, joka käytännössä tarkoittaa sitä, että tulkki yritti muuttaa tekstiä numeroksi, muttei onnistunut tässä. Seuraavaksi kokeilemme ottaa tämän virheen kiinni ja tehdä lähdekoodiin toipumissuunnitelman, jotta ohjelma ei enää kaatuisi virheeseen.

Virheiden kiinniottaminen

Nyt kun tiedämme, että tarvitsemme kiinnioton ja toipumissuunnitelman virheelle nimeltä `ValueError`, voimme rakentaa `try...except` –rakenteen. Käytännössä tämä tapahtuu siten, että laitamme `try`-osioon sen koodin, jonka haluaisimme tavallisesti ajaa, ja `except`-osioon sen koodin, joka ajetaan, mikäli virhe tapahtuu.

Esimerkki 10.1. Virheen käsitteleminen

```
# -*- coding: utf-8 -*-
# Tiedosto: kaato.py

try:
    luku = int(input("Anna luku: "))
    print("Annoit luvun", luku)
except ValueError:
    print("Et antanut kunnollista lukuarvoa.")
```

Tuloste

```
>>>
Anna luku: robottikana
Et antanut kunnollista lukuarvoa.
>>>
Anna luku: 100
Annoit luvun 100
>>>
```

Kuinka se toimii

Kirjoitimme lausekkeen, joka saattaa aiheuttaa virhetilanteen, osioon `try` ja virheen sattuessa ajettavan koodin osioon `except`. `except`–osiolle voidaan määritellä nimenomainen virheluokka – tässä tapauksessa `ValueError` - tai sitten voimme jättää luokan pois, jolloin `except`-osio suoritetaan aina, kun mikä tahansa virhe tapahtuu. Jos esimerkin tapauksessa keskeyttäisimme esimerkin näppäinyhdistelmällä Ctrl-C, joka aiheuttaa näppäimistökeskeytyksen, saisimme edelleen virheilmoituksen:

```
>>>
Anna luku:
Traceback (most recent call last):
  File "C:\Temp\test.py", line 5, in <module>
    luku = int(input("Anna luku: "))
KeyboardInterrupt
>>>
```

Nyt voisimme toimia kahdella tavalla; joko lisäämällä toisen `except`-osion `KeyboardInterrupt`ille tai tekemällä `except`-osiosta yleisen virheiden kiinniotto-osion. Jos lisäämme koodin loppuun rivin

```
except KeyboardInterrupt:
    print("\nKeskeytit ajon.")
```

Saisimme tulostukseksi

```
>>>
Anna luku:

Keskeytit ajon.
>>>
```

Toisaalta taas muuttamalla koodi muotoon

```
# -*- coding: utf-8 -*-
# Tiedosto: kaato2.py

try:
    luku = int(input("Anna luku: "))
    print("Annoit luvun", luku)
except:
    print("Ohjelmassa tapahtui virhe")
```

Saisimme tulostuksen “Ohjelmassa tapahtui virhe” riippumatta siitä, mikä virhe ohjelman ajon aikana tapahtuikaan. Tämä on kuitenkin ohjelman korjattavuuden kannalta ongelmallinen muoto, koska nyt emme tiedä mikä meni vikaan, emmekä silloin osaa lähteä etsimään syytä virheeseen.

Lisäksi muodossa on toinenkin ongelma: käyttäjä ei pysty keskeyttämään ajoa, koska `except` ottaa kiinni myös käyttäjän tuottaman `KeyboardInterrupt`-keskeytyksen sekä `SystemExit`-komennon. Tämä ongelma voidaan korjata käyttämällä muotoa

```
except Exception:
    print("Ohjelmassa tapahtui virhe")
```

Poikkeama `Exception` rajaa ulkopuolelleen `SystemExit`in – eli `sys.exit()`-komennon – ja `KeyboardInterrupt`in. Tämän vuoksi sen käyttäminen on aiheellista aina, kun haluamme saada kiinni kaikki virheet, mutta emme halua estää ohjelman keskeyttämistä käyttäjän toimesta. Kannattaa myös huomata, että mikäli toipumismekanismit ovat kahdelle eri virheluokalle samat, voidaan niiden toiminta yhdistää samaan `except`-osioon. Muoto

```
except (ValueError, TypeError):
    print("Ohjelmassa tapahtui virhe")
```

ottaisi kiinni sekä `ValueError`in että `TypeError`in. Lisäksi, mikäli haluamme ottaa kiinni useita erityyppisiä virheitä erilaisilla toipumismekanismeilla, voidaan `except`-osioita ketjuttaa peräkkäin niin monta kuin katsotaan tarpeelliseksi. Jokaista `except`-ryhmää kohti on oltava ainakin yksi `try`-osio. Lisäksi `try`-osiot eivät voi olla peräkkäin, vaan jokaista `try`-osiota on loogisesti seurattava ainakin yksi `except`-osio. Kannattaa myös pitää mielessä, että `try…except`-rakenteeseen voidaan liittää `else`-osio, joka ajetaan siinä tapauksessa, ettei yhtäkään virhettä synny:

Esimerkki 10.2. else-rakenne

```
# -*- coding: utf-8 -*-
# Tiedosto: kaato4.py

try:
    luku = int(input("Anna luku: "))
    print("Annoit luvun", luku)
except Exception:
    print("Tapahtui virhe.")
else:
    print("Ei huomattu virheitä.")

print("Tämä tulee try-except-else-rakenteen jälkeen.")
```

Tuloste

```
>>>
Anna luku: 42
Annoit luvun 42
Ei huomattu virheitä.
Tämä tulee try-except-else-rakenteen jälkeen.
>>>
```

try…finally

Entä, jos haluamme mahdollisuuden toteuttaa joitain komentoja siinä tapauksessa, että tapahtuu virhe ja ohjelma kaatuu? Tämä voidaan toteuttaa `finally` –osiolla. Jos `except` perustuu ohjelman jatkamiseen ja hallittuun virheestä toipumiseen, perustuu `finally` hallittuun alasajoon ja ohjelman lopettamiseen.

Esimerkki 10.3. Finally-osio

```
# -*- coding: utf-8 -*-
# Tiedosto: finally.py
import sys

import time

tiedosto = open("uutinen.txt", "r", encoding="utf-8")
try:
    while True:
        rivi = tiedosto.readline()
      if len(rivi) == 0:
            break
      time.sleep(2)
      print(rivi, end="")
finally:
    print("Puhdistetaan jäljet, suljetaan tiedosto.")
    tiedosto.close()
    sys.exit(-1)
```

Tuloste

```
>>>
Balin palapelitehdas
pisti pillit pussiin pian
pelipalojen palattua piloille pelattuina:
Puhdistetaan jäljet, suljetaan tiedosto.
Traceback (most recent call last):
  File "C:\Temp\test.py", line 13, in <module>
    time.sleep(2)
KeyboardInterrupt
>>>
```

Kuinka se toimii

Ohjelma aloittaa normaalisti avaamalla tiedoston ja lukemalla sieltä rivejä jo aiemmin kohtaamastamme uutisjutusta. Ohjelma odottaa kaksi sekuntia aina rivin jälkeen, kunnes tulostaa uuden rivin. Nyt, kun keskeytämme ajon KeyboardInterruptilla (Ctrl-C), saamme normaalin tulkin virheilmoituksen.

Huomionarvoista on kuitenkin se, että juuri ennen virheen esiintymistä suoritamme `finally`-osion ja vapautamme käyttämämme tiedoston. `finally`-osio onkin tehty juuri tätä varten - jos tapahtuu virhe, voi ohjelma vielä viimeisinä tehtävinään vapauttaa käyttämänsä tiedostot ja verkkoresurssit, jotta käyttöjärjestelmä voi antaa ne eteenpäin toisille ohjelmille. Tulee myös huomata, että `finally`a, kutsutaan **aina**, vaikka virhettä ei sattuisikaan.

Kuinka tiedostoja tulee käsitellä aikuisten oikeasti?

Nyt, kun virheenkäsittely alkaa jo luonnistua, mieleen tulee, että tiedostoista lukemiset ja niihin kirjoittamiset on todellakin hyvä suorittaa niin, ettei virhe kaada koko ohjelmaa. Toisaalta, jos try...except...finallya täytyy laittaa vähän joka väliin, niin koodin luettavuus kärsii, eikö tähän olisi jotain yksinkertaisempaa vaihtoehtoa?

Ja taas kerran pääsemme toteamaan, että Python tarjoaa mm. tiedostonkäsittelyyn toimivan ratkaisun avainsanaltaan `with`. Katsotaan esimerkki ja ihmetellään sen jälkeen.

Esimerkki 10.4. Turvallista tiedostonkäsittelyä

```
# -*- coding: utf-8 -*-
# Tiedosto: with.py
import sys

lista = []
try:
    with open("luvut.txt", "r", encoding="utf-8") as tiedosto:
        while True:
            rivi = tiedosto.readline()
            if len(rivi) == 0:
                break
            rivi = rivi[0:-1]
            lista.append(int(rivi))

except IOError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(-1)

print("Tiedoston käsittely suoritettu onnistuneesti.")
print("Kiitos ohjelman käytöstä.")
```

Kuinka se toimii

Tässä esimerkissä ei ole muuta tavallisuudesta poikkeavaa kuin `with`-lause ja tiedoston sulkemisen puuttuminen. `with`in kanssa voimme avata minkä tahansa tiedoston johonkin tiedostokahvaan. Kun `with`-lohkon suoritus loppuu, niin tiedostokahva suljetaan – **vaikka olisi sattunut virhe.** `with`in kanssa ei siis tarvitse murehtia mihin väliin tiedoston sulkeminen tulisi sijoittaa ja sen käyttö myös parantaa koodin luettavuutta pykälällä. Edellä mainittu ratkaisu on jopa pykälän parempi kuin seuraavana oleva luvun asiat kokoava esimerkki, joka on kyllä sekin täysin pätevä. Tulee kuitenkin huomata, että läheskään kaikki ohjelmointikielet eivät sisällä `with`in kaltaista rakennetta, joten esimerkki 10.5 on hieman universaalimpi

Luvun asiat kokoava esimerkki

Esimerkki 10.5. Turvallista tiedostonkäsittelyä

```
# -*- coding: utf-8 -*-
# Tiedosto: try_try.py
import sys

try:
    tiedosto_kahva = open("tiedosto.txt", "r", encoding="utf-8")
    try:
        rivi = tiedosto_kahva.readline()
        if len(rivi) == 0:
            print("Tiedosto on tyhjä!")
        else:
            asetus_vipu = rivi[:-1] # Määritetään asetus_vipu
    except IOError:
        print("Tiedostosta lukeminen epäonnistui.")
        sys.exit(-2)
    finally:
        tiedosto_kahva.close() # Suljetaan onnistuneesti avattu tiedosto
except IOError:
    print("Tiedoston avaaminen epäonnistui.")
    sys.exit(-1)

print("Tiedosto-operaatiot suoritettu onnistuneesti.")
print("Kiitos ohjelman käytöstä.")
```

Luku 11: Ongelmasta algoritmiksi, algoritmista koodiksi

Usein ohjelmointimaailmassa on tapana aloittaa ohjelmointi puoliohjelmoinnista, joka ei varsinaisesti ole mikään varsinainen ohjelmointikieli, mutta soveltuu kuvaamaan niistä useimpia. Tällaista kuvausta sanotaan usein pseudokoodiksi. Lisäksi monesti tietynlainen tapa ratkaista ongelma, kuten esimerkiksi alkioiden järjestely tai suurten kokonaislukujen kertolasku, on niin spesifinen, että sen toteutustapaa voidaan kutsua algoritmiksi. Useimmiten nimenomaisesti algoritmit kuvaavat jotain tiettyä tehokkaaksi todettua tapaa ratkaista hyvin määriteltyjä ongelmia, ja niiden esittelyssä kirjallisuudessa käytetäänkin usein ns. pseudokieltä.

Pseudokielet ovat ainutlaatuisia siksi, että niiltä puuttuu kokonaan kielioppi. Ne eivät varsinaisesti ole humaanisia kirjakieliä siinä, missä esimerkiksi englanti, saksa tai suomi saatikka sitten niin teknisiä kuten ohjelmointikielet. Yleisesti pseudokielet ovatkin rakenteeltaan edellisten sekoituksia, ohjelmointikielen kaltaisia esityksiä, joissa tapahtumien kuvaus on normaalisti esitetty kirjakielellä. Lisäksi ne saattavat sisältää jonkinlaisia yksinkertaisia malleja ja pseudorakenteita sekä hyödyntää muuttujien perusrooleja – säiliöitä ja askeltajia – yksinkertaistamaan esityksen kokonaisuutta.

Algoritmeista ja pseudokoodeista löytyy lisätietoja mm. Wikipediasta. Moneen ongelmaan Google tarjoaa myös pseudokielisiä ratkaisuja.

Tarkastellaan esimerkkinä ns. *lisäyslajittelualgoritmia* (eng. "insertion sort") annetun *n* kokonaisluvun lukujonon järjestämiseksi suuruusjärjestykseen. Menetelmän idea on ottaa jokainen luku vuorollaan käsiteltäväksi ja siirtää sitä niin kauas vasemmalle lajitellussa alkuosassa, kunnes vasemmalla puolella on pienempi tai samankokoinen alkio ja oikealla puolella suurempi alkio.

Pseudokoodiesitys lisäyslajittelusta

```
def Lisäyslajittelu(lista t)

   n = listan t alkioiden lukumäärä

   # Käydään taulukon jokainen (paitsi ensimmäinen) alkio läpi

   for i = 2 to n # Siirrytään toisesta alkiosta kohti loppua, alkuosa
                  # pysyy lajiteltuna
       j = i

    # Siirretään alkiota vasemmalle niin kauan kunnes seuraava
    # vasemmalla on siirtyjää pienempi tai samanarvoinen.

         while(t[j] < t[j-1] ja j > 1)

         # Vaihdetaan alkioiden t[j] ja t[j-1] arvot keskenään
         j = j - 1

         end while
   end for
```

Kuinka siis tästä pääsemme eteenpäin? Helpoin tapa lähteä hahmottelemaan vastausta on yksinkertaisesti rakentaa koodista jonkinlainen malli omalla ohjelmointikielellä; onhan pseudokoodissa jo valmiina näytetty järjestys, missä asiat toteutetaan sekä joitain konkreettisia rakenteita, kuten toistorakenteet `for` ja `while`:

```
def lisays
    for
        while
```

Tämän jälkeen voimme hahmotella algoritmiin parametrit; funktiohan selvästi ottaa vastaan ainoastaan yhden parametrin, joka on lajiteltava lista. Lisäksi voimme miettiä, miten esim. `for`-lauseen kierroslukumäärä saadaan oikein, eli käymään toisesta alkiosta viimeiseen asti:

```
def lisays(lista):
    for askel in range(1,len(lista)):
        while
```

Nyt huomaamme, että `for`-lausetta varten tekemämme muuttuja ”askel” on itse asiassa sama kuin pseudokoodin ”i”, joten otetaan sen sisältämä tietue talteen ja tehdään samalla muuttuja `j` ohjeen mukaisesti, jotta voimme käyttää tietueiden indeksejä:

```
def lisays(lista):
    for askel in range(1,len(lista)):
        muisti = lista[askel]
        j = askel
        while
```

Nyt huomaamme, että meiltä puuttuu enää `while`-rakenteen ehdot, joten lisäämme ne vielä loppuun:

```
def lisays(lista):
    for askel in range(1, len(lista)):
        muisti = lista[askel]
        j = askel
        while lista[j - 1] > muisti and j > 0: # Hakee tallennuspaikan
```

Nyt kun funktiomme osaa jo läpikäydä listaa sekä löytää paikan, johon tieto tallennetaan, tarvitsemme enää varsinaiset luku- ja kirjoitusoperaatiot:

```
def lisays(lista):
    for askel in range(1, len(lista)):
        muisti = lista[askel]
        j = askel
        while lista[j - 1] > muisti and j > 0: # Hakee tallennuspaikan
            lista[j] = lista[j - 1]
            j = j - 1
        lista[j] = muisti # Tallentaa sijoitettavan muuttujan arvon.
```

Huomaamme, että olemme luoneet pseudokoodista lisäyslajittelufunktion Pythonilla.

Huomautus

Yleisesti ottaen, mitä aikaisemmin opettelee lukemaan pseudoalgoritmeja, sen helpommin ohjelmointi sujuu myös vaikeita asioita toteutettaessa. Tämä, niin kuin monet muutkin ohjelmoinnin taidot, ovat kuitenkin sellaisia, että sen oppii ainoastaan harjoittelemalla. Pseudokielen luku- ja kirjoitustaito on asia, joka helpottaa lähdekoodin suunnittelua huomattavasti sekä mahdollistaa ei-natiivikielisten esimerkkien hyödyntämisen.

Toinen esimerkki

Ensimmäinen esimerkkimme oli aika suoraviivainen, koska lähtökohtanamme oli valmis pseudokoodi. Toisessa esimerkissä selvitämme, kuinka saamme aikaan parhaan makuista simaa vapuksi.

Ongelma on seuraava. Teija Teekkarilla on käytössään 3 kiloa sokeria, saimaallinen vettä, 1 kilo hiivaa ja 10 litran ämpäri. Siman laatu määräytyy seuraavasti: Sima on laadukkainta, kun hiivaa on mukana $-(h-10)^2+100$, missä h on hiivan prosenttiosuus koko seoksesta. Sokerille vastaava kaava on $-(s-15)^2+100$, missä s on sokerin prosenttiosuus. Lisäksi hiivaa tulee olla kolmasosa sokerin määrästä. Tehtävänä on nyt selvittää, kuinka paljon ämpäriin tulee laittaa hiivaa ja sokeria.

Saamme ongelmasta kaksi matemaattista ehtoa:

$-(h-10)^2+100 - (s-15)^2+100$ = mahdollisimman suuri

$h = 1/3\ s$

Sijoitetaan h s:n paikalle:

$-(h-10)^2+100 - (3*h-15)^2+100$

Nyt tarvitsee enää selvittää, millä h:n arvolla yhtälö antaa suurimman arvon. Tämän voisimme ratkaista derivoimalla, mutta koska olemme huomanneet, että Python on nopea laskemaan, voimme luoda ohjelman, joka ratkaisee ongelman.

Lähdetään ratkomaan tätä `for`-silmukalla. Silmukka lähtee liikenteeseen nollasta, koska negatiiviset tilavuusprosentit ovat vähän vaikeita toteuttaa fysikaalisesti, ja päättyy luonnollisesti sataan.

```
max_laatu = 0
oikea_prosentti
for  i = 0 to 100
    uusi_laatu = laske_laatu
    if uusi_laatu > max_laatu
        max_laatu = uusi_laatu
        oikea_prosentti = i
```

Tarvitsemme ohjelmaamme muuttujat `max_laatu`, joka pitää tallessa parhaimman laadun ja `oikea_prosentti`, jossa on tallessa prosenttiarvo parhaalle laadulle. Enää ei

puutu kuin pseudokoodin kääntäminen Pythoniksi.

```
max_laatu = 0
oikea_prosentti = -1
for i in range(0, 101, 3):
    uusi_laatu = -(i - 10) ** 2 + 100 - (3 * i - 15) ** 2 + 100
    if uusi_laatu > max_laatu:
        max_laatu = uusi_laatu
        oikea_prosentti = i
```

Hiomme vielä ohjelman käyttäjäystävälliseksi, jotta siman tekeminen onnistuu keltä vain.

Esimerkki 11.1. Simalaskuri

```
# -*- coding: utf-8 -*-
# Tiedosto: sima.py

max_laatu = 0
oikea_prosentti = -1
for i in range(0, 101):
    uusi_laatu = -(i - 10) ** 2 + 100 - (3 * i - 15) ** 2 + 100
    if uusi_laatu > max_laatu:
        max_laatu = uusi_laatu
        oikea_prosentti = i

vetta = int(input("Anna veden määrä litroissa: "))
print("Sima tarvitsee", oikea_prosentti / 100 * vetta * 1000, "grammaa \
hiivaa")
print("ja", oikea_prosentti / 100 * 3 * vetta * 1000, "grammaa \
sokeria.")
```

Tuloste

```
>>>
Anna veden määrä litroissa: 10
Sima tarvitsee 500.0 grammaa hiivaa
ja 1500.0 grammaa sokeria.
>>>
```

Huomautettakoon, että hyvän siman tekeminen ei ole näin helppoa. Ohjeestamme puuttuu jo simauutekin aivan täysin, eikä rusinoitakaan liemeen tullut. Haiskahtaa siis hyvin epämääräiselle koko ohje...

Muuttujan rooli: sopivimman säilyttäjä

Simalaskurissamme on kaksi muuttujaa `max_laatu` ja `oikea_prosentti`, joiden arvoa päivitetään aina vastaamaan parhainta löydettyä alkiota. Näiden muuttujien rooli on siis *sopivimman säilyttäjä*.

Luku 12: Tiedon esitysmuodoista

Tietokoneiden kanssa työskennellessä kuulee usein sanottavan, että tietokoneet käsittelevät asioita ainoastaan bitteinä. Tämä väittämä itsessään on tietenkin totta, mutta mitä nämä bitit oikein ovat ja miten niitä käytetään? Tässä kappaleessa tutustumme hieman tarkemmin bitteihin sekä erityisesti merkkitaulukoihin. Lisäksi lopussa esittelemme joitain kehittyneempiä tapoja käsitellä merkkijonoja.

Tietokoneen sisällä kaikki tieto käsitellään ja tallennetaan bittiarvoina, joita ovat arvot 0 ja 1. Jos tutkimme tietokoneen kiintolevyä, johon kaikki koneellemme säilötyt asiat - ohjelmat, pelit, dokumentit – on tallennettu, voisimme oikeilla työkaluilla havaita, että kaikki tieto on tallennettu magneettiselle aineelle jännitteenvaihteluina eli jonona nollia ja ykkösiä. Loogisesti nämä nollat ja ykköset on tallennettu 8 bitin (eli kahdeksan nollan tai ykkösen, esim 10001001) jonoihin, jota sanomme tavuksi. Yksi tavu taas on 2-kantainen esitysmuoto numeroarvolle väliltä 0-255. Seuraavan esimerkin avulla voimme tutkia, kuinka bittiarvoja voidaan laskea:

Esimerkki 12.1. Bittilukujen laskeminen, binaariluvusta kokonaisluvuksi

```
# -*- coding: utf-8 -*-
# Tiedosto: bin2int.py

def bittiluku():
    bittijono = input("Anna binääriluku: ")

    tulos = 0
    pituus = len(bittijono)
    bittijono = bittijono[::-1] # Bittijonoa luetaan lopusta alkuunpäin

    print("Bittijonosi on", pituus, "bittiä pitkä.")

    for i in range(0,pituus):
        if bittijono[i] == "1":   # Jos bitin arvo 1 eli otetaan mukaan
            tulos = tulos + 2**i  # lisätään tulokseen 2^i

    print("Bittijonosi on kokonaislukuna", tulos)

bittiluku()
```

Tuloste

```
>>>
Anna binääriluku: 1101
Bittijonosi on 4 bittiä pitkä.
Bittijonosi on kokonaislukuna 13
>>>
```

Kuinka se toimii

Ohjelma pyytää käyttäjää syöttämään bittijonon (eli vapaamuotoisen jonon nollia ja ykkösiä) ja tämän jälkeen käy läpi jonon laskien siitä samalla kymmenlukuarvon. Koska bittiarvoissa merkitsevin (lukuarvoltaan suurin) bitti tulee vasemmanpuoleisimmaksi, käydään bittijono läpi oikealta vasemmalle.

Bittijonossa lukuarvot edustavat kahden potensseja. Jos bittijono on esimerkiksi 5 bittiä

pitkä, on siinä silloin bitteinä ilmaistuna lukuarvot 2^4, 2^3, 2^2, 2^1 ja 2^0. Nämä toisen potenssit vastaavat numeroarvoja 16, 8, 4, 2 ja 1. Käytännössä biteillä ilmaistaan, lasketaanko kyseinen bitti mukaan vai ei: jos bitti saa arvon 1, lasketaan sitä vastaava toisen potenssi lukuarvoon, jos taas arvon 0, lukua ei lasketa mukaan. Jos tarkastelemme esimerkiksi binaarilukua 1101, laskettaisiin se seuraavasti:

Bittiluku	**1**	**1**	**0**	**1**
2:n potenssi	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Tulos	**1***8 + **1***4 + **0***2 + **1***1 = 8 + 4 + 1 = **13**			

Jos vastaavasti laskisimme bittiarvon esitykselle 101101, saisimme seuraavanlaisen tuloksen:

Bittiluku	**1**	**0**	**1**	**1**	**0**	**1**
2:n potenssi	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Tulos	**1***32 + **0***16 + **1***8 + **1***4 + **0***2 + **1***1 = 32 + 8 + 4 + 1 = **35**					

Toisaalta, voimme myös laskea bittiesityksen kokonaisluvuille seuraavanlaisella ohjelmalla:

Esimerkki 12.2. Bittilukujen laskeminen, kokonaisluvusta binaariluvuksi

```
# -*- coding: utf-8 -*-
# Tiedosto: int2bin.py

def laske_binaari(luku):
    potenssi = 0
    while True:  # Laskee kuinka monta bittiä esitykseen tarvitaan
        if 2**potenssi <= luku:
            potenssi = potenssi + 1
        else:
            break
    jono = ""

    while True:
        if luku - 2**potenssi < 0: # Jos arvo liian suuri, merkitään 0
            jono = jono + "0"
        else:
            jono = jono + "1"  # Bittiarvo voidaan vähentää, merkataan 1
            luku = luku – 2 ** potenssi
        potenssi = potenssi - 1  # Lähestytään arvoa 0 joka kierroksella
        if potenssi == -1:   # Ollaan tultu luvun loppuun
            break
    return jono

lukuarvo = int(input("Anna kokonaisluku: "))
tulos = laske_binaari(lukuarvo)
print("Antamasi kokonaisluku on binaariluvuilla esitettynä", tulos)
```

Tuloste

```
>>>
Anna kokonaisluku: 74
Antamasi kokonaisluku on binaariluvuilla esitettynä 01001010
>>>
```

Kuinka se toimii

Ohjelma pyytää käyttäjää syöttämään kokonaisluvun ja ohjelma laskee siitä bittiesityksen. Ensin ohjelma selvittää kahden n:nnen potenssin, joka on suurempi kuin annettu kokonaisluku. Tämän jälkeen ohjelma koittaa n kierroksella vähentää luvusta kahden senhetkisen potenssin $2^{\text{n-käytyjä kierroksia}}$. Jos vähennys on mahdollista (erotus > 0), lasketaan erotus ja merkitään bittijonoon 1. Jos taas ei, siirrytään pienempään potenssiin ja merkitään bittijonoon 0.

Merkkitaulukot

Kuinka bitit sitten vaikuttavat siihen, mitä tietokoneen kiintolevyltä luetaan? Koska kiintolevylle voidaan fyysisesti tallentaa ainoastaan bittijonoja, joudutaan niitä silloin myös käyttämään kirjainten ja muiden numeroiden tallentamiseen.

Puhuimme aiemmin, että kiintolevylle bittijonot tallennetaan kahdeksan bitin joukkoina, joita sanomme tavuiksi. Näillä kahdeksalla bitillä voimme kuvata numeroarvoja nollasta 255:een. Kun päätämme, että jokainen näistä arvoista ilmaisee yhtä nimenomaista merkkiä, ja kokoamme näistä merkeistä taulukon, olemme luoneet aakkoset bittiesityksillä. Jos ajattelemme esimerkiksi normaalia vuosikymmeniä vanhaa ASCII-taulukkoa, voimme havainnollistaa menetelmää käytännössä.

ASCII	Numero	ASCII	Numero	ASCII	Numero
A	65	a	97	0	48
B	66	b	98	1	49
C	67	c	99	2	50
D	68	d	100	3	51
E	69	e	101	4	52
F	70	f	102	5	53
G	71	g	103	6	54
H	72	h	104	7	55
I	73	i	105	8	56
J	74	j	106	9	57
K	75	k	107	. (piste)	46
L	76	l	108	=	61
M	77	m	109	:	58
N	78	n	110	;	59
O	79	o	111	(välilyönti)	32
P	80	p	112	(rivinvaihto)	10
Q	81	q	113	(	40
R	82	r	114	)	41
S	83	s	115	[	91
T	84	t	116	]	93
U	85	u	117	{	123
V	86	v	118	}	124
W	87	w	119	/	47
X	88	x	120	\	91
Y	89	y	121	+	43
Z	90	z	122	-	45

Yläpuolella olevaan taulukkoon on kerätty ASCII-taulukosta kirjaimia ja numeroarvoja, sekä joitain erikoismerkkejä vastaavat numeroarvot. Jos luemme levyltä bittisarjan 001000001, tarkoittaa se kokonaislukuina arvoa 65. Jos taas tulkitsemme tämän numeroarvon ASCII-taulukon avulla, voimme havaita että luimme tiedostosta ison A-kirjaimen. Jos tiedostossa vastaavasti lukisi ”01011100 01110101 01010100 01101111”, voitaisi se lukea arvoina ”97 117 84 111”, eli ”auTo”.

Meitä ajatellen ASCII-taulukossa on kuitenkin yksi suuri ongelma; se ei tunne skandinaavisia merkkejä. Alkuperäinen ASCII-taulukko suunniteltiin nimenomaisesti englanninkieliselle aakkostolle, joten alkuperäinen ratkaisu ei sisältänyt Ä, Ö eikä Å -kirjaimia. Tämän vuoksi olemme myöhemmin joutuneet laajentamaan olemassa olevia merkkitaulukkoja sekä luomaan joukon uusia. Lisäksi käyttöjärjestelmiin on toteutettu mahdollisuus valita mitä merkkitaulukkoa ohjelmien lukemisessa ja tulkitsemisessa käytetään.

Modernissa tietotekniikassa siirrytään kohti universaaleja merkkitaulukoita, joista mainittakoon ASCII-yhteensopiva UTF-8. Tässä merkkitaulukossa on käytössä yli miljoona merkkiä. Tällä merkkimäärällä voidaan latinalaisten, kyrillisten ja arabialaisten aakkosten lisäksi kuvata niin kiinan kuin japanin tai koreankin sanamerkit, jolloin tarve erillisille merkkitaulukoille poistuu pysyvästi. UTF-8 on käytettävissä kaikissa moderneissa käyttöjärjestelmissä, mukaan luettuna Windows-perheen uusimmat yksilöt. Python-ohjelmointikielessä UTF-8-tuki myös luonnollisesti löytyy ja se on Python 3:n oletustapa käsitellä merkkejä.

Windows XP ja Pythonin versiot ennen versiota 3 tuottivat välillä ongelmia UTF-8-merkistökoodauksen kanssa ja tästä syytä oli usein parempi käyttää merkkitaulukkoa 1252.

```
# -*- coding: cp1252 -*-
```

Käytettävän merkkitaulukon merkkejä voi testailla Python-tulkissa funktioilla `ord` ja `chr`, joista ensimmäinen palauttaa annetun merkin järjestysnumeron merkkitaulukossa, ja jälkimmäinen taas tulostaa annettua järjestysnumeroa vastaavan merkin:

```
>>> ord("a")
97
>>> chr(97)
'a'
>>> ord("€")
8364
>>> chr(54353)
'퍑'
>>>
```

Eli nykypäivänä yksi tavu ei enää vastaa yhtä merkkiä, vaan merkki voi koostua 1-4 tavusta. Python-ohjelmoijan ei tästä tarvitse sen kummemmin välittää, koska Python hoitaa kaikki tarvittavat muunnokset automaattisesti. Käyttämällä UTF-8 merkistökoodausta saavutetaan se, että kuka tahansa pystyy käsittelemään koodia ja näkee tulostukset varmasti oikein, mikäli käytössä vain on käyttöjärjestelmä tältä vuosituhannelta.

Pickle-moduulista

Joskus voimme tarvita mahdollisuutta tallentaa tietoa muutenkin kuin pelkästään merkkijonoina. Tätä varten Python-ohjelmointikieleen on luotu pickle-niminen moduuli, jolla voimme tallentaa binääritietona esimerkiksi kokonaisia luokkia, listoja sekä sanakirjoja. Pickle-moduulin käyttö on suoraviivaista ja perustuu kahteen funktioon, dump sekä load:

Esimerkki 12.2. Pickle-moduuli, binääritallennus

```
# -*- coding: utf-8 -*-
# Tiedosto: p_dump.py

import pickle
import sys

try:
    with open("testi.data","wb") as tiedosto:
        lista = ["Turtana","Viikinkilaiva",{"Joe-poika":"Papukaija"},
327000884764897.5]

        pickle.dump(lista,tiedosto)
except IOError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(-1)
```

Koska pickle-moduuli tallentaa tiedon binäärimuotoisena, näyttää käyttämämme tiedoston testi.dat sisältö hyvinkin sotkuiselta (riippuen erilaisista tekijöistä, tiedoston sisältö saattaa näyttää erilaiselle):

```
(lp0
S'Turtana'
p1
aS'Viikinkilaiva'
p2
a(dp3
S'Joe-poika'
p4
S'Papukaija'
p5
saF327000884764897.5
a.
```

Älä muokkaa tätä tiedostoa käsin, koska et pysty tekemään muuta kuin rikkomaan sen. Huomaa myös, että nyt emme määritelleet merkistökoodausta ollenkaan, kun avasimme tiedoston. Tähän on looginen selitys: emme kirjoittaneet tiedostoon merkkejä vaan tavuja, joten merkistökoodausta ei tarvita. Tiedoston sisältö ei ole ihmisen luettavissa, eikä sen ole tarkoituskaan olla.

Esimerkki 12.3. Pickle-moduuli, tallenteen lataaminen

```
# -*- coding: utf-8 -*-
# Tiedosto: p_dump.py

import pickle
import sys

try:
    with open("testi.data","rb") as tiedosto:
        luettu = pickle.load(tiedosto)
except IOError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(-1)

print(luettu,"\n", luettu[1], luettu[2])
```

Tuloste

```
>>>
['Turtana', 'Viikinkilaiva', {'Joe-poika': 'Papukaija'},
327000884764897.5]
Viikinkilaiva {'Joe-poika': 'Papukaija'}
>>>
```

Kuten näemme, voidaan pickle-moduulilla tallentaa ja ladata kehittyneempiä tietorakenteita ilman että niitä joudutaan muuttamaan ensin merkkijonoiksi. Pickle-moduuli osaakin käsitellä kaikkia tässä oppaassa esiteltyjä tietomuotoja. Huomioinarvoista onkin muistaa se, että pickleä käyttäessämme joudumme avaaman tiedoston binääritilaan avauskytkimillä ”rb” (read-binary) ja ”wb” (write-binary).

Pari sanaa pyöristämisestä

Olet ehkä tähän mennessä jo huomannutkin, etteivät tietokoneen käsittele numeroita yhtä tarkasti kuin ihmiset. Myös pyöristämisessä käytetään eri menetelmiä kuin mitä suomalaisessa peruskoulussa opetetaan.

Koska tietokoneen muistiin ei mahdu numeroita äärettömällä tarkkuudella tulee desimaaliosia sisältävillä numeroilla laskennassa joskus vastaan ”hassuja” tilanteita:

```
>>> 3 * 5.2
15.600000000000001
>>>
```

Tämä johtuu siitä, että liukuluvut on määritelty tietyillä standardeilla ja niitä käsitellään sitä kautta tiettyjen sääntöjen mukaan. Tästä voi lueskella esimerkiksi Wikipediasta hakusanalla ”liukuluku”. Englanninkielinen artikkeli on laajempi.

Toinen mielenkiintoinen asia on pyöristäminen. Me suomalaiset olemme tottuneet siihen, että vitonen pyöristyy ylöspäin. Tämä tuo mukaansa ongelmia. Ajatellaanpa seuraavaa: Kahvilassa on myynnissä pulla hintaan 2,5€ ja kakkupala hintaan 3,5€. Erittäin vanha kirjanpitojärjestelmä ei osaa kuitenkaan laskea kuin kokonaisluvuilla, joten se pyöristää aina numerot kokonaisluvuksi. Tällöin kun saadaan myytyä 10 pullaa ja 10 kakkupalaa on kirjapidossa tulona 70€, mutta kassassa on vain 60€.

Pythonissa tämä ongelma on ratkaistu pyöristämällä vitonen aina parillista numeroa kohti:

```
>>> round(2.5)
2
>>> round(3.5)
4
>>>
```

Tällöin pyöristysvirhe ei pääse kumuloitumaan. Kuulostaako tällainen pyöristäminen hassulta? Se voi sellaiselta kuulostaa suomalaiseen korvaan, mutta tulee huomata, että maailmassa on peruskoululaitoksia, jossa tällaista pyöristystä opetetaan ensimmäisestä luokasta lähtien, joten heille meidän suomalaisten käyttämä pyöristys tuntuu varmasti hyvin epäloogiselle. Tietokoneiden käsitellessä numeroita parilliseen numeroon suuntaava pyöristys on kätevä vaihtoehto, koska tällöin pyöristyksestä johtuva virhe minimoituu. Pyöristystavoistakin löytyy lisää tietoa ainakin Wikipediasta englanniksi hakusanalla ”rounding”.

Luku 13: Graafisten käyttöliittymien alkeet

Toisin kuin vielä esimerkiksi 20 vuotta sitten, nykyisin useimmat ohjelmat julkaistaan käyttöjärjestelmille, joissa on mahdollisuus käyttää graafista käyttöliittymää. Tämän vuoksi käytännössä kaikki laajemmat ohjelmat, ohjelmistot sekä työkalut toimivat nimenomaan graafisella käyttöliittymällä, jossa valinnat ja vaihtoehdot esitetään valikoina tai valintoina, joiden avulla käyttäjä voi hiirellä tai kosketusnäytöllä valita mitä haluaa tehdä. Monesti aloitteleva ohjelmoija kuitenkin luulee, että tällaisen käyttöliittymän toteuttamista varten tarvitaan jokin monimutkainen tai kallis kehitystyökalu tai että se olisi erityisen vaikeaa. Ehkä joidenkin ohjelmointikielien yhteydessä tämä pitää paikkansa, mutta Pythonissa yksinkertaisten graafisten käyttöliittymien tekeminen onnistuu helposti Tkinter-kirjastomoduulin avulla.

Tkinter on alun perin Tcl-nimisen ohjelmointikielen käyttöliittymätyökalusta Tk tehty Python-laajennus, jonka avulla voimme luoda graafisen käyttöliittymän ohjelmallemme. Tkinter-moduuli poikkeaa siinä mielessä ”perinteisistä” Python-moduuleista, että sitä käyttävä Python-koodi poikkeaa hyvin paljon tavallisesta Python-koodista ulkonäkönsä puolesta. Kuitenkin Python-kielen syntaksisäännöt sekä rakenne pysyvät edelleen samoina. Koodi voidaan edelleen tuottaa yksinkertaisesti tekstieditorilla, josta tulkki tuottaa graafisen esityksen. Kannattaa kuitenkin huomata, että laajempia rakenteita sisältävä Tkinter-ohjelmakoodi on käytännössä aina pitkä ja monesti myös melko työläs kirjoitettava, joten koodin ajamista suoraan tulkin ikkunassa ei kannata yrittää. Käytettäessä Tkinter-moduulia ainoa käytännössä järkevä lähestymistapa ongelmaan on luoda lähdekooditiedosto, joka tallennetaan ja ajetaan sellaisenaan tulkista.

Seuraavilla kahdella esimerkillä tutustumme siihen, kuinka yksinkertaisen graafisen ikkunan tekeminen onnistuu. Tämä tietenkin tarkoittaa, että ensimmäiseksi tarvitsemme pohjan, johon käyttöliittymän rakennamme.

Graafinen käyttöliittymä

Esimerkki 13.1. Perusikkuna

```
# -*- coding: utf-8 -*-
# Tiedosto: gui.py

from tkinter import *

pohja = Tk()
pohja.mainloop()
```

Esimerkkikoodin tuottama tulos

Kun ajamme esimerkin koodin, tuottaa tulkki seuraavanlaisen käyttöliittymäikkunan:

Kuva 1: Esimerkin 13.1. tuottama käyttöliittymäikkuna.

Kuinka koodi toimii

Ensimmäinen esimerkkikoodi on varsin lyhyt eikä vielä sisällä paljoakaan toimintoja. Ensimmäisellä rivillä meillä on aikaisemmistakin esimerkeistä tuttu näppäimistökartan määrittely. Toisella rivillä otamme käyttöön Tkinter-kirjaston. Tällä kertaa kannattaa huomata, että sisällytys toteutetaan hieman tavallisuudesta poikkeavalla from...import * -syntaksilla johtuen siitä, että sen käyttäminen on tämän kirjaston yhteydessä näppärämpää.

Kolmannella rivillä luomme käyttöliittymän perustan tekemällä muuttujasta `pohja` Tk()-luokan juuri- eli pohjatason (root). Tähän pohjatasoon lisäämme jatkossa kaikki haluamamme komponentit ja toiminnot. Tällä kertaa kuitenkin riittää, että jätämme sen tyhjäksi. Neljännellä rivillä käynnistämme Tkinter-käyttöliittymän kutsumalla pohjatasoa sen käynnistysfunktiolla mainloop. Ohjelma lähtee käyntiin ja tuottaa tyhjän ikkunan. Tämä tapahtuu sen vuoksi, että ohjelman pohjatasolle ei ole sijoitettu mitään muuta komponenttia. Jos olisimme laittaneet sille vaikka painonapin, olisi ruutuun ilmestynyt pelkkä painonappi.

Komponenttien lisääminen

Käyttöliittymästä ei ole paljoa iloa, jos siinä ei ole mitään muuta kuin tyhjiä ikkunoita. Tämän vuoksi haluammekin lisätä ruutuun komponentteja (widgets), joiden avulla voimme lisätä ruutuun tarvitsemamme toiminnot. Komponentit lisätään aina joko suoraan pohjatasolle tai vaihtoehtoisesti frame-komponenttiin, joka toimii säiliönä ja tilanjakajana Tkinter-käyttöliittymässä. Tarkastelkaamme ensin kuinka pohjatasolle lisätään tekstikenttä.

Esimerkki 13.2. Tekstikenttä perusikkunassa

```
# -*- coding: utf-8 -*-
# Tiedosto: tk-gui.py

from tkinter import *
```

```
pohja = Tk()

tekstikentta = Label(pohja, text="Moi maailma!")
tekstikentta.pack()

pohja.mainloop()
```

Esimerkkikoodin tuottama tulos

Kun ajamme esimerkin koodin, tuottaa tulkki seuraavanlaisen käyttöliittymäikkunan:

Kuva 2: Esimerkin 13-2 ikkuna.

Kuinka koodi toimii

Tässä lähdekoodissa tuotamme ensimmäisen käyttöliittymän, joka tekee jotain. Alkuun suoritamme samanlaiset alustustoimenpiteet kuten aiemmin. Määrittelemme näppäimistökartan, otamme käyttöön Tkinter-moduulin sekä luomme pohjatason muuttujaan `pohja`. Tästä eteenpäin määrittelemme ensimmäisen komponenttimme.

Tällä kertaa luomme komponentin nimeltä `Label`, jota käytetään tavallisimmin staattisen tekstin esittämiseen. Tämä sopii meille varsin hyvin, koska haluamme ainoastaan saada ohjelman tuottamaan tekstiä aiemmin tekemäämme ikkunaan. Tässä tapauksessa luomme komponentin muuttujaan `tekstikentta` ja annamme sille määrittelyssä seuraavat parametrit:

1. Ensimmäinen parametri `pohja` tarkoittaa sitä, että luomme komponentin pohjatason päälle. Tämä tarkoittaa sitä, että käynnistäessämme ohjelman, ilmestyy luomamme tekstikenttä `tekstikentta` pohjatason määrittelemän alueen sisään. Tähän annetaan tavallisesti parametrina se ikkunan osa tai alue, johon napin halutaan ilmestyvän, mutta koska meillä ei ole käytössä olevaa ikkunointiasettelua, niin komponentti voidaan laittaa suoraan pohjalle.

2. Toinen parametri määrittelee yksinkertaisesti sen, mitä tekstikentässä on tarkoitus lukea. Luonnollisesti parametriksi voi antaa myös muuttujannimen, mutta ilman lisätoimenpiteitä teksti luetaan muuttujasta ainoastaan alustuksen yhteydessä.

Seuraavaksi paketoimme kyseisen komponentin. Tällä käskyllä ilmoitamme tulkille, että emme anna kyseiselle komponentille enempää alustusmääritteitä ja että komponentin voi ”pakata” käyttöliittymään. Pakkauksen yhteydessä voimme vielä antaa joitain sijoittelua koskevia käskyjä, mutta tällä kertaa niitä ei tarvita. Muista, että **ainoastaan pakattu komponentti näkyy käyttöliittymän ikkunassa**. Jos komponenttia ei pakata, ei tulkki piirrä sitä ruudulle ja käyttöliittymäsi toimii väärin. Lopuksi laitamme ohjelman käyntiin kutsumalla pohjatason `mainloop`-funktiota.

Huomioita Tkinter-moduulista

Koska Tkinter-moduuli on hyvin laaja ja toimintatavoiltaan jonkin verran tavallisesta Python-ohjelmakoodista poikkeava, ei tässä oppaassa aihetta käsitellä tämän enempää. Lisää graafisten käyttöliittymien toteuttamisesta Tkinter-moduulilla voit lukea tämän oppaan jatko-osasta ”Python – Tkinter ja graafinen käyttöliittymä”. Opas löytyy mm. verkosta osoitteesta

http://wiki.python.org/moin/FinnishLanguage

Loppusanat

Olemme tämän oppaan turvin tutustuneet 13 luvussa Python-ohjelmoinnin alkeisiin sekä joihinkin ohjelmistokehityksen perusajatuksiin. Tähän pisteeseen päästyämme voimme jo alkaa puhumaan ohjelmistotuotannosta sekä varsinaisesta ohjelmoinnista tavoitteenamme tehdä toimivia, niin sanotusti ”oikeita” ohjelmia sen sijaan, että rakentelemme esimerkkejä olemassa olevien rakenteiden päälle. Tästä eteenpäin loogisia jatkoaiheita ovatkin laitteistoläheisemmät ohjelmointikielet sekä toisaalta graafisten käyttöliittymien toteuttaminen.

Python on siitä hyvä ohjelmointikieli, että voit halutessasi jatkaa vielä kauan kielen kanssa harjoittelua. Se, että ymmärsit perusasiat, on jo merkki siitä, että saat halutessasi tehtyä paljon muitakin asioita kuin mitä tässä oppaassa käsiteltiin. Vaikka et tietäisi tarvitsevasi ohjelmointitaitoja jatkossa, sinun ei kannata täysin unohtaa nyt oppimiasi asioita: Nykytietotekniikalla ohjelmointi on aihe, joka tulee vastaan hyvinkin yllättävissä paikoissa, kuten Excel-taulukoiden tai interaktiivisten PowerPoint-esitysten yhteydessä. Lisäksi, mikäli olet tietotekniikan opiskelija, on sinun hyvä opetella Pythonia, koska se on vahvassa kasvussa oleva nuori kieli, jonka käyttäjäkunnasta löytyy jo tässä vaiheessa mm. Nokia ja NASA.

Lisäluettavaa

Tästä oppaasta eteenpäin jatkavalle on olemassa useita vaihtoehtoja. Jos haluat tutustua tarkemmin esimerkiksi kuvien muokkaamiseen ja piirtämiseen Python-ohjelmointikielellä, on oppaalle julkaistu jatko-osa ”Python - Graafinen ohjelmointi Imaging Librarylla”. Jos taas haluat opetella tekemään Windows-ohjelmien kaltaisia graafisia käyttöliittymiä, kannattaa sinun tutustua oppaaseen ”Python - Tkinter ja graafinen käyttöliittymä”. Molemmat oppaat ovat suomenkielisiä ja ilmaisia.

Kaikki yllämainitut oppaat, sekä muutama liitteeksi tai lisäluettavaksi tarkoitettu miniopas on saatavilla Python Software Foundationin verkkokirjastosta osoitteesta http://wiki.python.org/moin/FinnishLanguage.

Lisäksi kannattaa muistaa, että verkosta on saatavilla myös paljon englanninkielistä materiaalia Python-ohjelmointiin liittyen. Verkon suurinta linkkikirjastoa Python-materiaaliin ylläpitää PSF:n wiki-kirjasto osoitteessa http://wiki.python.org/moin/. Myös http://diveintopython3.org/ tarjoaa paljon opastusta Pythonin maailmaan. Eikä koskaan kannata unohtaa Pythonin omia – laajoja – dokumentaatiota, jotka löytyvät osoitteesta http://docs.python.org/py3k.

Oppaassa käsiteltiin muutama eri muuttujan roolia. Lisää luettavaa Jorma Sajaniemen sivuilta osoitteesta http://cs.joensuu.fi/~saja/var_roles/.

Lähdeluettelo

Kasurininen, Jussi, 2008. Python – ohjelmointiopas, versio 1.2

Pilgrim, Mark, 2010. Dive Into Python 3

http://diveintopython3.org/

Python-materiaalia suomen kielellä.

http://wiki.python.org/moin/FinnishLanguage

Python Software Foundation, 2010. Python v3.x documentation

http://docs.python.org/py3k

Sajaniemen, Jorma, 2008. The Roles of Variables.

Lähde: http://cs.joensuu.fi/~saja/var_roles/

The Python Wiki

http://wiki.python.org/moin/

Liite 1: Lyhyt ohje referenssikirjastoon

Tässä liitteessä tutustumme lyhyesti Python-dokumenttien keskeiseen osion, Python Software Foundationin referenssikirjaston. Kyseinen kirjasto sisältää kaiken tarpeellisen tiedon Python-ympäristöstä, sen toiminnoista, funktioista, moduuleista sekä operaattoreista.

Allaolevaan taulukkoon on listattu ne luvut, joista luultavimmin löytyy lisätietoa tämän oppaan läpikäymistä asioista. Mikäli haluat tutustua luvun sisältöön, mene Internetissä osoitteeseen http://docs.python.org/py3k/library/index.html (Pythonin versiolle 3) ja etsi vastaava luku. Verkossa olevat dokumentit ovat englanninkielisiä.

Luku	**Sisältö**
2	Sisältää tietoa Python-ympäristön sisäänrakennetuista funktioista ja käskyistä kuten input(), print(), import sekä len()
5.2	Tietoa Boolean-arvoista sekä loogisista väittämistä (and, or, not).
5.3	Tietoa vertailuoperaattoreista (<,>, !=)
5.4	Tietoa numeerisista tyypeistä (int, float jne.) sekä operaatioista joissa niitä voi käyttää (+,-,*, int() jne.)
5.6	Tietoa merkkijonoista ja sarjamuuttujista, lisäksi käsitellään operaatioita joita niillä voidaan tehdä sekä leikkauksista. Mukana myös merkkijonojen metodit.
7.1	Tietoa merkkijonojen muotoilusta
5.9	Sisältää tietoa erilaisista tiedostokahvojen käsittelyyn käytettävistä funktiosta, esim. fseek(), readline() jne.
27.1	sys-kirjastomoduulin esittely
7.2	Merkkijonojen merkkisarjoja
9.2	math-kirjastomoduulin esittely
9.6	random-kirjastomoduulin esittely
15.1	os-kirjastomoduulin esittely
8.1	datetime-kirjastomoduulin esittely
15.3	time-kirjastomoduulin esittely
20.5	urllib.request-kirjastomoduulin esittely
24.1	Tkinter-kirjastomoduulin esittely
24.6	Lyhyt ohje IDLE-ohjelmointiympäristöön

Liite 2: Yleinen Python-sanasto

Tämä liite sisältää lyhyen sanakirjan sanoista, joita käytetään yleisesti ohjelmoinnin, Python-ympäristön tai ohjelmointilähtöisen tietotekniikan parissa.

Sana tai termi	**Selite**
ajoympäristö	Ajoympäristö on tietokoneen sisäisten osien, käyttöjärjestelmän ja tulkin yhdessä muodostama kokonaisuus, jossa ohjelma ajetaan.
alkio	Alkio on sarjallisen muuttujan yksi osamuuttuja. Lista ja tuple muodostuvat alkioista. Jos yhdessä osamuuttujassa on useita alkioita, puhutaan tällöin tietueesta. (kts. tietue).
argumentti	Argumentti on parametrien saama varsinainen arvo.
ASCII-taulukko (laajennettu -)	ASCII-taulukko on tietokonemerkistö, joka sisältää englanninkielen aakkoset, välimerkit sekä joitain ohjausmerkkejä. Merkistön tärkein etu on siinä, että merkistössä jokaista kirjainta ja merkkiä edustaa bittiarvo, jolla merkki voidaan tallentaa tietokoneen muistiin. Laajennettu ASCII-taulukko sisältää myös skandinaaviset merkit sekä muita erikoisaakkosia.
askeltaja (rooli)	Askeltaja on muuttuja, joka käy läpi arvoja systemaattisesti. Esimerkiksi toistorakenteen kierroslukulaskuri on malliesimerkki askeltajasta.
automaattinen muistinhallinta	Kts. dynaaminen muistinhallinta.
debuggeri	Debuggeri on ohjelma, jolla voidaan kontrolloida ohjelman ajonaikaista toimintaa, valvoa tulkin etenemistä sekä tarkastella ohjelmansisäisien muuttujien arvoja. Debuggeria käytetään virheiden löytämiseen ja poistamiseen.
dynaaminen muistinhallinta	Dynaamisella muistinhallinnalla tarkoitetaan järjestelmää, joka suorittaa automaattisesti tietokoneen keskusmuistin varaamisen ja vapauttamisen ohjelman tarpeiden mukaisesti. Vastakohta manuaalinen muistinhallinta
editori	Kts. koodieditori
ehtolauseke	Ehtolauseke on looginen väittämä, joka liitetään if- ja elif-rakenteisiin. Jos ehtolauseke on tosi, suoritetaan se osio, johon lause oli liitettynä. Muussa tapauksessa suoritetaan rakenteen else-osio tai sen puuttuessa rakenne ohitetaan kokonaan.
ehtorakenne	Ehtorakenne on rakenne, jossa suoritettavan koodiosion valinta perustuu siihen, täyttyykö ehtolauseke vai ei. Pythonissa ehtolausekkeena on if-elif-else-rakenne.
funktio	Funktio on erikseen kutsuttava koodista muodostettu looginen kokonaisuus, eli sisennetty koodilohko, joka voi itsenäisesti suorittaa sille annettuja tehtäviä.
funktiokirjasto	Kts. kirjastomoduuli

jäsenfunktio	Jäsenfunktio on funktio, jota kutsutaan pistenotaation avulla ja joka on osa jonkin tietyn kokonaisuuden toimintaa; esimerkiksi tiedostokahvan jäsenfunktiot. Jäsenfunktio on myös toiminnallinen osa luokka-tietorakennetta jolla voidaan esimerkiksi muokata jäsenmuuttujien arvoja tai suorittaa luokkaan liittyviä toiminnallisuuksia. Jäsenfunktion toinen nimi on metodi.
jäsenmuuttuja	Jäsenmuuttuja on luokkarakenteeseen määritelty muuttuja, johon viitataan pistenotaation avulla.
kehitysympäristö	Kehitysympäristö on joukko ohjelmia, jotka yhdessä muodostavat kokonaisuuden, jolla voidaan luoda, ajaa sekä testata jonkin tietyn ohjelmointikielen koodia. Yleisimmät osat ovat editori, tulkki/kääntäjä sekä debuggeri.
kiintoarvo (rooli)	Kiintoarvo on muuttuja, jossa muuttujalle määritellään yksi arvo, joka säilyy sillä koko ohjelman suorituksen ajan. Termiä käytetään joskus myös loogisen väittämän numeroarvoista.
kirjanmerkki	Kirjanmerkki on tiedostonkäsittelyssä käytettävä arvo, joka säilyttää kohdan, jossa tiedostoa ollaan lukemassa tai johon seuraavaksi kirjoitetaan. Siirtyy automaattisesti luku- ja kirjoitusfunktioiden mukana.
kirjastomoduuli	Kirjastomoduulilla tarkoitetaan sellaista moduulia, joka on toimitettu asennuspaketin mukana. Esimerkiksi *sys* tai *random.*
komentokehote /komentorivi	Komentokehote on ikkuna, johon käyttäjä syöttää tekstimuotoisia käskyjä joilla ohjataan tietokonetta.
kommentti	Kommentti on lähdekoodiin lisätty merkintä, jota tulkki tai kääntäjä ei huomioi ja joka on ensisijaisesti tarkoitettu koodaajan omiksi muistiinpanoiksi. Kommenttilauseita saatetaan kuitenkin joissain tilanteissa käyttää mm. käyttöympäristöä koskevien meta-tietojen antamiseen.
koodieditori	Tekstinkäsittelyohjelma, joka on suunniteltu ohjelmointia varten. Perusominaisuuksiin yleisesti kuuluu mm. käskysanojen ja rakenteiden korostaminen sekä automaattinen sisennyksien hallinta.
Koodilohko (osio)	Pythonissa koodilohkolla tarkoitetaan rakenteensisäistä palaa koodia, joka on merkitty yhdeksi kokonaisuudeksi sisennystason avulla. Esimerkiksi while-rakenne on oma koodilohkonsa.
käsky (-lause)	Käsky tarkoittaa yhtä loogista riviä koodia, jolla suoritetaan jokin operaatio, kuten tulostus, sijoitus tai vertailu.
kääntäjä	Kääntäjä on ohjelma, jolla voidaan kääntää lähdekooditiedosto konekieliseen muotoon ajamista varten. Kääntäjä lukee koko lähdekoodin ennen kuin tuottaa funktionaalisen ohjelman.
luokka	Luokka tarkoittaa rakenteista tietotyyppiä, johon voidaan käsin määritellä jäsenmuuttujat sekä jäsenfunktiot.
lähdekoodi	Tiedosto tai joukko tiedostoja, johon on tallennettu varsinaiset koodirivit yhdestä tehtävästä tai ohjelmasta.
manuaalinen muistinhallinta	Manuaalinen muistinhallinta on muistinhallintatapa, jossa ohjelmoija joutuu itse laskemaan, varaamaan ja vapauttamaan keskusmuistista tarvitsemansa alueet.

meta-tieto	Metatieto on tietoa koskevaa tietoa. Tällä siis tarkoitetaan, että esimerkiksi Python-koodin metatietoa voisi olla vaikka tieto kooditaulukosta, jolla koodi on kirjoitettu, tieto siitä, millä kielellä lähdekoodin kommentit ja tulostukset ovat jne.
metodi	Kts. jäsenfunktio
moduuli	Python-kielessä moduulilla tarkoitetaan lähdekoodiin ulkopuolelta sisällytettyä tiedostoa, joka sisältää funktioita sekä vakioita.
muuttuja	Muuttuja on "säiliö", johon voidaan tallentaa käyttäjän haluamaa tietoa ja muunnella sitä. Muuttujilla voi käyttötavoista riippuen olla erilaisia rooleja, jotka kuvaavat sen käyttötarkoitusta ja sisällön tyyppiä.
operaattori	Operaattori on merkki tai joukko merkkejä, jolla annetaan tulkille ohje siitä, mitä annetuille operandeille tullaan tekemään. Lauseessa "2 + 3" '+' on operaattori, koska se antaa tulkille ohjeen laskea arvot (operandit) '2' ja '3' yhteen.
operandi	Operandi on arvo tai muuttuja, joka on operaattorin suorittaman toimenpiteen lähtöarvo. Lauseessa "2 + 3" '2' ja '3' ovat operandeja, koska operaattori '+' laskee ne yhteen.
palautusarvo	Palautusarvo on se arvo, jonka funktio lähettää sitä kutsuneelle koodiosalle suoritettuaan toimintonsa loppuun.
parametri	Parametri on funktiokutsussa määritelty muuttuja, joka annetaan kutsuttavalle funktiolle ohjaustietona.
rakenne	Rakenne on looginen koodikokonaisuus, jonka muodostuu toisiinsa liittyvistä osioista. Esimerkiksi try-except-rakenne sisältää try- ja except-osiot.
roolit (muuttujan-)	Muuttujan rooleilla tarkoitetaan erilaisia käyttötapoja, joihin muuttujia käytetään. Yleisesti rooleja on 11 erilaista, joista kiintoarvo, askeltaja ja tuoreimman säilyttäjä kattavat noin 70 % aloittelijoiden tarpeista.
sarjallinen muuttuja	Sarjallinen muuttuja on muuttuja, joka sisältää alkioita tai tietueita ja jota for-lause voi muokata suoraan tietueesta seuraavaan siirtymällä. Pythonissa on kolme erilaista sarjallista muuttujaa: lista, tuple sekä sanakirja. Sarjallisen muuttujan englanninkielinen termi on ”*Sequence*”.
semantiikka	Semantiikka tarkoittaa kielen loogisuutta. Tämä eroaa syntaksista sillä, että semantiikka koskee kielen tarkoittamaa asiaa, ei sen rakenteellista oikeellisuutta. Esimerkiksi lause "Laiska ajatus myy sinisiä filosofeja autolle." on syntaksisesti aivan oikein mutta semanttisesti täysin mieletön.
shell-ikkuna	Linux-puolen vastaava termi Windows-puolen komentorivikehotteelle. Kts. Komentorivi.
sisäinen funktio	Funktio, jonka käyttämistä varten ei erikseen tarvitse tehdä funktiomäärittelyä tai antaa sisällytyskäskyä. Esimerkiksi len() tai print().
sopivimman säilyttäjä (rooli)	Sopivimman säilyttäjä on muuttuja, johon on tallennettu paras tai soveltuvin toistaiseksi löydetty arvo.
syntaksi	Syntaksi tarkoittaa kielioppia. Erityisesti tietojenkäsittelytieteissä syntaksilla tarkoitetaan varattuja sanoja sekä käskylauserakenteita. Esimerkiksi lause ”Python kieli ohjelmointi on.” on sisällöllisesti oikein, mutta syntaktisesti väärin. Syntaktisesti oikein kirjoitettuna lause on ”Python on ohjelmointikieli.”

säiliö (rooli)	Säiliö on tietorakenne, johon voidaan tallentaa ja josta voidaan poistaa tietoa tarpeen mukaisesti.
tiedostopääte	Tunniste, jolla käyttöjärjestelmä ja tekstieditori tunnistaa tiedoston tyypin, eli sen mitä tiedosto pitää sisällään ja millä ohjelmalla tiedoston sisältöä olisi tarkoitus käsitellä. Python-koodin tiedostopääte on py.
tietue	Tietue on useista tietoalkioista muodostuva kokonaisuus. Esimerkiksi sanakirjan osamuuttujat ovat tietueita, jotka sisältävät kaksi alkioa, avaimen ja arvon.
tilapäissäilö (rooli)	Tilapäissäilö on muuttuja, johon tallennetaan lyhytaikaista säilytystä varten jokin tietty arvo. Verrattavissa laskimen muisti-toimintoon.
toistoehto (lauseke)	Ehto, jonka totuusarvo tarkastetaan aina toistorakenteen uuden kierroksen alkaessa. Jos ehto on Tosi, suoritetaan kierros, jos taas Epätosi, lopetetaan toistorakenne.
toistorakenne	Toistorakenne on ohjelman rakenne, jota toistetaan kunnes jokin haluttu toistoehto saavutetaan. Pythonissa toistorakenteita ovat while- ja for-rakenteet
tulkki	Tulkki on ohjelma, jolla voidaan suorittaa lähdekooditiedostoja. Tulkki lukee lähdekooditiedostoja sitä mukaa, kun niitä ajonaikana tarvitaan.
tuoreimman säilyttäjä (rooli)	Tuoreimman säilyttäjä on muuttuja, johon tallennetaan viimeisin sisään otettu tai luettu arvo.
ulkoinen funktio	Ulkoinen funktio on funktio, joka on sisällytyskäskyllä otettu käyttöön ulkoisesta tiedostosta.
ulkoinen tiedosto	Ulkoinen tiedosto tarkoittaa mitä tahansa tiedostoa, joka ei ole varsinainen lähdekooditiedosto.

Liite 3: Tulkin virheilmoitusten tulkinta

Tässä liitteessä esittelemme joitakin yleisimpiä Python-tulkin virheilmoituksia sekä arvioita siitä, mitä luultavasti on tapahtunut tai mitä virheen poistamiseksi voidaan tehdä.

Virheilmoitus	**Mitä tarkoittaa**	**Mitä luultavasti tapahtui**
AttributeError	Muuttujalla tai funktiolla ei ole pyydetyn nimistä metodia tai arvoa.	Koetit manipuloida tuplea listan metodeilla tai kirjoitit jäsenfunktion nimen väärin. Toinen vaihtoehto on että yritit käyttää pistenotaatiota paikassa, jossa sitä ei voi käyttää.
EOFError	input() ei saanut luettavaa arvoa.	Lopetit ohjelman suorituksen CTRL-D-yhdistelmällä komentokehotteessa.
IOError	kirjoitus- tai tulostusoperaatio epäonnistui.	Koetit lukea tiedostoa, jota ei ole olemassa tai koitit kirjoittaa lukumoodilla tai levy, jolle kirjoitit, täyttyi.
ImportError	import-käskyn suoritus ei onnistunut.	Moduuli, jonka koetit sisällyttää, oli kirjoitettu väärin tai tallennettu paikkaan josta tulkki ei sitä löytänyt.
IndexError	Annettu sijainti ylitti jonon tai listan pituuden.	Yritit lukea merkkiä tai alkiota, joka on merkkijonon tai listan ulkopuolella.
KeyError	Pyydettyä avainta ei löydy.	Koetit lukea sanakirjasta tietuetta, jonka avainta ei löytynyt.
KeyBoardInterrupt	Tapahtui näppäimistökeskeytys.	Keskeytit tulkin ajon näppäinyhdistelmällä CTRL-C (tai vastaava).
NameError	Annettu nimi on virheellinen.	Koitit joko alustaa muuttujaa tai funktiota epäkelvolla nimellä tai kirjoitit nimen väärin.
RuntimeError	Tapahtui yleinen virhe.	Python-tulkki ei osannut määritellä millainen virhe tapahtui, mutta jotain meni vikaan.
SyntaxError	Koodin syktaksissa on virhe.	Olet luultavasti sisentänyt jonkin rivin väärin tai koodistasi puuttuu pilkkuja tai sitaattimerkkejä.
SystemError	Tulkin sisäisissä tiedostoissa tapahtui virhe.	Python-tulkin asennus on mennyt jostain syystä rikki. Aja korjausasennus asennuspaketista.

SystemExit	Ohjelma lopetti toimintansa.	Lopetit ohjelman funktiolla sys.exit().
TypeError	Tietotyypeissä on yhteensopivuusongelma	Koetit muuttaa merkkijonon numeroarvoksi tai tehdä laskutoimituksen merkkijonolla tai kirjoittaa tiedostoon ei-merkkijonoarvon.
UnboundLocalError	Viittasit muuttujaan, jolla ei ole arvoa.	Koetit käyttää tunnettua muuttujaa, jolle ei ole määritelty arvoa paikassa, jossa muuttujalla on oltava yksiselitteinen arvo.
Unicode … Error (useampia)	unicode-merkkirivin käsittelyssä tapahtui virhe.	Koetit käyttää unicode-riviä, jota ei saatu käännettyä tai joka sisälsi virheitä.
ValueError	Arvon määrittelyssä tapahtui virhe.	Koitit antaa operaattorille tai funktiolle arvon, joka on oikeantyyppinen mutta sopimaton.
ZeroDivisionError	Ohjelmassa tapahtui nollalla jako.	Yritit jakaa luvun nollalla tai muuttujalla, jonka arvo oli 0. Tapahtuu myös jakojäännös- ja tasajako-operaattorien kanssa.

Lisäksi on olemassa vielä joitakin virhetiloja, jotka kuitenkin ovat niin erikoisluonteisia tai epätodennäköisiä, että niiden läpikäyminen tässä oppaassa on epäoleellista. Täydellinen lista virheilmoituksista löytyy Python Software Foundationin kirjastosta osoitteesta www.python.org.

Varoituksista

Joissain tapauksissa tulkki saattaa myös antaa varoituksia. Näitä kaikkia yhdistää se, että niiden nimestä löytyy muodossa tai toisessa sana ”**Warning**”. Nämä tapahtumat eivät ole vielä varsinaisesti virheitä, mutta ovat muotoja tai tiloja, jotka voivat aiheuttaa jatkossa ongelmia. Esimerkiksi **SyntaxWarning** tarkoittaa sitä, että annetun koodin syntaksi ei täytä kaikkia semanttisia asetuksia. Hyvän ohjelmointitavan mukaista on korjata koodia siten, että näistä varoituksista päästään eroon.

Liite 4: Tyyliopas

Tämä tyyliopas on tarkoitettu Lappeenrannan teknillisen yliopiston Ohjelmoinnin perusteet -kurssin tueksi. Tyylioppaan tarkoitus on orientoida opiskelija käyttämään oikeita rakenteita oikeissa tarkoituksissa ja oikealla tyylillä kirjoitettuna.

Opas perustuu Jussi Kasurisen tyylioppaaseen, joka on vapaasti suomennettu ja koottu alkuperäisestä Python Enhancement Proposalista numero 8. Kyseinen PEP sisältää Guido van Rossumin ja Barry Warsawin tyylioppaista kootun kokonaisuuden, jolla määritellään kielen virallinen tyyliopas.

Guido van Rossumin alkuperäinen ajatus ohjelmoinnista on, että ihmiset käyttävät enemmän aikaa koodin tulkitsemiseen kuin sen kirjoittamiseen. Tämän vuoksi Python-ohjelmoinnissa ja – ohjelmointityylissä pyritään aina suosimaan kielen ymmärrettävyyttä sekä pitämään lähdekoodin kieliasu yhtenäisenä. Kannattaa kuitenkin pitää mielessä, että joskus on olemassa tilanteita, joihin tietyt tyylisäännöt tai tyylioppaat eivät vain yksinkertaisesti päde. Tällöin tulee pyrkiä pitämään linja yhtenäisenä ja huolehtia Python-koodin tärkeimmästä ominaisuudesta – sen ymmärrettävyydestä ja selkeydestä.

Tyyliohjeet

Sisennyksestä

Käytä sisennyksissä koodin tasolta toiselle aina neljää (4) välilyöntiä per taso. Ainoan poikkeuksen tähän tekee vanhojen lähdekoodien, jotka käyttävät aiemmin voimassa ollutta 8 välilyönnin standardia, ylläpito ja korjaus.

Älä koskaan käytä yhtäaikaisesti sisennysmerkkejä (tabulaattorimerkki) ja välilyöntejä. Mikäli et ole varma, tallentaako käyttämäsi editoriohjelma sisennysmerkit automaattisesti neljänä välilyöntinä, älä käytä molempia vaan ainoastaan toista merkkiä sisennyksen tekemiseen.

Rivien pituudesta ja monirivisistä lauseista

Pidä lähdekoodin yhden koodirivin pituus maksimissaan 80 merkkiä pitkänä.

Edelleen on olemassa laitteita, joiden näyttökyky rajoittuu 80 merkkiin per rivi; lisäksi noudattamalla tätä sääntöä on helpompaa tarkastella kahta koodia yhtä aikaa yhdeltä näytöltä. Huomioi myös, että mikäli kirjoitat funktioillesi dokumentaatiorivejä, käytä niissä rivin pituutena 72 merkkiä.

Käytä rivien katkaisemiseen Python-tulkin kenoviivaa (’\’). Jos välttämättä tarvitset, voit käyttää myös sulkeita, mutta useimmiten kenoviiva näyttää paremmalta. Lisäksi katkaisun jälkeinen rivi on sisennettävä oikealle tasolleen joko alkuperäisen rivin tasalle tai sulkeiden alkamistasolle.

Tyhjistä riveistä

Erottele koodissa olevat ylätason funktiot ja luokkamääritelmät toisistaan tyhjällä rivillä. Luokan sisällä olevat metodit tulee erotella toisistaan yhdellä rivillä. Sisällytyskäskyryhmät tulee erotella toisistaan tyhjällä rivillä.

Tyhjiä rivejä voidaan lisätä, mikäli tarkoituksena on erotella koodista toisiinsa liittyvät funktiot. Funktion sisällä tyhjiä rivejä voidaan käyttää erottelemaan koodin loogiset osat toisistaan.

Sisällyttämisestä

Kokonaisten moduulien sisällyttäminen tulee aina toteuttaa erillisillä riveillä:

```
import os
import sys
```

Mikäli kuitenkin sisällytät ainoastaan osia moduulista, voidaan käyttää merkintätapaa

```
from kirjasto import toiminto1, toiminto2
```

Sisällytyskäskyt tulee aina sijoittaa lähdekoodin alkuun. Ainoastaan koodisivun valinta ja lähdekoodin alkukommentit tulevat ennen sisällyttämiskäskyjä. Lisäksi sisällyttämisessä tulisi käyttää seuraavaa järjestystä:

1. Standardikirjastosta sisällytettävät moduulit (sys, os, time …)

2. Lisämoduuleista sisällytettävät moduulit (py2exe, image, numpy …)

3. Paikalliset lähdekooditiedostot (omat ulkopuoliset lähdekooditiedostot)

Kokonaisen moduulin sisällyttämisessä tulee aina pyrkiä käyttämään täydellä nimellä sisällyttämistä käskyllä import X. Mikäli sisällytät ainoastaan yksittäisiä funktioita, voit käyttää notaatiota from X import Y. Tähän tekee kuitenkin poikkeuksen Tkinter, jonka yhteydessä from Tkinter import * -notaatio on luonteva valinta.

Välilyönneistä

Välilyöntien käyttäminen koodin luettavuuden parantamiseksi on hyvä käytäntö, mutta siihen liittyy muutamia sääntöjä, joilla haluttua vaikutusta voidaan tehostaa.

Älä käytä välilyöntiä seuraavissa kohdissa:

Säännön jälkeen oleva esimerkki näyttää **oikean** tavan tehdä asia.

- Välittömästi kaarisulkeiden, hakasulkeiden tai aaltosulkeiden perään.

```
leipa(voita[1], {juustoa: 2})
```

- Ennen pilkkua, kaksoispistettä tai puolipistettä:

```
if x == 4:
    print(x, y)
```

- Ennen sulkeita, jotka aloittavat funktiokutsun parametrilistan:

```
kinkku(1)
```

- Ennen sulkeita, jotka määrittelevät leikkauksen tai alkioviittauksen:

```
dict['avain'] = list[index]
```

- Enemmän kuin yksi välilyönti viittausten tai sijoitusten ympärillä:

```
x = 1
y = 2
pitka_nimi = 3
```

Käytä välilyöntiä seuraavissa kohdissa:

- Erottele seuraavat merkit ja vertausoperaatiot aina molemmin puolin välilyönneillä:

```
sijoitus (=), arvoa muuttava sijoitus (+=, -= jne.),
vertailut (==, <, >, !=, <=, >=, in, not in, is, is not),
Boolean-arvot (and, or, not).
```

- Erottele numeroarvot ja muuttajat lasku- ja sijoitusoperaattoreista:

```
i = i + 1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Ainoa poikkeus tähän on parametrien avainsanat, joihin välilyöntejä **ei** tarvitse laittaa:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

- Kommentoinnissa #-merkin jälkeen

```
# Tämä on kommentti.
```

Kommentoinnista

Harhaanjohtavat kommentit ovat suurempi ongelma kuin puutteellinen kommentointi. Huolehdi siitä, että kommentit ovat aina ajan tasalla.

Kommenttien tulisi olla kokonaisia lauseita. Kirjoita kommenttisi siten, että ne alkavat isolla alkukirjaimella ja päättyvät pisteeseen. Ainoan poikkeuksen tähän tekee se, jos kommentti alkaa muuttujan tai funktion nimellä, joka koodissa kirjoitetaan pienellä; tällöin myös kommentti alkaa pienellä kirjaimella.

Jos kommentti on muutaman sanan pituinen, voidaan lauserakenteesta tinkiä. Jos kirjoitat pitkän kommenttitekstin, kirjoita teksti kieliopillisesti oikein. Pyri välttämään lyhenteitä. Kirjoita ennemmin liian yksityiskohtaiset kuin liian ylimalkaiset kommentit.

Tällä kurssilla käytämme suomen kieltä ja ohjelmakommentit kannattaa kirjoittaa suomeksi. Muutoin kirjoita kommentointi aina englanniksi, mikäli on pienikin todennäköisyys, että koodisi voi päätyä julkisesti saataville ja mikäli sinua ei ole erikseen ohjeistettu tekemään toisin.

Koodilohkon kommentointi

Jos kirjoitat koko koodilohkoa koskevaa kommenttitekstiä, sisennetään se samalle tasolle, millä koodiosio on. Monirivisessä kommentoinnissa käytä menetelmää, jossa ’#’-merkin jälkeen tulee ainakin yksi välilyönti. Kappaleenvaihto monirivisessä kommentissa merkitään tyhjällä kommenttirivillä, jolla on ainoastaan ”#”-merkki.

Koodinsisäinen kommentointi

Koodinsisäinen kommentointi tarkoittaa kommentteja, jotka kirjoitetaan koodirivin perään. Koodinsisäinen kommentti erotetaan koodirivistä vähintään kahdella välilyönnillä. Lisäksi koodinsisäiset kommentit häiritsevät koodin lukemista, joten niitä ei tulisi käyttää kuvaamaan itsestään selviä asioita. Esimerkiksi kommentti

```
x = x + 1  # x kasvaa yhdellä
```

on turha ja haittaa koodin luettavuutta. Sen sijaan kommentti

```
x = x + 1  # kasvatetaan listan ylärajaa yhdellä
```

on tietyissä tapauksissa hyödyllistä tietoa.

Nimeämiskäytännöistä

Vaikka alkuperäinen nimeämiskäytäntö Pythonin moduulikirjastoissa on välillä hieman

sekava, on silti tarkoituksena jatkossa pyrkiä nimeämään funktioiden ja muuttujien nimet siten, että ne noudattavat tiettyä kaavaa.

Nimeämislogiikasta

Käytä muuttujien, funktioiden ja rakenteiden nimeämiseen jotain seuraavista tavoista:

- `muuttujannimi (pelkkiä pieniä kirjaimia)`
- `muuttujan_nimi_viivalla_erotettuna`
- `IsotKirjaimet (muuttujan nimen sanat alkavat isolla kirjaimella)`

  Huomioi myös tämä: Jos käytät tätä nimeämislogiikkaa, niin käytä isoja kirjaimia myös lyhenteissä: HTTPServerError on parempi kuin HttpServerError.

- `pieniAlkuKirjain (sama kuin yllä, mutta alkaa pienellä kirjaimella)`

Vältettäviä nimiä

Pyri välttämään seuraavia merkkejä muuttujannimissä:

```
-’l’: pieni L
-’O’: iso o
-’I’, iso i
```

Nämä merkit voidaan joillain fonttityypeillä helposti sekoittaa numeroarvoihin 1 ja 0. Kokonaisina muuttujanniminä olisi suositeltavampaa olla kokonaan käyttämättä kyseisiä kirjaimia (esimerkiksi pientä L-kirjainta toistorakenteen askeltajana).

Moduulien, funktioiden ja muuttujien nimistä

Moduuleilla tulisi olla lyhyt, kokonaan pienillä kirjaimilla kirjoitettu nimi. Myös alaviivaa voidaan käyttää, mikäli sillä voidaan parantaa nimen luettavuutta. Koska tulkki hakee moduuleja niiden tiedostonnimistä, olisi moduulin nimen syytä olla erittäin lyhyt ja yksinkertainen. Mikäli oletetaan, että koodia voidaan käyttää vanhoissa Mac- tai DOS-koneissa, on nimen hyvä olla maksimissaan 7 merkkiä pitkä. Samoin alaviivan käyttöä kannattaa näissä tapauksissa välttää.

Funktioiden nimien tulisi olla kirjoitettu kokonaan pienillä kirjaimilla. Lisäksi niiden kanssa voi käyttää alaviivaa, mikäli se parantaa nimen ymmärrettävyyttä. (”laske”,”_tarkasta”)

Mikäli haluamasi muuttujannimi on järjestelmän varattu sana (esimerkiksi ”except” tai ”print”), on parempi menetelmä jättää muuttujannimestä kirjain pois (”xcept”, ”prnt”) kuin lisätä alaviiva (”except_”,”_print”). Tietenkin paras tapa on olla käyttämättä varattujen sanojen kaltaisia muuttujannimiä.

Normaalien muuttujien nimien tulisi olla kuvaavia, sekä mahdollisuuksien mukaan lyhyitä ja ytimekkäitä (luku1, luku2, syote, laskuri, askel …). Pyri välttämään lyhenteiden käyttöä muuttujan nimissä. **Älä käytä mitään tarkoittamattomia merkkijonoja (tlst, kmnt_x, asefw4, blaa1, blaa2 …) muuttujien niminä.**

Tietorakenteiden käytöstä

Python tarjoaa ohjelmoijan käyttöön useita valmiita tietorakenteita (lista, sanakirja, tuple, luokka). Tällä peruskurssilla keskitymme lähinnä lista- ja sanakirja-rakenteiden käyttöön.

Huomioita

Tässä on lyhyesti kuvattuna Pythonin tärkeimmät tyyliohjeet. Mikäli haluat lukea lisää ammattimaisen ohjelmoinnin tyylisäännöistä, voit tutustua laajempaan suomenkieliseen Python-tyylioppaaseen, joka löytyy Python Software Foundationin sivuilta osoitteesta http://wiki.python.org/moin/FinnishLanguage.

Lisäksi voit tutustua myös alkuperäiseen Python-tyylioppaaseen, joka löytyy osoitteesta http://www.python.org/dev/peps/pep-0008/. Tämä opas on englanninkielinen.

Liite 5: Esimerkkirakenteita

Tiedon tulostaminen ruudulle

```
# Normaali tekstin tulostus
print("Tekstiä.")
print('Lisää "tekstiä"')
print("Rivi 1\nRivi 2")
# Määritetään muuttujat ja tulostetaan niiden sisältö ruudulle
nopeus = 13
pii = 3.14159
kaupunki = "Lappeenranta"
print(nopeus + "km/h.")
print("Piin likiarvo on", pii)
# Muotoiltu tulostus, eli asetamme tekstin sekaan numeroita ja tekstiä
# ennalta määrättyjen muotoilusääntöjen mukaan
print("Suunta {0:.2f} astetta, nopeus {1:.3f} m/s.".format(pii,
float(nopeus)))
print("Sieltä löytyy {0:}.".format(kaupunki))
# Merkkijonon ja muuttujalista tulostuksen erot
print(kaupunki + " on kaupunki.")
print(kaupunki, "on kaupunki.")
```

Tiedon lukeminen käyttäjältä

```
# Kysytään merkkijonoa
teksti = input("Anna tekstiä: ")

# Kysytään syöte ja muunnetaan se kokonaisluvuksi
numero = int(input("Anna kokonaisluku: "))

# Kysytään syöte ja muunnetaan se liukuluvuksi
desimaali = float(input("Anna desimaaliluku: "))
```

Valintarakenne

```
# Tarkistetaan täyttävätkö muuttujat ehtoja ja tulostetaan tekstiä sen
# mukaan
luku = 1
nimi = "Esko"
if luku == 0:
    print("Luku on nolla.")
elif luku == 1:
    print("Luku on yksi.)
    if nimi == "Esko":
        print("Esko Mörkö.")
    else:
        print("Ei Mörköä.")
else:
    print("Luku ei ole binäärinen.")
```

Toistorakenteet

```
# Ikisilmukka, toistojen määrä ei ole tiedossa etukäteen
while True:
    sana = input("Anna sana (tyhjä lopettaa): ")
    if len(sana) == 0:
          break
    print("Sanasi oli", sana)
```

```
# Useamman ehdon toistorakenne
luku1 = 10
luku2 = 1
while ((luku1 >= luku2) and (luku2 < 10000)):
    print("luku1: {0}, luku2: {1}".format(luku1, luku2))
    luku1 = luku1 + luku2 - 1
    luku2 = luku2 * 2

# Tunnetun kokonaislukulistan läpikäynti
lkm = int(input("Anna kakkosen potenssien lukumäärä: "))
for i in range(0, lkm):
    print("2^" + str(i) + "=" + str(2 ** i))

# Listan läpikäynti
lista = ["Apina", "Banaani", "...", "Zeniitti"]
for alkio in lista:
    print(alkio)
```

Aliohjelmat

```
# Pelkkää tulostusta sisältävä aliohjelma
def valikko():
    print("Valikko")
    print("1) Avaa tiedosto")
    print("2) Tallenna tiedosto")
    print("3) Järjestä tiedoston sisältö")
    print("0) Lopeta")
# Kutsutaan aliohjelmaa sen nimellä
valikko()

# Aliohjelma parametrillä ja paluuarvolla eli funktio. Itse funktio
# palauttaa parametrinsä itseisarvon
def itseisarvo(luku):
    if luku < 0:
        luku = luku * -1
    return luku

# Luodaan kokonaislukulista ja tulostetaan listan alkiot ja niiden
# itseisarvot funktion itseisarvo avulla
luvut = [1, -3, 0, -192299, 3097]
for alkio in luvut:
    print("Luvun", alkio, "itseisarvo on", itseisarvo(alkio))
```

Tiedostoon kirjoittaminen

```
# Otetaan sys mukaan, jotta ohjelman suoritus voidaan keskeyttää
# sys.exit():llä
import sys
try:
    lista[] = ["1", "2", "3", "4"]
    tiedosto = open("luvut.txt", "w", encoding="utf-8")
    try:
        for alkio in lista:
            tiedosto.write(alkio + "\n")
    except:
        print("Tiedostoon kirjoittaminen ei onnistunut.")
    finally:
        tiedosto.close()
except IOError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(-1)
```

```
print("Tiedostoon kirjoittaminen suoritettu onnistuneesti.")
```

Tiedostosta lukeminen

```
# Avataan tiedosto kirjoittamista varten ja kirjoitetaan käyttäjän
# syötteet. Tällä kerralla käytetään with-lausetta huolehtimaan
# tiedoston sulkemisesta.
try:
    with open("luvut.txt", "r", encoding="utf-8") as tiedosto:
        while True:
            rivi = tiedosto.readline()
            if len(rivi) == 0:
                break
            rivi = rivi[0:-1]
            print(rivi)

except IOError:
    print("Tiedostoa ei voitu avata.")
else: # Käytetään else-osioita kertomaan kaiken menneen hyvin.
    print("Tiedoston lukeminen suoritettu onnistuneesti.")
```

Listan käsittely

```
# Luodaan lista ja kysytään käyttäjältä siihen positiivisia
# kokonaislukuja
lista = []
while True:
    luku = int(input("Anna positiivinen kokonaisluku (negatiivinen
lopettaa): "))
    if luku < 0:
        break
    lista.append(luku)
print(lista)

# Järjestetään listan alkiot pienimmästä suurimpaan
lista.sort()
print(lista)

# kysytään käyttäjältä arvo, joka poistetaan listasta
arvo = int(input("Anna poistettava arvo: "))
lista.remove(arvo)
print(lista)

# Kysytään käyttäjältä poistettavan alkion järjestysnumero ja poistetaan
# alkio
indeksi = int(input("Anna poistettavan alkion järjestysnumero: "))
del lista[indeksi-1]
print(lista)

# Lista voidaan tulostaa myös erilaisilla for-silmukoilla
for alkio in lista:
    print(alkio)
for i in range(0, len(lista)):
    print(lista[i])
```

Monimutkaisempi tietorakenne

```
# Luodaan kaksi auto-sanakirjaa ja autokauppa-lista
auto1 = {}
auto2 = {}
autokauppa = []
```

```
# Lisätään autokauppaan kaksi autoa
auto1["merkki"] = "Lada"
auto1["vuosimalli"] = 1982
autokauppa.append(auto1)
auto2["merkki"] = "Honda"
auto2["vuosimalli"] = 2003
autokauppa.append(auto2)

# Käydään kaikki autokaupan autot lävitse ja tulostetaan tiedot ruudulle
for myytavana in autokauppa:
    print("Myydään", myytavana["merkki"], "vuosimallia",
str(myytavana["vuosimalli"]) + ".")
```

Poikkeukset

```
# Kysytään kahta lukua ja yritetään jakaa ensimmäinen toisella.
luku1 = int(input("Anna ensimmäinen kokonaisluku: "))
luku2 = int(input("Anna toinen kokonaisluku: "))
try:
    print("{0}/{1}={2}".format(luku1, luku2, luku1 / luku2))
except ZeroDivisionError:
    print("Nollalla ei voi jakaa!")
print("Luvut olivat", luku1, "ja", luku2)
```