Python 3 – ohjelmointiopas

versio 1.2.1

Erno Vanhala ja Uolevi Nikula

LUT Scientific and Expertise Publications – Oppimateriaalit 22
LUT Scientific and Expertise Publications – Lecture Notes 22

Python 3 – ohjelmointiopas
versio 1.2.1

Erno Vanhala ja Uolevi Nikula

LUT-yliopisto
School of Engineering Science
Ohjelmistotuotanto
PL 20
53851 Lappeenranta

ISBN 978-952-335-622-1
ISSN-L 2243-3392
ISSN 2243-3392

Lappeenranta 2020

Tämä Python 3 -ohjelmointiopas perustuu Python-oppaaseen, joka on julkaistu Creative Commons Attribution-NonCommercial-ShareAlike 2.5 lisenssin alaisuudessa. Python Software Foundationin dokumentit on julkaistu GNU General Public Licence – yhteensopivan erillislisenssin alaisuudessa. How to think like a Computer Scientist: Learning with Python on julkaistu GNU Free Documentation –lisenssillä.

Tähän dokumenttiin sovelletaan ”Creative Commons Nimi mainittava-Ei kaupalliseen käyttöön- Sama lisenssi 2.5” – lisenssiä. Python 3 -ohjelmointiopas on ei-kaupalliseen opetuskäyttöön suunnattu opas.

Alkuperäisen suomenkielisen Python-oppaan toteutus:

Jussi Kasurinen

Python 3 oppaan toteutus:

Erno Vanhala

Toteutuksen ohjaus ja jatkokehitys:

Uolevi Nikula

LUT-yliopisto, Ohjelmistotuotanto

Lappeenranta 31.12.2020

Tämä ohjelmointiopas on tarkoitettu ohjeeksi, jonka avulla lukija voi perehtyä Python-ohjelmoinnin perusteisiin. Ohjeet sekä esimerkkitehtävät on suunniteltu siten, että niiden ei pitäisi aiheuttaa ei-toivottuja sivuvaikutuksia, mutta siitäkin huolimatta lopullinen vastuu oppimisesta on käyttäjällä. Oppaan tekemiseen osallistuneet henkilöt taikka LUT eivät vastaa käytöstä johtuneista suorista tai epäsuorista vahingoista, vioista, ongelmista, tappioista tai tuotannon menetyksistä. Opasta ei ole valmistettu tuotantolinjalla, jolla käsitellään pähkinää, mutta herneitä tuotantoprosessin läpi on kulkenut lukuisia.

Sisällysluettelo

Johdanto eli miksi ohjelmointi on kivaa ja tärkeää 1
Luku 1: Python ohjelmointiympäristö ja ohjelma 2
Python-ohjelmien tekeminen IDLEllä 2
Ohjelmat käsittelevät tietoa 6
Laskutoimitukset 9
Tiedon kysyminen käyttäjältä 12
Osaamistavoitteet 13
Luvun asiat kokoava esimerkki 14
Luku 2: Tietotyypit, muuttujien roolit ja tulosteiden muotoilu 15
Merkkijono tietotyyppinä 15
Merkkijonojen yhdistely 16
Merkkijonojen leikkaukset 17
Lukujen tyyppimuunnokset ja pyöristys 20
Muuttujien roolit 21
Tulosteiden muotoilu 22
Osaamistavoitteet 24
Luvun asiat kokoava esimerkki 24
Luku 3: Valintarakenne 25
Sisennys 25
if-valintarakenne 26
Ehtolausekkeet ja loogiset operaattorit 29
Operaattorien suoritusjärjestys 31
Boolen-arvoista 32
Osaamistavoitteet 33
Luvun asiat kokoava esimerkki 33
Luku 4: Toistorakenteet 34
while-rakenteen käyttäminen 34
for-rakenne 35
break-käsky 36
continue-käsky 37
range-funktiosta 38
else-osio toistorakenteessa 39
Osaamistavoitteet 39
Luvun asiat kokoava esimerkki 39
Luku 5: Ohjelman rakenne 40
Funktion määrittely 40

Funktiokutsu ja parametrien välitys 41
Funktion päättyminen ja paluuarvo 42
Nimiavaruus 44
Tunnus eli muuttuja, aliohjelma tai kiintoarvo 45
Pääohjelma eli `paaohjelma()` 45
Tunnusten näkyvyys 46
Funktioiden dokumentaatiorivit ja help-funktio 47
Pythonin valmiina tarjoamia funktioita 47
Osaamistavoitteet 48
Luvun asiat kokoava esimerkki 49
Luku 6: Tiedoston käsittely 50
Tiedostojen kanssa työskentely 50
Tiedostonkäsittelyyn liittyviä metodeja 53
Usean tiedoston käyttö samaan aikaan 55
Muotoiltu tulostus 56
Merkkijonojen jäsenfunktiot 57
Osaamistavoitteet 60
Luvun asiat kokoava esimerkki 61
Luku 7: Rakenteiset tietorakenteet 62
Lista 62
Yleisimpiä listan jäsenfunktioita 65
Lista aliohjelman parametrina 66
Arvo- ja muuttujaparametrit sekä paluuarvot 68
Luokka-rakenne 68
Tuple-rakenne 72
Osaamistavoitteet 73
Luvun asiat kokoava esimerkki 73
Luku 8: Kirjastot ja moduulit 75
Esikäännetyt pyc-tiedostot 76
from...import -sisällytyskäsky 76
Omien moduulien tekeminen ja käyttäminen 77
Kirjastomoduuleja 78
Ulkoisten kirjastojen lisääminen Pythoniin 83
Osaamistavoitteet 84
Luvut asiat kokoava esimerkki 84
Luku 9: Virheenkäsittelyä 86
Virhetilanteiden estäminen ennakolta 86
Virheistä yleisesti 86

try…except-rakenne 87
try…finally 89
Kuinka poikkeustenkäsittely kannattaa toteuttaa? 90
Osaamistavoitteet 91
Luvun asiat kokoava esimerkki 92
Luku 10: Data analytiikka ja uusia tietorakenteita 93
Sanakirja (eng. Dictionary) 93
Kuinka järjestää sanakirja tai tuple 95
Huomioita sarjallisten muuttujien vertailusta 96
Matriisi eli numpy-moduulin taulukko 97
Osaamistavoitteet 99
Luvun asiat kokoava esimerkki seuraavalla sivulla 99
Luku 11: Algoritmi, pseudokoodi ja rekursio 101
Käytännön ongelmasta ohjelmaksi 102
Rekursiivinen ohjelma 104
Osaamistavoitteet 105
Luku 12: Tiedon esitysmuodoista 106
Merkkitaulukot 108
Binaaritiedostot eli sananen pickle-moduulista 109
Pari sanaa pyöristämisestä 111
Ajan esitystapa tietokoneessa 112
Osaamistavoitteet 112
Luku 13: Käyttöliittymistä 113
Graafisten käyttöliittymien perusteet 113
Komentorivikäyttöliittymä ja komentoriviparametrit 116
Osaamistavoitteet 117
Loppusanat 118
Lisäluettavaa 118
Lähdeluettelo 119
Liite 1: Lyhyt ohje referenssikirjastoon 120
Liite 2: Yleinen Python-sanasto 121
Liite 3: Tulkin virheilmoitusten tulkinta 125
Liite 4: Tyyliopas 127
Liite 5: Esimerkkirakenteita 132

Alkusanat

Tervetuloa LUT-yliopiston Python-ohjelmointioppaan pariin. Tämä on oppaan kahdeksas uusittu ja päivitetty versio. Alkuperäisen ohjelmointioppaan toteutti Jussi Kasurinen. Tämä uusin versio pohjaa siihen, mutta kaikki materiaali on päivitetty Pythonin versiolle 3 ja sisältöä on kehitetty sekä laajennettu muutenkin.

Ohjelmointioppaan ensimmäisen versio tehtiin käyttäen hyväksi kolmea verkkolähdettä. Näistä ensimmäinen on CH Swaroopin kirjoittama teos 'Byte of Python' (www.byteofpython.info), toinen on Python Software Foundationin ylläpitämä Python-dokumenttiarkisto (docs.python.org) ja kolmas on Allen B. Downey'n kirjoittama 'How to Think Like a Computer Scientist: Learning with Python' (http://www.greenteapress.com/thinkpython/), josta on otettu lähinnä täydentävää materiaalia. Python 3 -päivityksen yhteydessä käytettiin myös Mark Pilgrim'n Dive Into Python 3 -oppaan materiaalia (http://diveintopython3.org/). Alkuperäinen opas oli vapaa käännös ja yleisesti ottaen käännöksen alkuperäinen kieliasu pyrittiin säilyttämään, mutta tekstejä on muunneltu luettavuuden ja jatkuvuuden parantamiseksi. Myös esimerkit on lokalisoitu englanninkielisestä Linux-shell-ympäristöstä suomenkieliseen IDLE-kehitysympäristöön. Myöhempien päivitysten yhteydessä oppaan sisältö on muuttunut ja kehittynyt, joten yhtäläisyydet alkuperäisiin lähteisiin vähenevät koko ajan.

Tämä opas on tehty LUTin Ohjelmoinnin perusteet -kurssin ohjelmointioppaaksi. Oppaan rakenne perustuu kurssin viikkojakoon ja sisältöä muokataan kurssin kehittämisen yhteydessä. Oppaan liitteissä on tiivistettyä tietoa Pythonin kirjastoista, Pythonin yleinen sanasto, tulkin virheilmoitusten tulkintaopas, ohjelmoinnin tyyliopas sekä Python ohjelmien tyypillisiä ohjelmointirakenteita. Oppaan tueksi on olemassa myös asennusvideo, jossa käydään läpi asennuksessa oleelliset asiat.

Kaikki oppaan tehtävät on tehty Pythonin IDLE-kehitysympäristössä Windows-ympäristössä Pythonin versiolla 3.7.2. Samat työkalut on saatavilla myös Mac- tai Linux-ympäristöihin.

Tämä opas on päivitetty ja korjattu versio aiemmin ilmestyneistä Python-ohjelmointioppaista ja korvaa aiemmat versiot ohjelmoitaessa Pythonin versiolla 3.

Johdanto eli miksi ohjelmointi on kivaa ja tärkeää

Tietokoneiden kehitys lähti kasvuun toisen maailmansodan jälkeen 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 ja siksi ne kirjoitettiin 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 konekielelle, mikä teki ohjelmien tuottamisesta paljon nopeampaa. Ensimmäisissä ohjelmointikielissä ei ollut paljoa toimintoja, mutta eipä ollut tietokoneissakaan tehoja – tai näyttöjä. Ohjelmakoodi oli usein varsin järjestelemätöntä, sillä niin kutsuttua spagettikoodia oli helppo kirjoittaa.

Seuraava vaihe oli siirtyä rakenteiseen ohjelmointiin. Ensin tulivat proseduraalinen ohjelmointi ja sitten olio-ohjelmointi, joista 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.

Kaikki edellä mainitut ohjelmointiparadigmat kehitettiin 1950- ja 60-luvuilla ja niiden nouseminen vallitsevaan asemaan kesti vain 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, puhumattakaan monimutkaisista Excel-taulukoista tai interaktiivisen PowerPoint-esityksen taustalla olevasta ohjelmoinnista.

Ohjelmia voidaan kirjoittaa oikein ja ei-niin-oikein eli väärin. Kannattaa jo alusta lähtien opetella ohjelmoimaan oikein, sillä se helpottaa elämää (nimim. 10 vuotta väärin ohjelmoinut). Tämän oppaan tarkoitus onkin perehdyttää lukija ohjelmoinnin ihmeelliseen maailmaan ja tarjota hyväksi todettuja tapoja ratkaista ongelmia ohjelmoimalla.

Kannattaa pitää mielessä, että ohjelmoimaan oppii vain yhdellä tavalla – ohjelmoimalla.

Oppimisen riemua!

Luku 1: Python ohjelmointiympäristö ja ohjelma

Ohjelmoinnin opettelu edellyttää ohjelmointiympäristöä, joten aloitetaan tutustumalla Python-ohjelmointiympäristöön. Sen jälkeen käydään läpi muutamia keskeisimpiä asioita ohjelmien tekemiseen liittyen kuten tiedon kysyminen käyttäjältä, tiedon muokkaaminen ml. tietotyyppien muunnokset ja tiedon tulostaminen. Tutustumme myös muuttujiin ja muihin ohjelmoinnissa tarvittaviin peruskonsepteihin. Tämän luvun tavoitteena on tulla tutuksi ohjelmointiympäristön ja ohjelmoinnin kanssa, jotta pystyt tekemään ja ajamaan ohjelmia sekä opetelemaan uusia ohjelmointikonsepteja laajentamalla aiempia ohjelmia.

Python-ohjelmien tekeminen IDLEllä

Python-tulkin lataamisen yhteydessä (http://python.org) samassa paketissa tulee IDLE-editori ohjelmien kirjoittamiseen. Python-ohjelmat ovat tekstitiedostoja, joissa käytetään .txt -päätteen sijasta .py -päätettä. Aloitetaan ensin yksinkertaisella ohjelmalla ja tutustutaan muutamiin ohjelman rakenteeseen liittyviin yleisiin asioihin, jotta jatkossa voimme keskittyä itse ohjelmointiin.

Editori ja tulostaminen

Windows-käyttäjille helpointa on aloittaa Pythonin käyttö IDLE-editorilla, joka on saatavilla myös Linux- ja MacOS-järjestelmille. IDLE on lyhenne sanoista Integrated DeveLopment Environment, ja sen haku- ja asennusohjeet löytyvät erillisestä asennusohjeesta. 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 kaksi erilaista ikkunaa, interaktiivinen komentorivitulkki- ja editori-ikkunat.

Huomaa, että jatkossa esimerkkien merkintä **`>>>`** tarkoittaa tulkille syötettyä Python-käskyä. Seuraavassa kirjoitamme interaktiiviseen ikkunaan käskyn **`print("Hello World!")`** ja painamme Enter-näppäintä.

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC
v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> print("Hello World!")
Hello World!
>>>
```

Huomaa, että Python palauttaa tulostetun rivin välittömästi näytölle. Kirjoitit siis itse asiassa yksinkertaisen Python-käskyn `print`. Python käyttää `print` –käskyä sille annettujen arvojen tulostamiseen ruudulle. Tässä esimerkissä annoimme sille tekstin ”Hello World!”, jonka se tulosti välittömästi ruudulle.

Lähdekooditiedostot

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”. Edellä tulostimme tämän tekstin interaktiivisesta ikkunasta, mutta kirjoitetaan tuo käsky nyt lähdekooditiedostoon, tallennetaan tiedosto ja suoritetaan ohjelma sen jälkeen IDLE-editorilla. Tässä vaiheessa jokaiselle ohjelmalle kannattaa tehdä oma tiedosto, jotta ohjelmat pysyvät erillään ja muodostavat selkeitä kokonaisuuksia.

Avaa valitsemasi editori, esim. IDLEssä File-valikosta valitaan New Window, ja kirjoita alla

olevan esimerkin mukainen koodi. Tämän jälkeen tallenna tiedosto nimellä `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. Ohjelma lähdekooditiedostossa

```
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!
>>>
```

Miten se toimii

Jos koodisi tuotti Tuloste-kohdassa näkyvän tekstin 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” on eri asia kuin ”print”. Varmista myös, että et epähuomiossa laittanut välilyöntejä tai muutakaan sisennystä rivin alkuun. Palamme tähän asiaan myöhemmin tässä luvussa.

Itse ohjelma sisältää nyt vain Python-käskyn `print`, joka siis tulostaa tekstin ”Hello World!” näytölle. Voidaan myös sanoa, että `print` on funktio ja ”Hello World!” on merkkijono, mutta näihin termeihin palataan myöhemmin tarkemmin. Tässä vaiheessa Python-ohjelma sisältää käskyjä ja `print`-käsky on yksi Pythonin yleisimmin käytettyjä käskyjä.

Kommenttirivit

Pythonissa voi 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 tunnistaa #-merkistä rivin ensimmäisenä merkkinä.

Kun tulkki havaitsee rivillä kommentin alkamista kuvaavan merkin ”#”, lopettaa tulkki kyseisen rivin lukemisen ja siirtyy välittömästi seuraavalle riville. Kannattaa huomata, että #-merkillä voidaan myös laittaa 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 seuraava koodi:

```
# 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 seuraavan tekstin näytölle

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

Käytännössä tulkki jättää kommenttirivit huomioimatta. Erityisesti tämä näkyy koodin toisessa tulostuskäskyssä, jota ei tällä kertaa huomioida, koska rivin alkuun on lisätty kommenttimerkki. Vaikka kommentoitu alue sisältäisi toimivaa ja ajettavaa koodia, ei sitä siitäkään huolimatta suoriteta – se on siis ”kommentoitu pois”.

Kommentteja kannattaa käyttää järkevästi. Kirjoita kommentteihin esim. mitä ohjelmasi mikäkin vaihe tekee, sillä tästä on hyötyä kun ohjelmaa tuntematon yrittää tulkita sen 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!

Loogiset ja fyysiset rivit

Fyysisellä rivillä tarkoitetaan sitä riviä, jonka näet koodia kirjoittaessasi – se alkaa ensimmäisestä sarakkeesta ja loppuu rivinvaihtomerkkiin. Looginen rivi taas on se kokonaisuus, minkä Python näkee yhtenä käskynä. Lähtökohtaisesti Python-tulkki käsittelee yhtä fyysistä riviä yhtenä loogisena rivinä.

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

On mahdollista kirjoittaa useampi looginen rakenne yhdelle riville, mutta se ei ole yleisesti ottaen suositeltavaa. Sen sijaan koodi kannattaa kirjoittaa niin, että yksi fyysinen rivi vastaa yhtä loogista käskyä. Käytä useampaa fyysistä riviä yhtä loogista riviä kohti ainoastaan poikkeustapauksessa, esim. jos rivistä on tulossa erittäin pitkä. Taka-ajatuksena tässä on se, että koodi pyritään pitämään mahdollisimman helppolukuisena ja yksinkertaisena tulkita tai tarvittaessa korjata.

Alla esimerkki tilanteesta, jossa looginen rivi jakautuu useammalle fyysiselle riville. Huomaa, että merkkijonon jakaminen useammalla riville edellyttää rivin lopettamista \-merkkiin (takakeno).

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

Tämä ohjelma tulostaa seuraavan merkkijonon:

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

Loogisen rivin jakaminen useammalla fyysiselle riville ei edellytä aina kenoviivaa rivin loppuun. Näin käy esimerkiksi silloin, kun looginen rivi sisältää sulkumerkin, joka loogisesti merkitsee rakenteen määrittelyn olevan kesken. Esimerkiksi listojen tai muiden sarjamuotoisten muuttujien kanssa tämä on tavallista ja palaamme tähän asiaan luvussa 8.

Tyypillisiä virheilmoituksia

Ohjelmia kirjoittaessa saattaa tulla kirjoitusvirheitä ja tulkki huomauttaa niistä. Esimerkiksi edellä oli puhetta print ja Print –käskyjen erosta eli siitä, että kirjoittamalla ohjelmaan käskyn Print, ei tulkki suorita käskyä vaan antaa seuraavan virheilmoituksen:

```
NameError: name 'Print' is not defined
```

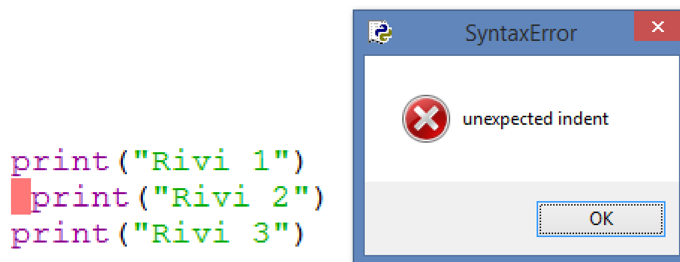
Tämä ongelma on helppo korjata vaihtamalla iso P pieneksi p:ksi.

Toinen varsin tyypillinen virhe liittyy koodin sisennykseen. Tässä vaiheessa ohjelmoinnin opiskelua kirjoitamme kaikki rivit alkamaan ensimmäisestä sarakkeesta ja jos näin ei toimita, antaa tulkki virheilmoituksen. Esimerkiksi seuraava kolmen rivin ohjelma johtaa virheilmoitukseen:

```
print("Rivi 1")
 print("Rivi 2")
print("Rivi 3")
```

Tällä kertaa tulkki ilmoittaa virheestä kahdella tavalla, näyttämällä SyntaxError –dialogin eli ”kirjoitusvirhe” ja sen lisäksi koodissa virheen aiheuttanut kohta on merkitty punaisella taustavärillä Kuva 1.1 mukaisesti. Virheilmoitusten muoto ja esitystapa voivat vaihdella eri versioissa ja ympäristöissä, mutta näitä virheilmoituksia kannattaa opetella lukemaan, sillä niiden avulla on tyypillisesti helppo löytää koodista virheen aiheuttanut kohta ja vähän asiaa pohtimalla syykin on usein helppo keksiä. Tulkki ei suorita ohjelmaa ennen kuin sen löytämä virhe on korjattu.

Kuva 1.1. Virhe lähdekoodissa



Tässä tapauksessa virheen syy on toisen rivin alussa olevassa ylimääräisessä välilyönnissä. *Syntax error* tarkoittaa sitä, että Python-koodissa on jotain täysin kieliopin vastaista, eikä tulkki pysty edes sanomaan mitä siitä pitäisi korjata. Toinen vastaava virheilmoitus on *IndentationError*, joka kertoo sisennyksen olevan pielessä. Kannattaa siis muistaa, että **et voi aloittaa fyysisiä rivejä satunnaisesta kohtaa koodiriviä.** Sisennyksiä ja rakenteita, joissa luodaan uusia osioita, käsitellään tarkemmin luvussa 3.

Yhteenveto

Tässä vaiheessa osaat käyttää IDLE-editoria, luoda ja tallentaa lähdekooditiedostoja sekä kirjoittaa ja ajaa Python-ohjelmia. Hallitset työkalujen käytön ja pystyt kirjoittamaan ohjelmia, jotka tulostavat ruudulle tekstiä alla olevan Esimerkki 1.2 mukaisesti. Myös muutamat tyypilliset virheet ja virheilmoitukset ovat sinulle tuttuja, joten pystyt etsimään ja korjaamaan ohjelmissa olevia virheitä. Kun ohjelmointiympäristö on sinulle nyt tuttu, voimme siirtyä tutustumaan tarkemmin ohjelmien tekemiseen.

Esimerkki 1.2. Sämpyläohje

```
# 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")
print()
print("Liota hiiva nesteeseen. Lisää muut aineet ja alusta")
print("vehnäjauhoilla kimmoisaksi. Anna kohota 15 min. Leivo")
print("taikinasta sopivia palleroita ja anna kohota pellillä.")
print("Paista 225°C n. 12-15 min.")
```

Ohjelmat käsittelevät tietoa

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

Pythonilla pystyy käsittelemään tietoa eri tavoin ja käsittelytavat riippuvat käsiteltävästä tiedosta. Lähtökohtaisesti **merkkijonot** ja **numerot** ovat erilaisia tietotyyppejä, esimerkiksi nimi kuten ”Kalle” on merkkijono, jossa on monta (aakkos/ASCII) merkkiä peräkkäin. Ja numero voi taas olla esimerkiksi **kokonaisluku** 5 tai **desimaaliluku** 5,12345. Lisäksi tieto voi olla **vakio**, joka ei siis voi muuttua, tai tieto voi muuttua, jolloin puhutaan **muuttuja**sta. Pythonissa ei ole ohjelman rakenteeseen liittyviä vakioita vaan kaikkea tietoa voi muuttaa. Useimmissa muissa ohjelmointikielissä tietoalkio voi olla vakio, jolloin ohjelmointiympäristö varmistaa, ettei tieto muutu ohjelman suorituksen aikana. Käydään nämä käsitteet läpi seuraavaksi vähän tarkemmin, jotta ymmärrämme paremmin miten ne liittyvä ohjelmointiin.

Vakio ja merkkijono

Tähän mennessä tekemämme ohjelmat ovat tulostaneet merkkijonoja ja ”Hello World” on tyypillinen *merkkijonovakio.* Kyseessä on siis jono merkkejä eli kirjaimia, jotka eivät muutu vaikka ohjelman ajaisi kuinka monta kertaa. Tilanne ei muutu, jos vaihdamme lainausmerkkien sisälle numeron `5, 10` tai `655.36`. Tällöin meillä on edelleen käytössä merkkijonovakio, koska ohjelma käsittelee tietoja kirjainmerkkeinä. Huomaa, että merkkijonot erotetaan ohjelmakoodissa lainausmerkeillä (”) tai heittomerkeillä (’) ja ne ovat siis vakioita.

Ohjelmaan voidaan kirjoittaa myös numeroita kuten `5, 10` tai `655.36` ilman lainausmerkkejä. Jos ohjelmassa on esimerkiksi käsky `print(1.2345)`, niin se tulostaa näytölle luvun `1.2345` yhtä monta kertaa kuin ohjelma suoritetaan. Pelkkä numero on vakio, eikä sitä voi muuttaa.

Numerot

Pythonissa on kolme erilaista numerotyyppiä: 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, jossa 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 rupeaa nopeasti rajoittamaan ohjelmien joustavuutta. Tarvitsemme jonkinlaisen keinon tallentaa tietoa sekä tehdä niihin muutoksia. Tämä on syy, miksi ohjelmointikielissä, ml. Python, 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 käyttöönottovaiheessa. Muuttujiin liittyy siis kaksi asiaa – muuttujan nimi ja sen arvo – ja muuttujan arvoa voidaan muuttaa aina tarpeen mukaan.

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, että käytettävä muuttuja on jo aiemmin määritelty joksikin soveltuvaksi arvoksi. Täten emme voi esimerkiksi tulostaa määrittelemättömän muuttujan arvoa.

Alla on määritelty kaksi merkkijonomuuttujaa, `etunimi` ja `sukunimi`, sekä kaksi numero muuttujaa, `ika` eli kokonaislukumuuttuja sekä `paino` eli liukulukumuuttuja (tai desimaaliluku).

```
etunimi = ”Brian”
sukunimi = ”Kottarainen”
ika = 23
paino = 123.4
```

Muuttujien nimeäminen

Muuttujan nimi ovat esimerkki *tunnisteesta*. 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. Huom. Python 3:n myötä ääkköset periaatteessa käyvät, mutta niitä **EI** tule käyttää.
- Nimet ovat aakkoskoosta riippuvaisia (eng. case sensitive), eli isot ja pienet kirjaimet ovat tulkille eri merkkejä. Tulkin kannalta ”talo” ja ”Talo” ovat eri sanoja samalla tavoin kuin ”talo” ja ”valo”.

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`.

Palaamme hyvään ohjelmointityyliin tässä oppaassa aina sopivan tilaisuuden tullen, mutta muuttujien nimeämiseen liittyy mm. alla olevia hyviä käytäntöjä. Muutaman rivin ohjelmissa nimillä ei ole suurta merkitystä, mutta ohjelmien koon kasvaessa kymmeniin ja satoihin riveihin muuttujien nopea ja tarkka tunnistaminen helpottaa ohjelmointia

merkittävästi. Kannattaa siis harjoitella hyvien nimien käyttämistä heti alusta alkaen.

- Muuttujien nimet kannattaa valita niin, että ne kuvaavat käsiteltävää tietoa.
- Älä käytä skandinaavisia merkkejä muuttujien, muiden tunnusten tai tiedostojen nimissä.
- Ole tarkkana isojen ja pienten kirjainten kanssa, sillä niiden sekoittuminen on tyypillinen syy ohjelmissa oleviin virheisiin.

Muuttujien tietotyypit

Muuttujille voidaan antaa arvoksi 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 eli `32 - "auto"` ei siis toimi.

Huomaa myös seuraavat erot kun käsitellään numeroita ja merkkijonoja:

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

Numero-muuttujilla voidaan tehdä normaaleja laskutoimituksia, mutta merkkijonojen osalta vain yhteenlasku on sallittua. Numero ja merkkijonomuuttujien välillä on myös määritelty muutamia operaatioita, mutta niihin palataan myöhemmin. Lähtökohtaisesti muuttujilla kannattaa käyttää kuvaavia nimiä ja laskentaoperaatioita tehdessä kannattaa käyttää maalaisjärkeä. Kuten edellä näkyy, numeroiden yhteenlasku tuottaa odotetun tuloksen, mutta merkkijonojen yhteenlasku johtaa niiden yhdistämiseen. Pythonissa kokonaisluvuilla ja desimaalinluvuilla voidaan laskea suoraan eikä Python erottele niitä tällaisessa käytössä mitenkään.

Laskentaohjelma

Kirjoita alla olevan Esimerkki 1.3 mukainen ohjelma IDLEllä ja tallenna tiedosto nimellä `muuttuja.py`. Muista antaa Python-koodille pääte **py**, jotta käyttöjärjestelmä ja editorit tunnistavat tiedostosi Python-koodiksi. Suorita ohjelma ja katso, että ymmärrät sen toiminnan.

Esimerkki 1.3. Muuttujien käyttäminen ja vakioarvot

```
# Tiedosto: muuttuja.py

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

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

Tuloste

```
>>>
4
5
4
20
Tässä meillä on tekstiä.
>>>
```

Kuinka se toimii

Ohjelmasi toimii seuraavasti: Ensin määrittelemme muuttujalle `luku` vakioarvon `4` käyttäen sijoitusoperaattoria `(=)`. Tätä riviä sanotaan käskyksi, koska rivillä on määräys, että jotain pitäisi tehdä. Tässä tapauksessa siis sijoittaa arvo `4` muuttujaan `luku`. Sijoitusoperaatio toimii loogisesti aina siten, että se arvo, johon sijoitetaan on operaattorin vasemmalla puolella ja se, mitä sijoitetaan oikealla. Seuraavalla rivillä olevalla toisella käskyllä me tulostamme muuttujan `luku` arvon ruudulle funktiolla `print.` Ja kun hyppäämme taas rivin eteenpäin tulostamme ruudulle arvon `luku + 1`, eli tässä tapauksessa arvon `5`. Muuttujamme `luku` on edelleen arvoltaan `4`, minkä kertoo myös seuraava `print`-lause.

Tämän jälkeen kasvatamme luvun arvoa yhdellä ja tallennamme sen takaisin muuttujaan `luku`. Eli vasta tässä vaiheessa muuttujan arvo muuttuu! 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.

Laskutoimitukset

Pythonin interaktiivista ikkunaa voi käyttää laskimena. Anna 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. Sijoitusoperaattoriin päättyvä lauseke ei tuota välitulosta vaan sillä voit syöttää vakiotietoja, joilla taas voit suorittaa laskutehtäviä. Alla olevassa esimerkissä muuttujiin `leveys` ja `korkeus` sijoitetaan arvot, joilla sitten tehdään varsinainen laskutoimitus:

```
>>> 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
```

Operaattorit ja operandit

Seuraavaksi tutustumme yleisimpiin operaattoreihin, jotka liittyvät muuttujilla operointiin. Aikaisemmin olet jo tutustunut niistä muutamaan, (+) (*) ja (=) -operaattoreihin ja tärkeimmät operaattorit on kuvattu Taulukko 1.1.:ssa.

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

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

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

Laskentajärjestyksestä

Lähtökohtaisesti suoritusjärjestys noudattaa matemaattisia laskusääntöjä. Kuten normaalisti matematiikassa, voit 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`.

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

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

Taulukko 1.1 Laskentaoperaattorit

Operaattori	Nimi	Selite	Esimerkki
=	Sijoitus	Sijoittaa annetun arvon kohdemuuttujalle	`luku = 5` sijoittaa muuttujalle luku arvon 5. Operaattori toimii ainoastaan mikäli kohteena on muuttuja.
+	Summa	Laskee yhteen kaksi operandia	`3 + 5` antaa arvon `8`. `'a' + 'b'` antaa arvon `'ab'`.
-	Erotus	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`.

Laskut ohjelmakoodissa

Lähdekooditiedostoon tallennetut Python-käskyt toimivat vastaavalla tavalla kuin interaktiivisessa ikkunassa. Voit siis kokeilla kaavoja ja käskyjä interaktiivisessa ikkunassa ja sitten tallettaa ne tiedostoon ja suorittaa ohjelman osana, kuten Esimerkki 1.4 osoittaa.

Esimerkki 1.4. Lausekkeiden käyttö

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

Tiedon kysyminen käyttäjältä

Usein tulee tilanne, että etukäteen luodut merkkijonot tai koodiin kiintoarvoina määritellyt muuttujat eivät pysty hoitamaan kaikkia tarvittavia tehtäviä. Haluamme syöttää ohjelmaan arvoja ajon aikana ja antaa käyttäjälle mahdollisuuden vaikuttaa ohjelman tuottamiin tuloksiin. Seuraavaksi tutustumme Pythonin tarjoamiin tapoihin pyytää käyttäjältä syötteitä ja tallentaa niitä muuttujiin.

Pythonissa käyttäjältä voidaan kysyä tietoa `input`-käskyllä (funktiolla). Käskyn perään laitetaan sulkuihin käyttäjälle näytettävä teksti ja käsky palauttaa käyttäjän antaman syötteen. Huomaa, että käsky palauttaa kaikki merkit rivinvaihtomerkkiin asti, mutta ilman sitä. Ja tieto tosiaan tulee ohjelmaan yhtenä merkkijonona. Nimen kysyminen on tyypillinen esimerkki tästä (Esimerkki 1.5).

Esimerkki 1.5. input-funktion käyttö

```
# Tiedosto: input.py

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

Tuloste

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

Kuinka se toimii

Tässä esimerkissä pyydämme käyttäjältä merkkijonon `input`-käskyllä, sijoitamme sen muuttujaan ja tulostamme sen lopuksi käyttäjälle `print`-käskyllä. Kaikki tieto käyttäjältä saadaan Pythonissa `input`-käskyllä ja tiedot tulostetaan käyttäjälle `print`-käskyllä, joten niitä kannattaa opetella käyttämään.

Tyyppimuunnokset merkkijonosta luvuksi

Kuten aiemmin oli puhetta, Pythonissa on kolme perustietotyyppiä – merkkijono, kokonaisluku ja desimaaliluku – ja näiden välillä voidaan tehdä tyyppimuunnoksia. Muunnokset luvuiksi tehdään funktioilla `int`() ja `float`(), joista `int` palauttaa kokonaisluvun, integer'n, `float` palauttaa desimaaliluvun, floating point number, ja molemmille annetaan suluissa muutettava arvo. Esimerkki 1.6. demonstroi näitä funktioita. Huomaa, että Pythonissa desimaalierotin on piste eikä pilkku.

Esimerkki 1.6. Lukujen kysyminen käyttäjältä, int ja float

```
# Tiedosto: lukuja.py

x = input("Anna ympyrän säde kokonaislukuna: ")
sade = int(x)
p = input("Anna piin likiarvo desimaalilukuna: ") # odotetaan arvoa 3.14
pii = float(p)

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

Tuloste

```
>>>
Anna ympyrän säde kokonaislukuna: 3
Anna piin likiarvo desimaalilukuna: 3.14
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, nyt ”Anna ympyrän säde kokonaislukuna: ”, ja tulostaa tämän merkkijonon ruudulle toimintaohjeeksi käyttäjälle. Käyttäjä syöttää mieleisensä luvun vastauksena kehotteelle, jonka jälkeen annettu luku tallentuu muuttujaan `x`. Pythonin `input`-funktio ei tiedä, että käyttäjä syöttää nimenomaan numeron, vaan se ottaa vastaa merkkijonon, tässä tapauksessa merkin ”3”. Laskussa tarvitsemme numeroa, joten muutamme `int`-funktiolla käyttäjän syöttämän merkkijonon numeroksi. Piin arvon osalta idea on sama, mutta nyt pyydämme käyttäjältä desimaaliluvun kokonaisluvun sijasta. Tämä jälkeen voimme laskea ja tulostaa pinta-alan ja kehän pituuden edellisen esimerkin tavoin.

Muunnettaessa merkkijono kokonaisluvuksi tai desimaaliluvuksi ei ole varmaa, että muunnos onnistuu. Siksi tyyppimuutoksissa kannattaa olla tarkkana ja palaamme myöhemmin tarkemmin tyyppimuunnoksiin liittyvään virheenkäsittelyyn. Siihen asti kysyttäessä käyttäjältä tietoja kannattaa antaa hyvä ohjeet siitä, minkälaista tietoa hänen tulisi syöttää: kokonaisluku, desimaaliluku tai merkkijono.

Osaamistavoitteet

Tämän ensimmäisen luvun tavoitteena oli käydä läpi kaikki perusasiat Python-ohjelmointiympäristöön ja perusohjelmaan liittyen, jotta pääset jatkossa tekemään Python-ohjelmia ja opettelemaan uusia käskyä ja rakenteita. Sinulla pitäisi nyt siis olla toimiva Python-ympäristö käytössä, jossa tärkeimmät asiat ovat IDLE-editori ja interaktiivinen ikkuna, jossa ohjelmia ajetaan. Itse ohjelmointiin liittyen kävimme läpi lyhyesti muuttujat, tietotyypit, ja tietojen kysymisen sekä tulostamisen. Erotat koodin seasta kommentit ja tiedät mitä eroa on loogisella ja fyysisellä rivillä sekä osaat käyttää Pythonia laskimena. Tältä pohjalta on hyvä jatkaa ohjelmoinnin opettelua.

Luvun asiat kokoava esimerkki

Esimerkki 1.7 toimii luvun asiat kokoavana esimerkkinä ohjelmasta, jonka pystyt nyt tekemään. Tiivistetysti siinä näkyy tiedon kysyminen, tyyppimuunnoksia ja tiedon tulostaminen.

Esimerkki 1.7. Leivontaohjeen rakennus

```
# Tiedosto: jauhopeukalo2.py

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

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

Luku 2: Tietotyypit, muuttujien roolit ja tulosteiden muotoilu

Tutustuimme tietotyyppeihin jo ensimmäisessä luvussa ja nyt katsomme vähän tarkemmin mistä merkkijonoissa, kokonaisluvuissa ja desimaaliluvuissa on kysymys sekä miten tietotyyppejä niitä muuttaa Python-ohjelmissa. Merkkijonon osalta keskeisiä asioita ovat perusteiden lisäksi niiden yhdistely sekä alimerkkijonojen käsittely, kun taas desimaalilukuja pitää pystyä pyöristämään ja muuttamaan kokonaisluvuiksi. Tutustuimme muuttujiin jo edellisessä luvussa, mutta nyt katsomme lyhyesti mistä muuttujien rooleissa on kysymys, sillä ne auttavat tekemään selkeitä ohjelmia. Luvun päätteeksi palaamme tietojen tulostamiseen ja katsomme tapoja muokata tulosteita käyttäjän haluamaan muotoon.

Merkkijono tietotyyppinä

Merkkijono on jono peräkkäisiä merkkejä. Merkkijonot voivat olla esimerkiksi sanoja tai lauseita, mutta merkkijonoksi lasketaan mikä tahansa joukko merkkejä. Merkkijonoja käytetään ohjelmoinnissa paljon, joten seuraava osio kannattaa lukea ajatuksella lävitse ja palata siihen aina tilanteen niin vaatiessa. Pythonissa merkkijonoihin liittyy erityisesti seuraavat asiat:

Heittomerkki (')

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

Lainausmerkki (”)

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

Ohjausmerkit

Oletetaan, että haluat käyttää merkkijonoa, joka sisältää heittomerkin ('). 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 heittomerkkiin merkkijonon olisi tarkoitus päättyä. Tässä tilanteessa joudut jotenkin kertomaan tulkille, mihin heittomerkkiin tulee lopettaa. Tämä onnistuu ohjausmerkillä (\), jonka avulla voit merkata yksinkertaisen heittomerkin ohitettavaksi tyyliin \’. Nyt esimerkkirivi `'vaa\'an alla'` toimii ilman ongelmia.

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

Entä jos haluat tulostaa useammalle riville? Voit käyttää rivinvaihtomerkkiä (\n). Rivinvaihtomerkki tulee näkyviin tekstiin normaalisti kenoviiva-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.

Esimerkkejä

Alla on esimerkkejä merkkijonoista ja miten ne tulostuvat Pythonissa. Huomaa erityisesti heitto- ja lainausmerkkien näkyminen tulosteessa.

```
>>> '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.'
```

Merkkijonojen yhdistely

Usein merkkijonoja joutuu yhdistämään toisiin merkkijonoihin tai numeroihin ja nämä molemmat onnistuvat vaikka ne tehdäänkin eri tavoin. Heti alkuun on syytä huomata, että jos laitat kaksi merkkijonoa vierekkäin, Python yhdistää ne automaattisesti. Esimerkiksi merkkijonot `'Vaa\'an' ' alunen'` yhdistyvät tulkin tulostuksessa merkkijonoksi `"Vaa'an alunen"`.

Tyypillinen tarve on yhdistää kaksi erillistä merkkijonoa yhdeksi merkkijonoksi. Kuten luvussa 1 oli puhetta, voidaan merkkijonoja yhdistä toisiinsa ”+”-operaattorilla. Ja ihan vastaavasti kuin matematiikassa monta summausoperaatiota voidaan hoitaa kertomerkillä, Pythonissa saman merkkijonon voi tulostaa monta kertaa ”*”-operaattorilla.

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

Merkkijonon ja luvun yhdistäminen vastaa omenan ja appelsiinin yhdistämistä eli se ei onnistu, koska ne ovat erilaisia asioita eikä ole selvää, mitä lopputuloksen tulisi olla – omena, appelsiini vai hedelmäsalaatti. Toinen asia on, että nämä kaksi eri asiaa voidaan

tulostaa yhdessä print-lauseessa ja toisaalta ne voidaan muuttaa samantyyppisiksi, jolloin ne voidaan yhdistää. Jos lukua ja merkkijonoa yrittää yhdistää toisiinsa print-lauseessa, tulee siihen tyypillisesti seuraavanlaisia kommentteja tulkilta:

```
>>> ika = 3
>>> print(ika + "-vuotias poika.")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(ika + "-vuotias poika.")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

tai

```
TypeError: Can't convert 'int' object to str implicitly
```

Kun koodia on enemmän, myös virheilmoitus vaihtuu hieman. Molemmat kertovat kuitenkin samasta virheestä: merkkijonon ja numeron yhdistäminen ei onnistu.

Tarve on selkeä – haluamme tulostaa lauseen, jossa `ika`-kokonaislukumuuttujaa seuraa ”`-vuotias poika.`”-merkkijono. Ongelma voidaan ratkaista muuttamalla kokonaisluku merkkijonoksi `str`-funktiolla:

```
>>> print(str(ika) + "-vuotias poika.")
3-vuotias poika.
>>>
```

Nyt `print` saa sisäänsä kaksi merkkijonoa, jotka voidaan yhdistää toisiinsa ja ohjelma tulostaa halutun lopputuloksen. Toinen vaihtoehto on hyödyntää print-käskyn muita ominaisuuksia, joita käsitellään tarkemmin tämän luvun lopussa Tulosteiden muotoilu –kohdassa.

Merkkijonojen leikkaukset

Merkkijonoja voidaan yhdistellä toisiinsa ”+”-operaattorilla, mutta merkkijonojen jakaminen ”-”-operaattorilla ei onnistu. Joskus merkkijonosta pitää kuitenkin saada otettua kiinnostava kohta erilleen tarkempaa käsittelyä varten. Esimerkiksi suomalainen henkilötunnus sisältää paljon tietoa ja irrottamalla siitä sopivia osia tarkempaan käsittelyyn saadaan esimerkiksi selvitettyä ko. henkilön syntymävuosi ja siten voidaan laskea hänen ikä.

Merkkijonojen käsittely perustuu merkkien paikkaan jonossa. Esimerkiksi merkkijono ”heippa” sisältää 6 merkkiä, ensimmäinen merkki on ’h’, viimeinen merkki on ’a’ ja 4. ja 5. merkin sisältävä alimerkkijono on ”pp”. Jos tätä merkkijonoa ajattelee Python-ohjelman kannalta, ovat ensimmäinen ja viimeinen merkki samoja, mutta 4. ja 5. merkin sisältävä alimerkkijono olisi ”pa”.

Merkkijonon pituuden pystyy selvittämään Pythonissa sisäänrakennetulla funktiolla `len`. Funktiolle annetaan syötteenä muuttuja tai merkkijono, ja funktio palauttaa pituuden merkkeinä:

```
>>> s ='Apumiehensijaisentuuraajankorvaajanlomittajanpaikka'
>>> len(s)
51
>>> len(”heippa”)
6
```

On tärkeää huomata, että viimeinen merkki on paikalla `s[50]` ja `s[5]`, sillä funktio `len` palauttaa todellisen pituuden merkkeinä ja merkkijonon ensimmäisen merkin järjestysnumero on 0. Koska ”merkkijonojen leikkaukset” tarkoittaa alimerkkijonon ottamista merkkijonosta merkkien indekseihin perustuen, tulee nuo indeksi laskea Pythonin

tavalla. Yksi tapa muistaa miten merkkijonon numeroiden leikkaukset lasketaan, on ajatella numeroiden sijaan niiden välejä alla olevan esimerkin mukaisesti. Kuvan numerorivi kertoo merkin sijainnin laskettuna normaalisti vasemmalta oikealle.

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

Merkkijonon merkkejä ja alimerkkijonoja voidaan siis käsitellä indeksien avulla. Pythonin indeksointi alkaa nollasta ja asiaa on perusteltu monilla tavoin. Esimerkiksi voidaan ajatella, että indeksi kertoo siirtymän sanan alusta ja koska sanan ensimmäinen merkki on jo alussa, on siirtymä 0 ja siten indeksi on myös sama 0. Asiaa on perusteltu myös matemaattisella kauneudella eli nolla on pienin ei-negatiivinen kokonaisluku.

Käytännössä merkkijonon leikkaukset määritellään hakasuluilla ja numerosarjalla, jossa ensimmäinen numero kertoo aloituspaikan, toinen leikkauksen lopetuspaikan ja kolmas siirtymävälin eli [alku:loppu:siirtymä]. Hakasulkujen sisällä numerot erotellaan toisistaan kaksoispisteillä. Kaikissa tilanteissa kaikkia kolmea indeksiä ei ole pakko käyttää, kuten alla olevat esimerkit osoittavat:

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

Leikkaus sisältää hyödyllisiä ominaisuuksia. Kaikkia lukuarvoja ei aina ole pakko käyttää ja ensimmäisen luvun oletusarvo on 0, toisen luvun oletusarvo viimeinen merkki ja siirtymävälin oletusarvo on +1.

```
>>> sana[:2]    # Ensimmäiset kaksi kirjainta
'Te'
>>> sana[2:]    # Kaikki muut kirjaimet paitsi kaksi ensimmäistä
'ekkari'
>>> sana[0::2]  # Alusta loppuun joka toinen kirjain
'Tekr'
>>> sana[::]    # Koko sana
'Teekkari'
```

Lisäksi tulee muistaa, että Pythonissa merkkijonojen muuttamisessa 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 käyttää myös 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'
```

Huomaa, 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'
```

Nollasta alkava indeksointi ja negatiivisten lukujen käyttö indeksinä voi tuntua alussa oudolta, mutta nekin oppii harjoittelemalla. Kertauksen vuoksi vielä yllä ollut kuva, jossa nyt ylempi numerorivi kertoo kirjaimen sijainnin laskettuna normaalisti vasemmalta oikealle ja alempi rivi negatiivisilla luvuilla laskettuna oikealta vasemmalle. Ja indeksiä -0 ei ole olemassa.

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

Lukujen tyyppimuunnokset ja pyöristys

Pythonissa on sisäänrakennettuja tyypinmuunnosfunktiota tietotyypin vaihtamiseen, 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` muuttaa liukulukuja kokonaisluvuiksi, mutta ei pyöristä niitä. Tämä tarkoittaa käytännössä sitä, että `int` ainoastaan katkaisee luvun desimaaliosan pois. Lisäksi `int` osaa 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 liittyy läheisesti **lukujen pyöristys**. Käyttäjän kannalta ei ole mielekästä katsella pitkää desimaalijonoa ruudulla etenkin, 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 3 (parametreista ja funktiosta lisää myöhemmin), pyöristää `round` luvun kolmen desimaalin tarkkuuteen. Toinen esimerkki selventää lisää.

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

Kannattaa huomata, että `round` ei lisää lukuun (turhia) desimaaleja, vaan pyöristyksessä näytetään vain merkitsevät numerot. Eli `round(3.10000, 3)` tulostaa 3.1. Tällainen pyöristys on käytössä Pythonin versiolla 3.7.2 ja aiemmilla. Tulevaisuudessa asia voi tietysti muuttua, mikäli muutos koetaan tarpeelliseksi. Mikäli lukuun halutaan tasamäärä numeroita, täytyy käyttää muotoiltua tulostusta, johon palaamme myöhemmin.

Tyypillisiä virheilmoituksia

Tyyppimuunnosfunktio samoin kuin pyöristysfunktio eivät muuta alkuperäistä tietoa vaan tekevät siitä uuden esitysmuodon. Tämä tarkoittaa sitä, että jos uutta esitysmuotoa halutaan hyödyntää myöhemmin, se pitää tallentaa muuttujaan. Esimerkki havainnollistaa tätä tyypillistä ongelmaa:

```
>>> 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
>>>
```

Yllä tyyppimuunnoksen jälkeen tieto tallennetaan alkuperäiseen muuttujaan, mikä on tässä tapauksessa luonteva ratkaisu, koska muuttujan nimi on ”`luku`” eli voidaan olettaa, että sillä voi laskea jotain matemaattisiin kaavoihin perustuen. Käytännössä tyyppimuunnettu arvo tulee sijoittaa muuttujan arvoksi sijoitusoperaattorilla (”=”).

Muuttujien roolit

Olemme puhuneet muuttujista ja nimensä mukaisesti muuttuja sisältää tietoa, joka voi *muuttua*. Tarkemmin ottaen muuttujilla on useita erilaisia käyttötarkoituksia ohjelmoinnissa ja näitä tarkoituksia voidaan kutsu rooleiksi. Yksi tyypillinen muuttajan rooli on tiedon säilyttäminen ilman muutoksia eli toimiminen *kiintoarvona*. Kiintoarvo saa arvonsa yhden kerran ja pitää ko. arvon ohjelman loppuun asti. Ohjelman lähtökohtana voi esimerkiksi olla, että tenttiin voi osallistua korkeintaan 100 tenttijää ja tämän rajoitteen muistamiseksi voimme määritellä ohjelman alkuun kiintoarvon seuraavalla tavalla:

```
TENTTI_OSALLISTUJIA_MAX = 100
```

Tämän jälkeen voimme ohjelmassa tarkistaa esimerkiksi uuden tentti-ilmoittautumisen yhteydessä, ettei tenttijöiden määrä ylity vertaamalla aiemmin ilmoittautuneiden määrää maksimimäärään. Teknisesti `TENTTI_OSALLISTUJIA_MAX` on siis Pythonissa muuttuja, mutta kiintoarvo-roolin takia sen arvoa ei tule muuttaa missään kohdassa ohjelmaa.

Toinen tyypillinen muuttujien rooli on *tilapäissäilö*, jolloin muuttuja on tilapäinen tiedon säilytyspaikka, sitä käytetään tyypillisesti ymmärrettävyyden vuoksi ja ohjelma olisi voitu tehdä myös ilman sitä. Alla on esimerkki laskusta, jossa kahden luvun summa sijoitetaan tilapäissäilöön `summa`, jotta laskentaa on helpompi seurata. Etenkin jos laskussa on useita (kymmeniä) välivaiheita, helpottavat tilapäissäilöt laskennan ymmärtämistä ja oikeellisuuden tarkistamista merkittävästi.

```
luku1 = 2
luku2 = 7
summa = luku1 + luku2
tulos = summa * 5
print(luku1, luku2, tulos)

>>>
2 7 45
>>>
```

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

Tulosteiden muotoilu

Pythonissa tulostus tapahtuu `print`-käskyllä niin kuin kaikissa tähän asti tehdyissä ohjelmissa näkyy. Olemme kuitenkin käyttäneet vasta `print`-käskyn perusmuotoa ja käsky tarjoaa merkittävästi laajemmat mahdollisuudet tulosteiden muotoiluun.

Lähtökohtaisesti `print`-käsky tulostaa sille suluissa annetun parametrin näytölle. Käytännössä parametreja voi olla monta pilkuilla eroteltuina ja ne kaikki tulostetaan samalla riville välilyönnein eroteltuna.

```
>>>
>>> print(5)
5
>>> nimi = "Ville"
>>> print("Moi", nimi, 5, "vuotta.")
Moi Ville 5 vuotta.
>>>
```

Vaikka tulostus toimiikin näin ihan hyvin, ei lopputulos ole kaikkien mielestä riittävän viimeisteltyä. Esimerkiksi yllä olevaan tervehdykseen pilkun laittaminen nimen perään ilman välilyöntiä ei onnistu vaan tarvitaan jotain lisäkeinoja tulostuksen muotoiluun. Yksi vaihtoehto on ajatella tulostusta yhtenä merkkijonona ja käyttää aiemmin tässä luvussa esiteltyjä yhdistämistä ja leikkauksia hyväksi tulosteen muodostamisessa.

```
>>> nimi = "Ville"
>>> tuloste = "Moi " + nimi + ", " + str(5) + " vuotta."
>>> print(tuloste)
Moi Ville, 5 vuotta.
>>>
```

Tätä esimerkkiä katsoessa voi miettiä, että onko yhdestä välilyönnistä eroon pääseminen kaiken tuon vaivan arvoista, mutta jos maksava asiakas pyytää muotoilemaan tulosteen tarkemmin niin silloin on hyvä olla keinoja siihen. Usein `print`-lauseen tuloste kannattaa tosiaan muodostaa merkkijonoista yhdistelemällä ja silloin pitää muistaa, että kaikki

yhdistettävät osat ovat merkkijonona eikä välissä ole esimerkiksi lukuja.

`print`-käskyssä voi käyttää myös parametreja `end` ja `sep`, joiden oletusarvot ovat vastaavasti `'\n'` ja `' '` eli rivinvaihtomerkki ja välilyönti. `print`-käsky tulostaa suluissa olevat parametrit näytölle eroteltuina `sep`-parametrin merkillä (ts. separator-merkki, kenttäerotin) ja lisää niiden perään `end`-parametrin osoittaman merkin (ts. rivin loppumerkki). Vaihtamalla näiden parametrien arvoja voidaan muuttaa tulosteen rakennetta alla olevan esimerkin 2.1 mukaisesti.

Esimerkki 2.1. Tulosteen muotoilua `end` ja `sep` parametreilla

```
# 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.")

print()
print("Kenttäerotin vaikuttaa tulosteeseen myös: ")
print(1, 2, 3, 4, 5)
print(1, 2, 3, 4, 5, sep='-')
```

Tuloste

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

Kenttäerotin vaikuttaa tulosteeseen myös:
1 2 3 4 5
1-2-3-4-5
>>>
```

Kuinka se toimii

Ensimmäinen printti toimii tutulla tavalla 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 normaalin tekstin lisäksi oleva avainsana `end=""` kertoo `print`-funktiolle, että rivin loppuun tuleva merkki on tyhjä ts. sitä ei ole. Oletusarvona käytössä oleva rivinvaihto (`"\n"`) siis poistuu ja seuraava rivi tulostuukin edellisen perään. `end`-avainsanalla voidaan määrittää mikä tahansa merkkijono rivin loppuun `print`-lausekkeessa, myös nyt käytössä oleva tyhjä merkkijono.

`print`-lauseen toinen avainsana on `sep` eli separator. Yllä olevan esimerkin lopussa näkyy, miten oletusarvona olevan välilyönnin korvaaminen tavuviivalla näkyy tulosteessa. Huomaa myös, että tyhjän rivin saa tulostettua yksinkertaisesti `print()` –käskyllä.

Osaamistavoitteet

Tässä kappaleessa kävimme tarkemmin läpi tietotyypit eli merkkijonon, kokonaisluvun ja desimaaliluvun sekä niihin liittyviä käytännön seikkoja. Merkkijonojen yhdistely ja leikkaukset sekä desimaalilukujen pyöristäminen ovat normaaleja toimenpiteitä ohjelmoitaessa ja käsiteltäessä tietoa. Tiedon esittäminen eli tulostaminen on keskeisessä roolissa kun käyttäjä haluaa nähdä mitä on saatu aikaiseksi ja silloin on hyvä pystyä muokkaamaan tulosteet käyttäjän haluamaan muotoon. Tulostukseen tulemme palaamaan vielä kerran laajemmin sen tärkeyden takia entistä tehokkaampiin työkaluihin tutustumisen merkeissä.

Luvun asiat kokoava esimerkki

Esimerkki 2.2. Jalojuoma

```
# 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 3: Valintarakenne

Tähän asti olemme tehneet ohjelmia, joissa on ollut aina joukko samassa järjestyksessä suoritettavia peräkkäisiä käskyjä. Entäpä, jos haluaisimme koodin tekevän vertailuja tai laittaa mukaan osioita, jotka ajetaan ainoastaan tarvittaessa? Esimerkiksi miten tekisimme ohjelman, joka sanoo ”Hyvää huomenta” tai ”Hyvää iltaa” kellonajan perusteella?

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

Sisennys

Ohjelmia kirjoitettaessa on jo pitkän aikaa pähkäilty sitä, miten koodi saataisiin helppolukuiseksi. Aiemmin mainitut loogiset ja fyysiset rivit ovat yksi koodin lukemista helpottava käytäntö eli yksi rivi, yksi asia. Toinen koodin luettavuutta ja ylläpidettävyyttä parantava asia on koodin sisennys.

Pythonissa **tyhjät merkit (whitespace) rivin alussa ovat merkitseviä merkkejä**. Tätä sanotaan **sisennykseksi**. Nämä rivin alun ”tyhjät merkit” eli välilyönnit ja tabulaattorimerkit määrittelevät loogisen rivin sisennyksen syvyyden, jolla määritellään mihin loogiseen joukkoon rivi kuuluu. Tämä tarkoittaa sitä, että loogisesti samaan ryhmään kuuluvat koodirivit sijoitetaan samalle sisennystasolle. Sisennystasoa, joka taas sisältää loogisen Python-käskyn, sanotaan osioksi tai koodilohkoksi. Tässä ja seuraavassa luvussa tutustumme näiden osioiden käyttöön.

Seuraava esimerkki näyttää, millaisia ongelmia sisennystason asettaminen väärin tyypillisesti 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 näyttää “syntax error” –ikkunan, mutta edelleen tietokone kieltäytyy suorittamasta kirjoittamaasi koodia.

Virheen syy on toisen rivin alussa olevissa ylimääräisissä välilyönneissä. *Syntax error* tarkoittaa sitä, että Python-koodissa on jotain kieliopin vastaista, eikä tulkki pysty tulkitsemaan koodia yksikäsitteisesti. Yllä oleva *IndentationError* kertoo sisennyksen olevan pielessä. Käytännössä sinun tulee muistaa tästä osiosta lähinnä se, että **et voi aloittaa koodiriviä satunnaisesta paikasta vaan sinun on noudatettava sisennysten ja osioiden kanssa loogista rakennetta**.

Ohjeita sisennysten hallintaan

Älä sekoita välilyöntejä ja tabulaattoreita sisennyksissä. 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 valita itselle luontaisen tyylin, joka toimii ja pysyä siinä. Ole johdonmukainen sisennystesi kanssa ja pysyttele vain ja ainoastaan yhdessä tyylissä.

if-valintarakenne

`if`-rakenne perustuu koodissa olevaan loogiseen väittämään eli jos väittämä on tosi (`True`) suoritetaan `if`-lauseeseen liittyvä koodiosio. Usein väittämään liittyy kaksi vaihtoehtoista toimintoa, joista ensimmäinen tehdään väittämän ollessa tosi (”jos tosi niin toimi näin”) ja väittämän ollessa epätosi tehdään toinen toiminto (”muutoin toimi näin”). Tämä toinen osio toteutetaan Pythonissa `else`-osiona eli `if...else...`. Aina `if`-rakenteen jälkeen koodi jatkuu alkuperäisellä tasolla eteenpäin. Rakenteessa voi olla myös `elif` (else if)-osioita, joilla useita `if`-lauseita voidaan ketjuttaa peräkkäin siten, että useita eri vaihtoehtoja voidaan testata samassa rakenteessa. `elif`-osioita voi `if`-rakenteessa olla mielivaltainen määrä; `else`-osio on vapaaehtoinen eli se voi puuttua tai se voi olla mukana, mutta kumpaakaan ei voi olla olemassa ilman `if`-osiota, joita voi olla ainoastaan yksi per rakenne. Kuulostaako vaikealta? Ei hätää, onneksi näin ei ole ja seuraava esimerkki selventää asiaa.

Esimerkki 3.1. Käytetään if -rakennetta

```
# 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.

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 käyttäjän syöttäessä arvon 6, sillä 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 koodilohkoon ja tulostetaan ”Taitaa olla parempi pitää rokulipäivä... Zzz...”. Lopuksi tulostetaan käyttäjän syötteestä riippumatta ”Kiitos käynnistä!”.

Kannattaa muistaa, että `if`-rakenteita voi olla sisäkkäin eli `if`-osio, `elif`-osiot tai `else`-osio voivat kaikki sisältää vapaasti lisää `if`−rakenteita. On myös hyvä muistaa, että `elif` ja `else`-osien käyttö on valinnaista. Yksinkertaisin `if`−rakenne näyttää esim. tällaiselta:

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

`if`-rakenne vaatii tarkkuutta eli muista laittaa `if`-, `elif`- ja `else`-osien perään kaksoispiste, jotta Python-tulkki tietää uuden osion alkavan siitä. Lisäksi sisennykset vaativat tarkkuutta. Jotta `if`-rakenteen käyttäminen tulisi selväksi, otamme vielä toisen esimerkin.

Esimerkki 3.2. if-elif-else-rakenne, kertaus

```
# 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 esimerkissä kysymme ensin käyttäjän nimeä ja käytämme tämän jälkeen `if-elif-else`-rakennetta tulostamaan 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 huomata, että `if-elif-else`-rakenteesta vain yksi sisennetty koodilohko suoritetaan. Mikäli nimeksi tunnistetaan ”*Matti*”, ei seuraavia `elif`-osioita tai `else`-osiota suoriteta koskaan.

Toinen huomion arvoinen asia on yhtäsuuruusvertailuoperaattorin (”==”) ja sijoitusoperaattorin (”=”) ero. **Vertailua ei koskaan toteuteta vain yhdellä yhtäsuuruusmerkillä!** Tulkki kyllä huomauttaa, jos tällaista yritetään.

Huomioita if-rakenteesta

Kuten aiemmin mainittiin, ei `if`-rakenteesta tarvitse aina käyttää 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 tekemättä.

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

Huomaa, että `if-if-if-else`- ja `if-elif-elif-else`-rakenteissa on selkeä ero. Koska `if`-rakenteita voidaan sijoittaa monta peräkkäin, ei mikään varsinaisesti estä meitä käyttämästä useita peräkkäisiä `if`-rakenteita korvaamaan `elif`-osiot. Tässä yhteydessä on kuitenkin muistettava, kuinka `if`-rakenne toimii. Tyypillisesti haluamme, että `if`-rakenteessa ohjelma tekee oikeanlaiset loogiset päätelmät annetuista ehdoista ja jatkaa eteenpäin. Katsotaan tarkemmin seuraavia kahta valintarakenteen toteutusta:

```
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 toteuttavat näennäisesti samat testit, tulostaa ylempi vastauksen "pienempi kuin 1000" kun taas alempi tulostaa "pienempi kuin 100". Miksi näin tapahtuu?

Kyse on 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 erillisten `if`-lauseiden kohdalla jokainen `if`-väittämä testataan erikseen siitä huolimatta, oliko aiempi samaan rakenteeseen kuuluva `if`-lause ollut totta.

Ohjelma toimii molemmissa tapauksissa näennäisen 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 käytettäisiin vaikkapa 10-potenssin tunnistamiseen, olisi vastaus hyödytön vaikkakin teknisesti täysin oikein. Ohjelma voi siis sisältää loogisia virheitä, vaikka se olisi syntaksin osalta kirjoitettu oikein.

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.

Tähän liittyy läheisesti Boolen logiikka, josta löytyy lisätietoa esim. Wikipediasta hakusanoilla *boolen logiikka*. Taulukoissa 3.1 ja 3.2 on esitelty tarvittavia operaattoreita ja niiden käyttöä demonstroidaan sen jälkeen esimerkissä 3.3.

Taulukko 3.1. Loogiset- eli vertailuoperaattorit

Operaattori	**Nimi**	**Selite**	**Esimerkki**
<	Pienempi kuin	Palauttaa tiedon onko x pienempi 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 suurempi kuin y.	`5 > 3` palauttaa arvon `True`. Jos molemmat operandit ovat numeroita, ne muutetaan automaattisesti vertailukelpoisiksi eli samantyyppisiksi. Merkkijonoista verrataan ensimmäisiä (vasemmanpuolimmaisia) kirjaimia merkistön mukaisessa järjestyksessä. Tähän palataan luvussa 12.
<=	Pienempi 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 3.2 Boolen-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 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 laskenta-aikaa.
or	Boolean OR	Jos x on `True`, palautuu arvo `True`, muulloin 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.

Taulukossa 3.3 on listattuna Pythonin operaattorit niiden suoritusjärjestyksen mukaisesti korkeimmasta matalimpaan. Tämä tarkoittaa sitä, että useita operaattoreita sisältävässä lausekkeessa niiden toteutusjärjestys on listan mukainen.

Taulukko on oiva apuväline esimerkiksi testauslausekkeita tarkastettaessa, mutta käytännössä on suositeltavampaa käyttää sulkeita laskujärjestyksen varmistamiseen. Esimerkiksi `2 + (3 * 4)` on lukijan kannalta paljon selkeämpi kuin `2 + 3 * 4`. Sulkeiden kanssa kannattaa kuitenkin käyttää järkeä ja välttää turhien sulkeiden käyttöä, koska se haittaa laskutoimituksen luettavuutta ja ymmärrettävyyttä. Esimerkiksi laskutoimitus `(2 + ((3 + 4) – 1)+(3))` on hyvä esimerkki siitä, mitä ei pidä tehdä.

Yksi selkeä tapa käyttää sulkuja valintarakenteessa on laittaa jokainen vertailulauseke sulkujen sisään. Näin jokaisen sulkulausekkeen totuusarvon voi selvittää helposti ja tarpeen tullen Boolen operaattoreilla yhdistetyn lausekkeen tulos on helppo katsoa niiden pohjalta.

Taulukko 3.3. Operaattorien suoritusjärjestys

Operaattori	**Selite**	**korkea prioriteetti**
`sisältö, ...`	Merkkijonomuunnos	
{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 sitä, että lauseke `a = b = c` tulkitaan olevan `a = (b = c)`.

Boolen-arvoista

Python tukee loogisten lausekkeiden kanssa työskennellessä Boolen-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
>>>
```

Boolen–arvojen (tosi/epätosi) käyttö on loogisissa lausekkeissa luonnollisempaa kuin pelkkien numeroiden käyttö.

Esimerkki 3.3. Boolen-arvot ja operaattorit

```
# 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 Boolen-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.

Osaamistavoitteet

Ymmärrät `if-elif-else` rakenteen ja sen eri variaatiot sekä osaat käyttää niitä ohjelmissa. Ymmärrät vertailu- ja Boolen-operaattorien käytön sekä suoritusjärjestyksen.

Luvun asiat kokoava esimerkki

Esimerkki 3.4. Ehtoilua

```
# Tiedosto: valikkorakenteista.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 4: Toistorakenteet

Ihmisen elämässä usein laatu korvaa määrän eli käytämme vaikka derivointiin hienoa derivointikaavaa. Toisaalta tietokone osaa tehdä vain rajatun määrän erilaisia toimintoja (eikä se osaa derivoida suoraan), joten usein toistojen määrä saattaa olla suurikin. Tässä luvussa tutustumme kahteen eri tapaan toteuttaa toistorakenne: `while` ja `for`.

while-rakenteen käyttäminen

Esimerkki 4.1. Numeronarvauspeli while-rakenteen avulla

```
# 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 silmukka pyörii näennäisen loputtomasti.

Muuttujan rooli: tuoreimman säilyttäjä

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

Silmukan sisällä kysymme käyttäjältä arvauksena kokonaislukua ja vertaamme sitä kiintoarvoon `oikea_numero`. Mikäli luvut ovat samoja, 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 ohjelmoijan täytyy huolehtia siitä, että toistorakenne voi joskus saavuttaa lopetusehtonsa. Usein tämän lisäksi käyttäjän on annettava lopetusehdon täyttävä syöte, sillä muussa tapauksessa `while`-lause jää ikuiseen toistoon ja ohjelma menee jumiin. `while`-rakennetta voidaan myös käyttää numeroarvojen kanssa työskennellessä:

Esimerkki 4.2. while-rakenteen toiminta

```
# 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 sen jälkeen toteudu.

for-rakenne

`for`-rakenne on toinen Pythonin kahdesta tavasta toteuttaa toistorakenne. `for`-lause eroaa `while`-rakenteesta kolmella merkittävällä tavalla. Ensinnäkin `for`-lauseen kierrosmäärä on tiedettävä lauseen suorituksen alkaessa. Toiseksi `for`-lausetta voi käyttää monialkioisten rakenteiden kuten listojen alkioiden läpikäymiseen. Kolmanneksi `for`-lause on suunniteltu etukäteen tiedettävien asioiden läpikäyntiin, eikä sen kanssa voida käyttää useita lopetusehtoja. Tällä kertaa keskitymme `for`-lauseen käyttämiseen sen perinteisemmässä muodossa eli silmukoiden tekemisessä. Listoihin ja niiden läpikäyntiin palaamme myöhemmin.

Esimerkki 4.3. for-rakenteen toiminta

```
# 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, sekvenssin, yhdestä viiteen pois lukien 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.

Muuttujan rooli: askeltaja

Esimerkin 4.2 muuttuja `arvo` ja 4.3 `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.*

Tässä on hyvä huomata, että esimerkiksi aiemmin käytetyn `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 joukon, jonka `for`-lause käy läpi kohta kerrallaan, tässä tapauksessa tulostaen aina jäsenen numeron. `for`-lause osaakin käydä tällä tavoin läpi kaikkia Pythonin sarjarakenteisia muotoja, vaikka tällä erää riittää ymmärtää kuinka `for`-lause yleisesti ottaen toimii.

`for` ja `while` -rakenteiden erot näkyvät kätevästi liitteessä 5 olevista koodiesimerkeistä.

break-käsky

Ensimmäisessä `while`-esimerkissämme 4.1 törmäsimme 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`.

Esimerkki 4.4. Toistorakenne break-käskyllä höystettynä

```
# Tiedosto: break.py

for i in range(1,11):
    if (i == 7):
        print("Onnennumero", i, "löytyi!")
        break
    print("Tutkitaan numeroa", i)

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, hypätään ulos silmukasta ja jätetään loput numerot tutkimatta. `break`-käsky tarkoittaa välitöntä poistumista koko toistorakenteesta eikä mitään muita siihen liittyviä käskyjä enää suoriteta.

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

continue-käsky

`continue`–käskyn toteutus vastaa `break`-käskyn toteutusta, vaikka tavoite onkin erilainen. `break`-käsky lopettaa toistorakenteen suorittamisen jättäen sen loput käskyt suorittamatta, kun taas `continue` jättää toistorakenteen loppuosan suorittamatta ja siirtyy suoraan ohjelman seuraavan kierroksen alkuun.

Esimerkki 4.5. Toistorakenne continue-käskyllä höystettynä

```
# Tiedosto: 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 samalla tavalla myös `for`-rakenteen kanssa.

range-funktiosta

Aiemmin `for`-rakenteen yhteydessä mainittiin funktio `range`. Kyseinen funktio on sisäänrakennettu Pythoniin ja se mahdollistaa `for`-lauseen käyttämisen normaalin toistorakenteen tavoin myös silloin, kun tieto ei ole tallennettuna sarjamuotoiseen muuttujaan. `range`-funktion käyttö on varsin yksinkertaista kuten alla olevasta esimerkistä näkyy. Käytämme esimerkissä myös `list`-funktiota; funktiot käydään tarkemmin läpi seuraavassa luvussa ja listat sen jälkeen.

```
>>> list(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 pyydetyt 10 arvoa, mutta viimeinen arvo on 9. Lisäksi `range`-funktiota käytettäessä voidaan käyttää leikkausmaisia arvomäärittelyjä, kun määritellään aloituslukua, lopetuspaikkaa ja siirtymäväliä. `rangen` oletusarvot ovatkin aloituspaikalle 0 sekä askelvälille 1. Lopetuspaikka täytyy aina määritellä käsin, koska sille oletusarvoa ei ole annettu.

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

Helpoiten `rangen` käytön hahmottaa seuraavan esimerkin avulla. Esimerkki tulostaa saman tulostuksen kahteen kertaan.

```
>>> for i in [1,2,3]:
        print(i)
1
2
3
>>> for i in range(1,4):
        print(i)
1
2
3
```

Palaamme listoihin luvussa 7 ja tällä erää riittää, että osaat käyttää `range`-funktiota.

else-osio toistorakenteessa

`else`-osiota voi käyttää toistorakenteessa kertomaan mitä tehdään, jos toistorakenteen läpikäyminen ei ole tuottanut toivottua tulosta ja toistorakenne loppuu toistoehdon täyttymiseen. Jos toistorakenne lopetetaan `break`-käskyllä, ei `else`-osiota suoriteta. Tästä tarkentavana esimerkkinä voidaan ottaa ohjelma, joka laskee alkulukuja:

Esimerkki 4.6. else-osio toistorakenteessa, alkuluvut

```
# 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, "*", int(n/x))
            break
    else: # Huomaa, että tämä else on jälkimmäiselle forille!
        # 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
>>>
```

Osaamistavoitteet

Osaat käyttää sekä `while`- että `for`-toistorakenteita ja ymmärrät niiden erot. Ymmärrät miten `break`, `continue`, `range` ja `else` -osioita käytetään toistorakenteiden yhteydessä.

Luvun asiat kokoava esimerkki

Esimerkki 4.7. Silmukointia

```
# 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 5: Ohjelman rakenne

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-ohjelmakin (virke) voi sisältää pääohjelman (päälause) ja aliohjelmia (sivulauseita).

Tähän mennessä olemme kirjoittaneet ohjelmia, joissa on vain pääohjelmaa vastaavaa päätason koodia. Tässä luvussa perehdymme päätason ohjelman tueksi 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 saamansa lopputuloksen. 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. Samalla funktioista muodostuu itsenäisiä kokonaisuuksia, joiden sisällä ohjelmoija saa tyypillisesti toteuttaa halutun toiminnallisuuden omalla tavallaan.

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. Tätä seuraa funktion koodi sisennettynä ja loppuen `return` -lauseeseen. Parhaiten asian ymmärtää alla olevasta esimerkistä; alkuun funktiot voivat vaikuttaa hämmentäviltä, mutta itse asiassa niiden käyttö on varsin suoraviivaista.

Aliohjelma vai funktio?

Tässä ollaan perimmäisten kysymysten äärellä. Kumpi oli ensin muna vai kana? Joissain ohjelmointikielissä käytetään erillisiä avainsanoja luomaan erilaisia aliohjelmia – proseduureja ja funktiota. Näiden erona pidetään sitä, että proseduuri ei palauta mitään, kun taas funktio palauttaa. Näin ollen Pythonissakin aliohjelma on funktio, jos se palauttaa jotain. Näiden kahden termin käyttö on kuitenkin harmaantunut vuosien saatossa ja usein funktiota käytetään aliohjelman synonyymina kuten myös tässä oppaassa. Loppujen lopuksi Pythonkin palauttaa aina jotain. Jos ohjelmoija ei laita funktiota palauttamaan mitään, lisää tulkki funktion loppuun automaattisesti tyhjän palautuksen.

Funktion määrittely

Esimerkki 5.1. Funktion määrittely ja kutsu

```
# Tiedosto: funktio1.py

# Aliohjelma alkaa
def sano_terve(): # Funktion määrittely alkaa tästä
    print("Terve vaan!")
    print("Tämä tulostus tulee funktion sisältä!")
    return None
# Aliohjelma loppuu

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

Tulostus

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

Kuinka se toimii

Edellä märittelimme funktion nimeltä `sano_terve` aiemmin mainittujen periaatteiden mukaisesti. Tämän funktion kutsussa ei ole muuttujia (parametreja), 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 ja paluukäsky `return` ilman palautettavaa arvoa, joten tähän on lisätty `None`. **Kun funktio saadaan suoritettua, palataan siihen kohtaan ohjelmaa, jossa funktiokutsu oli**, eli tässä tapauksessa suoritus jatkuu pääohjelman tulostuksella ”Funktio suoritettu”.

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 suoritettu). Lisätään päätasolle 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!
Toistetaan
Terve vaan!
Tämä tulostus tulee funktion sisältä!
>>>
```

Funktiota, joka on esitelty lähdekoodin alussa, voidaan kutsua aina uudelleen ja uudelleen päätason koodista niin useasti kuin haluamme. Tästä johtuen itse tehdyt funktiot esitellään ennen pääohjelmaa, koska Python-tulkki lukee asiat samassa järjestyksessä kuin me ihmiset eli tiedoston alusta loppuun. 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 kutsua myös Pythonin komentorivitulkissa. 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. Komentorivitulkissa ajo on mahdollista ohjelman suorittamisen jälkeen, jotta tulkki on lukenut ohjelman muistiin ja tietää missä se on.

Funktiokutsu ja parametrien välitys

Funktiokutsu voi sisältää parametreja, jotka ovat käytännössä 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

ja siksi niiden avulla voidaan tuota tietoa funktion ulkopuolelta sen sisälle.

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

Esimerkki 5.2. Funktio, jolle annetaan parametreja

```
# 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.")
    return None

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 siten, että a:n arvo siirtyy muuttujaan `nimi`, b:n arvo muuttujaan `osasto` ja c:n arvo muuttujaan `vuosikurssi` niiden järjestyksen mukaisesti.

Funktion päättyminen ja paluuarvo

Funktion suoritus päättyy jossain vaiheessa, jonka jälkeen ohjelman suoritus jatkuu funktiokutsun jälkeisestä toiminnosta ja funktion kohdalla tähän liittyy paluuarvon käsittely. Pythonissa funktiosta paluuta varten on varattu sana `return`, johon päätyminen lopettaa funktion suorittamisen samalla tavoin kuin `break` keskeyttää toistorakenteen. `return` - käskyn kanssa voidaan myös määritellä paluuarvo, joka palautetaan funktiokutsun paikalle. Jos `return`-käskylle ei anneta mitään palautusarvoa, Python lisää käskyyn tiedon paluuarvon puuttumisesta varatulla sanalla `None`. Ja mikäli ohjelmoija jättää koko `return`-käskyn laittamatta, lisää Python aliohjelman loppuun automaattisesti käskyn `return None`. Tyypillisesti funktio kuitenkin palauttaa arvon ja se voidaan tehdä vakiolla tyyliin ”`return 5`” tai käyttäen muuttujaa. Tämäkin asia on helpoin ymmärtää esimerkin avulla.

Esimerkki 5.3. Paluuarvo käytännössä

```
# 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 kutsussaan kaksi parametria ja vertailee niitä keskenään yksinkertaisella `if-else`-rakenteella. Tämän jälkeen funktio palauttaa kutsun tehneelle lauseelle suuremman arvon `return` –käskyllä. Huomaa, että funktion palauttama arvo sijoitetaan muuttujan arvoksi eli otetaan talteen myöhempää käyttöä varten.

Esimerkki 5.4. Paluuarvo käytännössä, osa 2

```
# 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 lopputulosta `plakaatti` nimiseen muuttujaan. Ensimmäiseksi siihen laitetaan ”*”-merkkiä käyttäjän syötteen pituuden verran + 4

kappaletta ja rivinvaihto. Seuraavalla rivillä `plakaatti`in 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 `plakaattin`.

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

Tämän jälkeen 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 *
*********

>>>
```

Nimiavaruus

Nimiavaruus on asia, johon tulet törmäämään funktioiden kanssa. 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 eri funktioiden välillä muuttujilla ei ole minkäänlaista yhteyttä. Tämän vuoksi funktionsisäisiä muuttujia sanotaan paikallisiksi tai lokaaleiksi muuttujiksi. Ne näkyvät ainoastaan oman funktionsa sisällä.

Esimerkki 5.5. Paikallisten muuttujien toiminta

```
# Tiedosto: funktio_lokaali.py

def funktio():
    x = 2
    print("x funktion sisällä", x)
    return None

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

Tuloste

```
>>>
x on ennen funktiota 50
x funktion sisällä 2
x on funktion kutsumisen jälkeen edelleen 50
>>>
```

Kuinka se toimii

Funktion sisällä näemme, kuinka paikallisen muuttujan `x` arvoksi määritellään 2. Tämä ei kuitenkaan vaikuta päätason muuttujan `x` arvoon, koska funktion muuttuja `x` ja päätason muuttuja `x` ovat ainoastaan samannimisiä, mutta eri nimiavaruudessa olevia muuttujia. Ajattele asiaa vaikka siten, että funktio toimii omalla planeetallaan eikä siellä tapahtuvat asiat heijastu muille planeetoille suoraan. Koska funktioiden on kommunikoitava keskenään, on siihen oltava toimivat mekanismit ja ne ovat tiedon vieminen funktioon *parametreina* sekä tiedon palauttaminen aliohjelmasta *paluuarvon* avulla kuten edellä käytiin läpi.

Tunnus eli muuttuja, aliohjelma tai kiintoarvo

Luvussa 1 oli puhetta, että muuttujien nimet ovat tunnuksia ja tunnusten nimeämiselle oli sääntöjä. Myös aliohjelmien nimet ovat tunnuksia ja niiden nimeämistä koskevat samat säännöt kuin muuttujia. Monissa muissa ohjelmointikielissä on muuttujien lisäksi olemassa vakio-tunnuksia, joiden arvoa ei voi muuttaa ja ohjelmointiympäristö varmistaa, ettei niitä muuteta. Pythonissa tällaisia vakioita ei ole. Vakioilla voidaan kuitenkin lisätä ohjelman ymmärrettävyyttä ja ylläpidettävyyttä merkittävästi, sillä niiden avulla voidaan esimerkiksi määritellä merkkijonon pituudelle maksimipituus `MAX_PITUUS = 80` tai minimi ikäraja `IKA = 18`, jotka ovat käytössä kaikkialla ohjelmassa muuttumattomina. Näin ollen Pythonissa tämä asia pitää hoitaa luvussa 1 mainitun muuttujan roolin *kiintoarvo* avulla. Käytännössä tämä tarkoittaa sitä, että ohjelmoijan on oltava tarkkana ja katsottava, ettei kiintoarvon arvo muutu ohjelman suorituksen aikana. Koska vakio-konsepti on keskeinen ohjelmointiin liittyvä peruskonsepti, tullaan tässä oppaassa käyttämään kiintoarvoja, joka siis tarkoittaa sitä, että (1) lähtökohtaisesti kiintoarvot näkyvät kaikkialla ohjelmassa, (2) Pythonissa ohjelmoija vastaa siitä, ettei kiintoarvon arvo muutu ohjelman aikana ja (3) muissa ohjelmointikielissä käytettäisiin tyypillisesti vakio-tunnuksia.

Pääohjelma eli `paaohjelma()`

Python-ohjelmoinnin vahvuuksia on sen keveys ja ketteryys, joiden pohjalta minimaalinen ohjelma voi olla yksi `print`-lause. Tämä mahdollistaa Pythonin opiskelun tehokkaasti vaiheistettuna ja myös höydyllisten ohjelmien tekemisen minimaalisella koodimäärällä. Koodimäärän kasvaessa ohjelmien luettavuus, selkeys ja ylläpidettävyys tulevat tärkeämmäksi kuin nopea toteutus ja kun ohjelmointikieli ei pakota näitä ominaisuuksia, jää niiden toteutus ohjelmoijan vastuulle (vrt. vakio-kiintoarvo –keskustelu yllä). Tärkeimpiä ominaisuuksia luettavuuden ja ymmärrettävyyden kannalta on selkeä rakenne, jota tavoitellessa keskeinen asia on jakaa ohjelma pieniin selkeisiin kokonaisuuksiin eli aliohjelmiin. Kun aliohjelman toimintaperiaatteet ymmärtää, ei sen sisäisestä toteutuksesta tarvitse välittää vaan voi keskittyä sen käyttämiseen niin kuin tilanne on esimerkiksi funktioiden `print`, `input` ja `len` kohdalla.

Yksi Pythonin keino nopeuttaa ohjelmien tekemistä on tulkita päätasolla oleva koodi suoraan monissa muissa ohjelmointikielissä olevana pääohjelma-rakenteena. Esimerkiksi C-kielessä on `main()`-ohjelma, joka on ainoa pakollinen funktio ohjelmassa ja jota käyttöjärjestelmä kutsuu ohjelmaa käynnistettäessä. Pythonin ratkaisu toimii hyvin niin kauan kuin päätasolla oleva koodin määrä vastaa tyypillistä funktiota eli kun se mahtuu yhdellä kertaa näytölle. Tätä isompien ohjelmien kohdalla riski virheisiin kasvaa ja siksi suositaan lyhyitä ohjelmia.

Nimiavaruuksien kannalta Pythonin päätasolla oleva koodi on riski, sillä päätasolle kirjoitettu koodi näkyy kaikkialle tiedostossa ja esim. päätasolla olevien muuttujien arvoja voi muuttaa myös aliohjelmien sisällä. Siksi ohjelmien koon ja ylläpidettävyysvaatimusten

kasvaessa päätasolla oleva koodi kannattaa minimoida ja laittaa kaikki ohjelmakoodi aliohjelmien sisälle erillisiin nimiavaruuksiin.

Hyvän ohjelmointityylin mukaisessa isommassa Python-ohjelmassa kaikki koodi kirjoitetaan siis erillisiin aliohjelmiin ja päätasolla on vain kutsu pääohjelmaksi nimettyyn funktioon. Pääohjelman nimeksi suositellaan tässä oppaassa yksinkertaisuuden vuoksi käyttämään aina `paaohjelma()`-nimeä C-kielen `main`-ohjelman mukaisesti. Päätasolla on `paaohjelma()`-kutsun lisäksi kaikkien aliohjelmien määrittelyt (`def …():`) sekä kiintoarvojen määrittelyt sekä muita myöhemmin oppaassa mainittavia globaalisti määriteltäviä rakenteita.

Alla on esimerkki 5.6 hyvän ohjelmointityylin mukaisesta minimaalisesta ohjelmatiedoston ja ohjelman rakenteesta. Huomaa, että merkittävä osa ohjelmasta liittyy aliohjelmien määrittelyihin ja kirjoittamalla vain itse toiminnallisuuden päätasolle ohjelma olisi lyhempi ja nopeampi tehdä. Siksi tämä ohjelmarakenne sopii isompiin, kasvaviin ohjelmiin, joissa halutaan varmistaa ohjelman ymmärrettävyys ja ylläpidettävyys muutoksista ja pitkästä käyttöiästä huolimatta. Näin ollen tämänkin oppaan tulevissa esimerkeissä palataan tyypillisesti lyhyeen päätasolle kirjoitettavaan koodiin, jotta esimerkeissä voidaan keskittyä uuteen asiaan ja tarpeen tullen tämän isoon ohjelmaan sopivan ohjelmarakenteen voi tarkistaa täältä.

Esimerkki 5.6. Ison ohjelman rakenne

```
# Tiedosto: ohjelma_rakenne.py

def tulosta(lkm): # Aliohjelma(t), ennen pääohjelmaa
    print(lkm)
    return None

def paaohjelma():
    luku = 1
    tulosta(luku)
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
# eof
```

Tunnusten näkyvyys

Tässä oppaassa lähtökohtana on, että muuttujat ovat aina paikallisia omassa nimiavaruudessaan yhden funktion sisällä niiden hallitsemattoman muuttumisen estämiseksi. Toisaalta aliohjelmat tulee aina tehdä päätasolle siten, että ne näkyvät ja ovat käytettävissä kaikkialla tehdyssä ohjelmassa. Kiintoarvot, eli vakiot, voivat olla paikallisia nimiavaruuteen liittyen, mutta lähtökohtaisesti niistä on eniten hyötyä silloin kun ne näkyvät kaikissa aliohjelmissa ja sama tieto on käytettävissä kaikkialla yhdessä ohjelmassa eli kun ne on määritelty ohjelman päätasolla globaaleina kiintoarvoina. Ja tästä seuraa myös, että tässä oppaassa **globaalit eli yhteiset muuttujat, ovat kiellettyjä** – ellei niitä erikseen käsketä käyttämään. Eräissä poikkeustapauksissa näin joudutaan tekemään oppaan lopussa, mutta silloin asia mainitaan erikseen. Alla olevassa taulukossa näkyy tässä oppaassa tähän mennessä esitellyt tunnukset sekä niille suositeltava näkyvyys ohjelmassa.

Taulukko 5.1. Tunnusten näkyvyys

Tunnus	Näkyvyys
Muuttuja	Paikallinen
Kiintoarvo (Vakio)	Paikallinen tai globaali
Aliohjelma	Globaali

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."""
     uusin = 0
     fib_luku1 = 0
     fib_luku2 = 1
     for i in range(0, maara):
         uusin = fib_luku1 + fib_luku2
         print(uusin, end = " ")
         fib_luku2 = fib_luku1
         fib_luku1 = uusin

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

Huomaat varmaan, että funktion ensimmäiselle riville on ilmestynyt kolmen lainausmerkin notaatiolla tehty merkkirivi. Tämä ei aiheuta tulkissa virhettä, 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 aukeavassa apuruudussa funktiokutsun mallin niin kuin kirjoitit sen itse määrittelyyn, sekä sen alapuolella kirjoittamasi dokumentaatiorivin. Rivin käyttö ei ole pakollista, mutta se helpottaa funktioiden käyttöä ja on hyvän ohjelmointitavan mukaista.

Dokumentaatiorivi näkyy myös IDLEn interaktiivisessa 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 valmiina tarjoamia funktioita

Tähän mennessä olemme tutustuneet jo muutamiin Pythonin tarjoamiin funktioihin, 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 parametrina olevan luvun itseisarvon (absolute value).

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

max(*x*)

`max` palauttaa parametrina olevan joukon suurimman alkion. `max` voi ottaa parametrinaan myös listan, johon tutustutaan luvussa 7.

```
>>> 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 pyöristettävä luku 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 parametrina olevan listan alkioiden summan (toimii vain numeroilla). Valinnainen `start` parametri kertoo, mistä alkiosta laskeminen lähtee liikkeelle. Oletuksena on alku eli indeksi 0.

```
>>> sum(range(1,6))
15
>>> 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'>
```

Osaamistavoitteet

Ymmärrät termit pääohjelma, aliohjelma ja funktio. Osaat luoda funktion ja tiedät kuinka sille välitetään parametreja sekä miten funktio palauttaa arvon. Ymmärrät nimiavaruuskäsitteen ja miten se liittyy tunnusten näkyvyyteen, aliohjelmiin ja paaohjelma() –konseptiin. Osaat käyttää funktion dokumentaatioriviä, muutamia Pythonin yleisesti käytettyjä funktioita ja tehdä kokoavassa esimerkissä 5.7 olevaa ideaa noudattavia valikkopohjaisia Python-ohjelmia.

Luvun asiat kokoava esimerkki

Esimerkki 5.7. Valikko-pohjainen ohjelma

```
# Tiedosto: valikkopohjainen_ohjelma.py

def valikko():
    print("Käytettävissä olevat toiminnot:")
    print("1) Tulosta 'Moi'")
    print("2) Tulosta 'Hoi'")
    print("0) Lopeta")
    valinta = int(input("Valintasi: "))
    return valinta

def tulosta(sana):
    print(sana)
    return None

def paaohjelma():
    while (True):
        toiminto = valikko()
        if (toiminto == 1):
            tulosta("Moi")
        elif (toiminto == 2):
            tulosta("Hoi")
        elif (toiminto == 0):
            print("Kiitos ohjelman käytöstä.")
            break
        else:
            print("Valintaa ei tunnistettu, yritä uudestaan.")
        print()
    return None

paaohjelma()
# eof
```

Luku 6: Tiedoston käsittely

Usein kohtaamme tilanteen, jossa haluamme tallentaa muuttujien arvot koneelle myöhempää käyttöä varten, jotta ohjelman ja koneen voi sammuttaa välillä. Tällöin voisimme lukea muuttujien arvot suoraan koneelta ilman, että joudumme joka kerta aloittamaan ohjelman kirjoittamalla muuttujien tiedot koneelle. Tällaisten tilanteiden varalta Python tukee tiedostoihin kirjoittamista sekä niistä lukemista, joista puhumme tässä luvussa.

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

Tiedostojen kanssa työskentely

Käsiteltäessä tiedostoja olisi hyvä tietää, mistä niitä löytyy. Jos teemme töitä IDLEn editorilla tai interaktiivisella ikkunalla, on oletuskansio se, mihin lähdekoodi on tallennettu. Nyt, kun tiedämme mihin tekemämme muutokset tallentuvat, voimme avata tiedoston, kirjoittaa sinne jotain ja lukea aikaansaannoksemme:

Esimerkki 6.1. Tiedoston avaaminen, kirjoittaminen ja lukeminen

```
# Tiedosto: tiedtesti.py

# Määritellään useita merkkijonoja
rivi1 = "Merkkijono."
rivi2 = "Kokonaislukuja: 1 2 3 4."
rivi3 = "Liukuluku: 2.345."

# Tekstitiedostoon kirjoittaminen
tiedosto = open("data.txt", "w") # avataan kirjoitusta varten
tiedosto.write(rivi1 + '\n') # kirjoitetaan rivi1 + rivinvaihtomerkki
tiedosto.write(rivi2 + '\n') # kirjoitetaan rivi2 + rivinvaihtomerkki
tiedosto.write(rivi3 + '\n') # kirjoitetaan rivi3 + rivinvaihtomerkki
tiedosto.close() # suljetaan tiedosto

# Tekstitiedostosta lukeminen
tiedosto = open("data.txt", "r") # avataan tiedosto lukemista varten
while (True):
    rivi = tiedosto.readline() # Luetaan tiedostosta yksi rivi
    if len(rivi) == 0: # Jos rivin pituus on 0, ollaan lopussa
      break
    print(rivi, end = "") # Tulostetaan rivi, ei lisätä rivinvaihtoa
tiedosto.close() # suljetaan tiedosto
```

Tuloste

```
>>>
Merkkijono.
Kokonaislukuja: 1 2 3 4.
Liukuluku: 2.345.
>>>
```

Kuinka se toimii

Tällä kertaa esimerkki on jaettu kommenteilla kolmeen osaan, joista ensimmäinen sisältää tiedostoon tallennettavat merkkijonot, toinen kirjoittaa nuo merkkijonot tiedostoon ja kolmas lukee tiedoston sekä tulostaa luetut merkkijonot näytölle. Käsiteltävissä

merkkijonoissa ei ole mitään uutta, mutta on syytä huomata, että tekstitiedostossa kaikki tieto on tekstinä eli merkkijonoina – myös kokonaislukuja ja liukulukuja käsitellään merkkijonoina.

Tiedoston kirjoittaminen muodostuu kolmesta osasta: tiedoston avaaminen, merkkijonojen kirjoittaminen tiedostoon sekä tiedoston sulkeminen. **Tiedoston avaaminen** tapahtuu rivillä `tiedosto = open("data.txt", "w")`, jossa `open`-käskyllä avataan `data.txt` niminen tekstitiedosto kirjoittamista varten eli käskyn ensimmäinen parametri on avattavan tiedoston nimi ja toinen on sen tila ja avauskäskyn paluuarvo otetaan talteen `tiedosto`-muuttujaa eli tiedostokahvaan, jolla avattua tiedostoa voidaan käyttää jatkossa. Avattavan tiedoston nimen on oltava merkkijono (tai merkkijonon sisältävä muuttuja) ja samoin avaustila on merkkijono. Avaustila tai moodi voi olla kirjoittaminen eli 'w' (write), lukeminen eli 'r' (read) tai lisäys eli 'a' (append), joka mahdollistaa vastaavien operaatioiden tekemisen tiedostolle eli lukemisen tai kirjoittamisen. Kirjoittamisen yhteydessä on hyvä muistaa, että w-moodi luo uuden tiedoston ja mahdollisesti aiempi samanniminen tiedosto tietoineen häviää samalla hetkellä. Siksi vaihtoehto kirjoittamiselle on lisäys, jolloin olemassa oleva tiedosto säilyy ja kirjoitusoperaatiot lisäävät tiedoston loppuun uusia merkkejä/rivejä. Ja koska nyt suoritettava ohjelma on tiedostossa `tiedtesti.py`, kirjoitetaan uusi tiedosto samaan hakemistoon tämän tiedoston kanssa.

Varsinainen **tiedostoon kirjoittaminen** tapahtuu käskyllä `tiedosto.write()`, joka saa parametrina kirjoitettavan merkkijonon. Tässä kannattaa huomioida seuraavat asiat:

1. Tiedostokahva `tiedosto` ei päästä meitä käsiksi tiedoston sisältöön vaan se tarjoaa "kulkuyhteyden" tiedoston sisälle. Tiedostoa itsessään manipuloidaan kahvan kautta, ja siksi funktiot kuten `write` tai `close` merkataan pistenotaation avulla, koska ne ovat tiedostokahvan jäsenfunktioita. Tässä vaiheessa tärkeintä on käyttää muuttujanimeä ja funktionimeä sekä lisätä piste niiden väliin.
2. Kannattaa muistaa, että olemme nyt avanneet tiedoston kirjoittamista varten. Nyt voimme kirjoittaa mitä haluamme, mutta jos tila olisi valittu toisin – kuten esimerkiksi lukutila – kirjoitusyritys aiheuttaisi virheilmoituksen.
3. Tekstitiedostoon voi kirjoittaa vain merkkijonoja. Tämä tulee ottaa kirjaimellisesti; jos haluat tallentaa lukuarvoja tiedostoihin, pitää ne ennen kirjoittamista muuttaa merkkijonoiksi. Tämän voi tehdä esim. funktioilla `str()` ja `round()`.
4. `write` -käsky poikkeaa `print` –käskystä kahdella tavalla eli se ottaa parametrikseen vain yhden merkkijonon, ei siis pilkuilla eroteltuja parametreja, ja se ei lisää mitään merkkijonoon. Käytännössä tämä tarkoittaa sitä, ettei `write`-kirjoita merkkijonon loppuun rivinvaihtomerkkiä niin kuin `print` –tekee.

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

Esimerkin kolmas osio on *tiedoston lukeminen*. Ohjelma avaa tiedoston uudelleen ja tällä kertaa lukutilaan moodilla 'r' (read), joka on myös Pythonin oletusarvo tälle parametrille. 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 loimme juuri tiedoston, voimme olla varmoja sen olemassa olosta ja ohjelma jatkaa itse lukuvaiheeseen.

Luemme tiedoston yksi rivi kerrallaan käskyllä `tiedosto.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 on 0, ja tällöin tiedämme, että voimme lopettaa lukemisen ja poistua toistorakenteesta. Mikäli readline-palautti rivin, tulostamme sen print-käskyllä, mutta lisäämättä print:n rivinvaihtomerkkiä tiedostosta luetun rivinvaihtomerkin perään. Lopuksi suljemme taas tiedoston sen käytön jälkeen, jonka jälkeen ohjelman suoritus on päättynyt, ja voimme tulostuksesta todeta, että kirjoittaminen ja lukeminen onnistui suunnitellulla tavalla.

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

Avaustiloista

Kuten edellisestä esimerkistä huomasit, on tiedostoa avattaessa määriteltävä sen tila (tai moodi). Pythonissa on sisäänrakennettuna useita erilaisia tiloja, joista yleisimmät eli kirjoittaminen ja lukeminen käsiteltiin yllä ja myös lisäys eli 'a'/append tuli mainittua.

Pythonista löytyy myös luku- ja kirjoitustila `'r+'`, johon avattaessa tiedostoon voidaan sekä kirjoittaa että lukea tietoa. Tällöin tulee olla tarkkana luku- ja kirjoituspaikan eli kirjanmerkin kanssa, jottei tiedostorakenne mene sekaisin hallitsemattomien kirjoitus- ja lukuoperaatioiden seurauksena. Lähtökohtaisesti ohjelma tulee suunnitella niin, että yhdessä paikassa kirjoitetaan tiedostoon ja toisessa luetaan sieltä eikä sekoiteta näitä.

Pythonista löytyy myös muita tiloja mm. binäärisen datan käsittelylle. Näihin palataan luvussa 12. Lisää tietoa avaustiloista löytyy kiinnostuneille myös esim. Python Software Foundationin dokumenteista.

Kirjanmerkistä

Huomasit varmaan, että yllä olevissa kappaleissa puhuttiin mystisestä kirjanmerkistä. Käytännössä kirjanmerkki on tiedostokahvassa oleva tieto siitä, missä kohdin tiedostoa olemme etenemässä. Tästä voi myös päätellä, ettei tiedostokahva ole yksinkertainen muuttuja kokonaisluvun tyyliin vaan se sisältää useita erilaisia tietoja. Näihin rakenteisiin muuttujiin palataan tarkemmin myöhemmissä luvuissa.

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 edellinen kirjoitusoperaatio päättyi. Esimerkiksi lisäystila `'a'` siirtää kirjanmerkin automaattisesti tiedoston loppuun. Kirjanmerkkiä voi myös siirtää käsin funktiolla `fseek`, josta enemmän myöhemmin tässä luvussa.

Erilaisia merkkien koodaustapoja

Esimerkissä 6.1 käytimme tiedoston avaamiseen käskyä `open("data.txt", "r")` ja se toimi hyvin. `open`-käskyllä on myös kolmas parametri, `encoding`, jolla voidaan kertoa käytetty merkistö. Lähtökohtaisesti sitä ei tarvita, mutta jos merkistö – tyypillisesti ääkköset – eivät toimi oikein, käytetyn merkistön kertominen ohjelmalle voi poistaa ongelmia. Pythonin oletusarvo tälle parametrille toimii useimmiten, mutta olettamuksiin luottaminen

johtaa usein jossain vaiheessa ongelmiin. Merkistön koodauksen voi määritellä aina tiedostoa avattaessa avaustilasta riippumatta, ja tällöin kannattaa käyttää universaalia UTF-8 –koodausta. Merkistöt käsitellään tarkemmin luvussa 12, mutta merkistön määrittely tiedostoa avattaessa tehdään seuraavalla tavalla:

```
tiedosto = open("data.txt", "r", encoding="utf-8")
```

Tiedostonkäsittelyyn liittyviä metodeja

Tiedoston lukemiseen on useita erilaisia tapoja. Yksinkertaisin ja samalla avoimin niistä on **`read`**`(koko)`, joka lukee tiedostoa ja palauttaa sen sisällön merkkijonona. Parametria *koko* voidaan käyttää, jos halutaan lukea tietyn kokoisia viipaleita – koko annetaan kokonaislukuna, 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 tietoa. 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 suorittamalla funktio yhden kerran saa 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 tiedostokahva `tiedosto`, voit tulostaa sen sisällön helposti seuraavasti:

```
>>> 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)
```

Tiedostoa luettaessa merkkijonot on tyypillisesti muunnettava tietoalkioiksi, joten luettaessa merkkijonoja pilkotaan pienempiin osiin. Etenkin taulukkolaskentaohjelmista pystyy tallettamaan tietoja tekstitiedostoon ja silloin taulukon peruselementit, rivi ja sarake, pyritään toistamaan vastaavalla tavalla. Käytännössä rivien erottaminen käy helpolla rivinvaihtomerkillä ja sarakkeiden erottelu tehdään vastaavasti sovitulla merkillä. Sarake-erotin ei ole yhtä selkeästi sovittu kuin rivinvaihtomerkki, vaan sarake-erottimena käytetään usein tabulaattoria (\t), pilkkua (,) ja puolipistettä (;). Tyypillinen taulukkolaskentaohjelman tallentama tiedosto voisi näyttää alla olevalta, jossa ensimmäisellä rivillä on sarakkeiden otsikot ja seuraavilla riveillä itse tietoalkiot:

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

Tällainen tieto jaetaan lukemisen jälkeen Pythonilla tyypillisesti **`split`**-metodin avulla antamalla kenttäerotin parametrina. split palauttaa lista-muuttujan, joka esitellään seuraavassa luvussa, mutta tässä vaiheessa eri sarakkeiden arvot voidaan tulostaa alla olevan esimerkin avulla. Esimerkin kenttäerotin on puolipiste (;) eikä pilkku (,), koska Suomessa kenttäerotinpilkku sekoittuu helpolla desimaalipilkkuun.

```
rivi = "Ville;Vallaton;0404567890"
tiedot = rivi.split(';')
print(tiedot)
for alkio in tiedot:
    print(alkio)
```

Aiemmin mainittuun kirjanmerkkiin liittyy mm. **`tell`**–funktio, joka palauttaa kokonaislukuarvona tiedon siitä, kuinka monta merkkiä tiedoston alusta on tultu, eli siis kuinka monta merkkiä tiedostosta on luettu. Näin saadaan siis selville kirjanmerkin tämänhetkinen 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 ja *`mista_laskettuna`* **toimii vain binääritiedostoille**, joihin palataan oppaan luvussa 12.

```
>>> f = open("/tmp/workfile", "r+", encoding="utf-8")
>>> f.write("0123456789abcdef")
>>> f.seek(5)     # Mene tiedoston 6. merkkiin
>>> f.read(1)
'5'
>>> f.seek(10)    # Mene tiedoston 10. merkkiin
>>> f.read(2)
'ab'
```

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 "<pyshell#37>", line 1, in <module>
    f.read()
ValueError: I/O operation on closed file.
```

Tiedostojen käsittelyyn on myös muita työkaluja, mutta niiden käyttö on edellä esiteltyihin nähden marginaalista. Kaikki tiedostonkäsittelyyn liittyvät työkalut on esitelty kattavasti Python Software Foundationin dokumentaation luvussa, joka käsittelee I/O-operaatioita (Input/Output, syöttö/tulostus).

Usean tiedoston käyttö samaan aikaan

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

Esimerkki 6.2. Tiedoston avaaminen, kirjoittaminen ja lukeminen

```
# 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 otetaan käyttöön 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.

Muotoiltu tulostus

Python-ohjelmointikielessä on toimintoja merkkijonojen aiempaa tarkempaan muotoiluun. Tätä sanotaan formatoiduksi eli muotoilluksi tulostukseksi ja tässä käytämme hyväksi merkkijonoille tarjolla olevaa jäsenfunktiota `format`. Joskus on nimittäin tulostettava merkkijono tietyn kaavan mukaisesti ja halutaan tulostaa esimerkiksi ”nelinumeroinen luku” tai ”kaksi-desimaalinen liukuluku”. Teknisesti tämä toteutetaan siten, että tulostettavaan merkkijonoon laitetaan haluttuihin kohtiin muotoilumerkit `{}`-merkeillä erotettuna, ja syötteen perään sijoitetaan tulostettavat arvot `.format(muuttujalista)`-muodossa. `format` on siis merkkijono-muuttujan metodi, joka muokkaa merkkijonoa laittamalla haluttuihin kohtiin muuttujalistan arvot määritellyssä muodossa ja lopputulos on merkkijono, jonka `print`-lause tulostaa.

`print`-funktiolle voidaan antaa useita parametreja tuttuun tyyliin seuraavasti:

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

`format`-metodin kanssa sama asia tehdään hieman eri tavalla (tilapäissäilön s käyttö korostaa merkkijonon muokkaamista):

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

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

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

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 mukaan. Aiemman muuttujan indeksin lisäksi nyt on määriteltävä luvun tyypiksi desimaaliluku eli float/f, ja haluttu kentän leveys pisteen ja f:n välissä:

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

Tämä antaa tuloksena seuraavan tulosteen:

```
13 13.3 13.27 13.27334
```

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

Lisäämällä 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.

Edellä olevissa `format`-esimerkissä muotoilu kohdistuu siis {} –sulkujen kohdalle ja ensimmäinen numero ennen kaksoispistettä, on tulostettavan muuttujan indeksi. Tulostettavat muuttujat ovat `format`-käskyn perässä olevissa suluissa, yllä vain `luku`, ja siksi edellä kaikki tulostettavat arvot ovat muuttujan indeksillä 0. Muuttujan indeksin jälkeen voi olla osio ”`:m.n`”, jossa : ja . ovat kenttäerottomia kun taas m kertoo koko tulosteen leveyden merkkeinä ja n desimaalien määrän.

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. Tulostuksen muotoilukoodin vaihtoehdot näkyvät Taulukossa 6.1.

Taulukko 6.1 Muotoillun tulostuksen muotoilukoodit

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

Merkkijonojen jäsenfunktiot

Tekstitiedostojen käsittely edellyttää yleensä intensiivistä työskentelyä merkkijonojen ja niiden muotoilun kanssa, kuten edellä olleesta *Muotoiltu tulostus* –osuudesta kävi ilmi. Aiemmin esiteltyjen merkkijonojen yhdistelyn (+ ja * -operaattorit) sekä leikkausten ([::]) lisäksi Pythonissa on edellä esitelty `format`-metodi ja monta muuta metodia merkkijonojen

muokkaamiseen.

Metodi eli jäsenfunktio tarkoittaa tiettyyn muuttujatyyppiin liittyviä funktioita. Palaamme tähän asiaan luokkien ja olioiden yhteydessä eli jäsenfunktiot liittyvät aina luokkaan, mutta nyt riittää muistaa, että tiedostokahvaan liittyviä tiedostoja voidaan käsitellä tiedostometodeilla ja merkkijonomuuttujia voidaan muokata merkkijonometodeilla.

Yleisimmät Pythonin merkkijonometodeista ovat seuraavat:

capitalize()

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

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

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

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

Sama kun endswith() mutta testaa merkkijonon alkua.

expandtabs([*tabsize*])

Palauttaa merkkijonosta kopion, jossa kaikki sisennysmerkit (välilyönti ja sarkainmerkki(tab)) on korvattu tabsize –määrällä välilyöntejä. `Tabsize` on oletusarvoisesti 8. (Huom! IDLEssä 4)

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

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

format(*parametrilista***)**

Muotoilee parametrilistassa olevat arvot ja sijoittaa ne merkkijonoon. Ks. tarkemmin yllä *Muotoiltu tulostus* –kohta.

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

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

isalnum()

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

isalpha()

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

isdigit()

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

islower()

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

isupper()

Sama kuin `islower()`, mutta isoille kirjaimille.

isspace()

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

lower()

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

lstrip([chars])

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

```
>>> '   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 alkiota).

```
'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. Itse lista-asiaan palataan myöhemmin.

upper()

Muuttaa kaikki merkkijonon kirjaimet isoiksi kirjaimiksi.

Esimerkki 6.3. Merkkijonojen muokkailua

```
# 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 demonstroida, 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.

Osaamistavoitteet

Ymmärrät tiedostojen lukemisen ja kirjoittamisen. Hahmotat miten eri avaustilat vaikuttavat tiedostojen käsittelyyn ja tunnet yleisemmät työkalut tiedostojen käsittelyyn. Ymmärrät muotoillun tulostuksen sekä osaat käyttää sitä ja osaat myös käsitellä merkkijonoja niiden metodien kautta.

Luvun asiat kokoava esimerkki

Esimerkki 6.4. Tiedostojen käsittelyä

```
# (c) LUT 2019 L06KokoavaEsimerkki.py un
# Tämä esimerkki on tarkoitettu omatoimisen oppimisen tueksi ohjelmoinnin
# opiskeluun. Muu käyttö kielletty.
########################################################################

def valikko():
    print("Mitä haluat tehdä:")
    print("1) Tallenna tiedosto")
    print("2) Lue tiedosto")
    print("0) Lopeta")
    valinta = int(input("Valintasi: "))
    return valinta

def tallenna(nimi):
    tiedosto = open(nimi, 'w', encoding="utf-8")
    tiedosto.write("Ville;178\n")
    tiedosto.write("Kalle;192\n")
    tiedosto.close()
    return None

def lue(nimi):
    tiedosto = open(nimi, 'r', encoding="utf-8")
    while (True):
        rivi = tiedosto.readline()
        if (len(rivi) == 0):
            break
        rivi = rivi[:-1] # rivinvaihtomerkki pois
        sarake = rivi.split(';')
        print("Tiedostossa oli '"+sarake[0]+"' ja '"+sarake[1]+"'.")
    tiedosto.close()
    return None

def paaohjelma():
    tiedosto = "L06Testi.txt"
    while (True):
        toiminto = valikko()
        if (toiminto == 1):
            tallenna(tiedosto)
        elif (toiminto == 2):
            lue(tiedosto)
        elif (toiminto == 0):
            print("Kiitos ohjelman käytöstä.")
            break
        else:
            print("Valintaa ei tunnistettu, yritä uudestaan.")
        print()
    return None

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

Luku 7: Rakenteiset tietorakenteet

Tietorakenteet ovat yksinkertaisesti sanottuna juurikin itseään tarkoittava asia – eli ne ovat rakenteita, jotka sisältävät tietoa. Olemme tähän mennessä käyttäneet Pythonin ”yksinkertaisia” tietorakenteita kokonaisluku ja liukuluku, joissa muuttuja viittaa suoraan koko tietoalkioon ja ohjelmoijan ei tarvitse tietää tai välittää sen sisäisestä rakenteesta. Olemme myös käyttäneet merkkijonoa, jossa muuttujan nimi viittaa lukujen tavoin koko merkkijonoon, mutta lisäksi voidaan viitata muuttujan osiin eli merkkeihin yksitellen ja siitä voidaan erottaa haluttaessa leikkaus eli osa jatkokäsittelyyn.

Pythonissa on myös tietorakenteita, joissa on ohjelmoijan tiedossa oleva sisäinen rakenne eli ne sisältävät useita tietoalkioita, jotka liittyvät toisiinsa ja ovat täten osa samaa kokonaisuutta. Näistä rakenteisista tietorakenteista yleisimpiä ovat lista, luokka/olio, tuple ja sanakirja. Tässä luvussa keskitymme listaan ja luokkaan, mutta esittelemme lyhyesti myös listaa muistuttavan tuplen. Sanakirja on läheistä sukua listalle ja palaamme siihen luvussa 10. Tämän luvun päätteeksi katsomme kokoavassa esimerkissä ohjelmaa, jossa on toteutettu tyypillisen perusohjelman käyttöliittymä ja tiedostonkäsittely käyttäen hyväksi ajonaikaisia tietorakenteita lista ja luokka sekä tiedon pysyvään tallennukseen tekstitiedostoa.

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 listan asioista, esim. ostoslistan, niin todennäköisesti listaat asiat allekkain. Python-listan idea on sama ja erilaisesta teknisestä toteutuksesta huolimatta lopputulos 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. Toisin sanoen listaa voidaan muokata vapaasti toisin kuin esimerkiksi merkkijonoja. Lisäksi listan käsittelyyn on olemassa tehokkaita apuvälineitä, jäsenfunktioita eli metodeja, joita käymme kohta lävitse.

Tämä saattoi kuulostaa vähän sekavalle, mutta ei hätää, esimerkki selkeyttää taas asiaa kummasti.

Esimerkki 7.1. Listan käyttö

```
# 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, joka sisältää hankittavia tuotteita. Listan alkioina on merkkijonoja eli ostettavien tuotteiden nimet. Periaatteessa tämä ei perustu mihinkään rajoitteeseen; listan alkioiksi voi tallentaa 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.*

Huomaa myös, että käytimme hieman aiemmasta poikkeavaa `for...in`-toistorakennetta listan läpikäymiseen. Tämä 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, heittomerkit sekä pilkut jäävät alkioiden väliin.

Tämän jälkeen suoritamme listan järjestämisen 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, joihin palaamme myöhemmin. Kannattaa myös huomata, että tässä tapauksessa järjestettyä listaa ei tarvitse erikseen tallentaa uuteen muuttujaan. Kaikki metodit vaikuttavat suoraan käsiteltävään listaan, 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 alkionsisäiset leikkaukset suoritetaan ensin valitsemalla alkio, ja tämän jälkeen sille tehtävä leikkaus. Ensimmäisen alkion viisi ensimmäistä merkkiä saadaan siis kirjoittamalla `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 7.2. Listan leikkaukset

```
# Tiedosto: lista_l.py
# Ostoslistan märittely 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 tule 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
```

Listan tyhjennys kannattaa tehdä clear()-metodilla, jolloin muuttujan käyttöä voi jatkaa:

```
>>> lista = [-1, 1, 66.25, 333, 333, 1234.5]
>>> lista.clear()
>>> lista.append(123)
>>> lista
[123]
```

Yleisimpiä listan jäsenfunktioita

Listan manipulointiin on erilaisia tapoja. Seuraavassa esitellään yleisiä jäsenfunktioita listan sisällön muokkaamiseen.

append(*x*)

Lisää alkio x listan loppuun.

clear()

Tyhjentää listan, mutta jättää muuttujan käyttöön.

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 tapahtuisi käskyllä `a.insert(0, x)`, ja loppuun lisääminen – samoin kuin `append` tekee – tapahtuisi käskyllä `a.insert(len(a), x)`.

remove(*x*)

Poistaa listalta ensimmäisen alkion, jonka arvo on `x`, eli siis jossa `x == lista[i] == True`. Palauttaa virheen, mikäli tämän arvoista alkiota 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 numeroarvon, joka kertoo millä kohdalla listaa on alkio, jolla on arvo x. Palauttaa virheen mikäli lista ei sisällä alkiota, jonka arvo on x.

count(*x*)

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

sort()

Järjestää listan alkiot arvojärjestykseen.

reverse()

Järjestää listan alkiot käänteiseen järjestykseen, 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]
```

Lista aliohjelman parametrina

Tässä vaiheessa Python-opiskelua on hyvä huomata, että lista näyttää toimivan tietyissä tilanteissa eritavoin kuin esimerkiksi merkkijono. Eli aliohjelmassa merkkijonoon tehdyt muutokset eivät ole pysyviä, mutta listaan tehdyt muutokset ovat. Tässä on kyseessä siitä, että kun funktiolle annetaan parametrina merkkijono, niin todellisuudessa annetaan vain kopio, mutta listasta siirtyy viittaus alkuperäiseen listaan. Esimerkki selventää:

Esimerkki 7.3. Lista vs. merkkijono

```
# Tiedosto: mutable.py
def muuta(merkkijono, lista):
    merkkijono = merkkijono + "def"
    lista.append("alkio3")
    print("2. Merkkijono '"+merkkijono+"' ja lista:", lista)
    return None

m = "abc"
l = ["alkio1", "alkio2"]
print("1. Merkkijono '" + m + "' ja lista:", l)
muuta(m, l)
print("3. Merkkijono '" + m + "' ja lista:", l)
```

Tuloste

```
>>>
1. Merkkijono 'abc' ja lista: ['alkio1', 'alkio2']
2. Merkkijono 'abcdef' ja lista: ['alkio1', 'alkio2', 'alkio3']
3. Merkkijono 'abc' ja lista: ['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.

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 muutokset listassa jäivät voimaan. Syy tähän parametrien erilaiseen toimintaan on se, että merkkijono on arvoparametri ja lista on muuttujaparametri.

Merkkijono on yksinkertainen tietotyyppi kuten kokonaisluku ja liukuluku, ja aliohjelmakutsujen yhteydessä Python-tulkki tekee näistä argumenteista kopioita, jotka se välittää kutsuttavalle funktiolle **arvoparametreina**. Koska aliohjelmassa on alkuperäisen argumentin *kopio*, ei arvoparametriin tehdyt muutokset välity takaisin kutsuvalle ohjelmalle. Lista-muuttuja on rakenteinen tietotyyppi eli se voi sisältää useita tietoalkioita ja näiden kohdalla Python-tulkki välittää aliohjelmalle alkuperäisen argumentin osoitteen **muuttujaparametrina**. Koska nyt tietoja ei kopioitu vaan lähetettiin *viittaus samaan muistipaikkaan* kuin kutsuvassa ohjelmassa, aliohjelmassa tehdyt muutokset tehdäänkin alkuperäisiin tietoihin ja muutokset näkyvät myös kutsuvassa ohjelmassa. Näin ollen edellisessä esimerkissä funktiota kutsutaan merkkijono-muuttujalla eli arvoparametrilla ja funktio sai sisäänsä kopion alkuperäisestä muuttujasta. Tässä tapauksessa merkkijonon ”abc” kopio ja alkuperäinen pysyi muuttumattomana. Listasta ei kuitenkaan tehty kopiota vaan välitettiin viittaus alkuperäiseen listaan, jolloin funktiossa tapahtuvat muutokset heijastuvat koko ajan alkuperäiseen listaan.

Mahdollisuus listan muuttamiseen aliohjelmassa ei ole aina hyvä asia ja siksi se voidaan estää kahdella tavalla. Funktiolle voidaan välittää listan sijaan tuple, joka on listaa vastaava rakenne ilman muokkausmahdollisuutta ja johon palaamme tarkemmin muutaman sivun päästä. Toinen vaihtoehto on välittää funktiolle kopio listasta.

Esimerkki 7.4. Lista vs. Merkkijono, osa II

```
# Tiedosto: mutable2.py

def muuta(merkkijono, lista):
    merkkijono = merkkijono + "def"
    lista.append("alkio3")
    print("2. Merkkijono '"+merkkijono+"' ja lista:", lista)
    return None

m = "abc"
l = ["alkio1", "alkio2"]
print("1. Merkkijono '" + m + "' ja lista:", l)
muuta(m, l[:]) # Huomaa listan sijasta sen leikkaus
print("3. Merkkijono '" + m + "' ja lista:", l)
```

Tuloste

```
>>>
1. Merkkijono 'abc' ja lista: ['alkio1', 'alkio2']
2. Merkkijono 'abcdef' ja lista: ['alkio1', 'alkio2', 'alkio3']
3. Merkkijono 'abc' ja lista: ['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.

Arvo- ja muuttujaparametrit sekä paluuarvot

Arvo- ja muuttujaparametrien erottelu voi tuntua työläältä. Sen oppii kyllä äkkiä päivittäisessä työssä, mutta turhien ongelmien välttämiseksi tässä oppaassa suositellaan, että aliohjelmien yhteydessä kaikki parametrit käsitellään arvoparametreina. Käytännössä tämä tarkoittaa sitä, että aliohjelman parametrit toimivat juuri niin kuin edellä on kerrottu, mutta aliohjelmasta palautetaan aina muuttuneet arvot paluuarvona `return`-käskyllä ja sijoitetaan ne muuttujan arvoksi kutsuvassa ohjelmassa. Alla oleva esimerkki selventää asiaa.

Esimerkki 7.5. Listan palautus paluuarvona

```
# Tiedosto: lista_paluuarvona.py
def lisaaNumeroita(nrot):
    for i in range(10):
        nrot.append(i)
    return nrot

numerot = []
print(numerot)
numerot = lisaaNumeroita(numerot)
print(numerot)
```

Tuloste

```
>>>
[]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Kuinka se toimii

Päätasolla luodaan tyhjä lista, jonka tyhjyys varmennetaan tulostamalla se. Tämän jälkeen lista lähetetään parametrina `lisaaNumeroita`-aliohjelmaan, joka lisää listaan numerot 0-9 ja palauttaa listan paluuarvona kutsuvaan ohjelmaan. Kutsuva ohjelma sijoittaa palautetun listan aiempaan muuttujaan ja tulostaa listan, jossa on nyt arvot 0-9. Palauttamalla lista aina aliohjelmasta paluuarvona ei tarvitse miettiä oliko kyseessä arvo- vai muuttujaparametri ja sen vuoksi se on tässä oppaassa suositeltava toimintatapa.

Luokka-rakenne

Lista on kätevä tietorakenne, mutta aina sen rakenne ei ole optimaalinen vaan joskus on parempi rakentaa uusia tietotyyppejä ja antaa niille sekä niiden tietoalkioille omat nimet. Esimerkiksi jos haluaisimme tallentaa henkilötietoja, olisi varmaan helpoin tapa tallentaa ne tietotyyppiin, joka on suunniteltu ihmisten henkilötietojen tallentamiseen. Tällöin voisimme määritellä, että jokaisella ihminen-tietotyypin jäsenellä on etunimi, sukunimi, ikä ja ammatti. Python-ohjelmointikielessä tämä onnistuu kätevästi luokkarakenteella ja sen tunnisteena on avainsana `class`. Seuraavassa esimerkissä tutustumme hieman tarkemmin siihen, kuinka tämä rakenne toimii.

Esimerkki 7.6. Luokka rakenteisena tietotyyppinä

```
# Tiedosto: luokka.py

# Määritellään ihminen;
# Ihmisellä on etu- ja sukunimi, ikä ja 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, \
"vuotta vanha", mies.ammatti, ".")
```

Tuloste

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

Kuinka se toimii

Kuten huomaat, koodi aloitetaan tietorakenteen määrittelyllä, joka muistuttaa funktion määrittelyä. Komennolla `class IHMINEN:` kerromme Python-tulkille, että määritelemme uuden **luokan**, jonka nimeksi tulee `IHMINEN`. Tämän jälkeen voimme määritellä `IHMINEN`-luokkaan kuuluvat jäsenmuuttujat, joita on tällä kertaa neljä: `sukunimi`, `etunimi`, `ammatti` ja `ika`. Huomaa, että jäsenmuuttujien tyypeillä tai alkuarvoilla ei varsinaisesti ole merkitystä vaan 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, että niille annetaan uusi arvo.

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

Esimerkki 7.7. Luokka rakenteisena tietomuotona, esimerkki 2

```
# Tiedosto: luokka2.py

# Määritellään luokka koira ja annetaan sille oletusarvot
class KOIRA:
   rotu = "Sekarotuinen"
   nimi = "Hurtta"

# Luodaan kaksi uutta Koira-oliota
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 luokan `KOIRA`, josta teimme kaksi oliota 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 luokan määrittelyn yhteydessä. Tärkeää tässä on muistaa se, että luokan 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 pistenotaatiota lukuun ottamatta.

Luokka globaalina tunnuksena

Luokka helpottaa yhteenkuuluvien tietojen keräämistä käyttäjältä ja usein sen jälkeen tiedot tallennetaan tiedostoon yhden olion tiedot aina yhdelle riville eroteltuina sopivalla erottimella (ks. edellinen luku). Näin ollen samaa tiedostoa luettaessa on luonnollista käyttää samaa luokkaa lähtökohtana olioille, kun riveillä olleita tietoja käsitellään taas ohjelmassa. Ja kun samaa tietorakennetta käytetään useissa eri paikoissa ohjelmaa – tietojen kysyminen sekä tiedoston luku ja kirjoitus – kannattaa joka paikassa ohjelmaa käyttää samaa luokan määrittelyä. Tämän mahdollistamisiksi luokan määrittely sijoitetaan tyypillisesti ohjelmatiedoston alkuun päätasolle globaaliksi tunnukseksi, jotta se näkyy kaikkialle ohjelmassa.

Tässä oppaassa luokkien nimet kirjoitetaan isoilla kirjaimilla, esim. ”class IHMINEN:”. Tämä ei ole välttämätöntä, mutta helpottaa luokan ja olioiden/muuttujien erottamista toisistaan ja siten vähentää virheitä.

Luokan jäsenfunktioista

Luokkien yhteydessä puhutaan *jäsenmuuttujista*, joten herää kysymys liittyvätkö aiemmin oppaassa käsittelemämme *jäsenfunktiot* myös luokkiin? Vastaus tähän on kyllä. Luokassa

voi olla jäsenmuuttujien lisäksi myös jäsenfunktioita, joiden avulla rakenne joko muokkaa omia tietojaan tai käsittelee saamiansa parametreja. Jäsenfunktiot ovat olio-ohjelmoinnin keskeinen ominaispiirre kun taas tietorakenteet eli tietojen paketointi yhden muuttujan taakse on tyypillinen piirre ohjelmointikielissä. Esimerkiksi C-kielessä uusia tietorakenteita voi tehdä struct-rakenteella, mutta C-kielessä ei ole jäsenfunktioita. Koska tämä ei ole olio-ohjelmoinnin opas, keskitymme tässä oppaassa olioihin ja jäsenmuuttujiin tietorakenteina ja jäsenfunktioita käytämme tarpeen mukaan, mutta emme esim. käsittele ollenkaan niiden tekemistä vaan se jää olio-ohjelmointikursseille.

Tämän oppaan kannalta luokkarakenteeseen liittyy jäsenfunktioita, joiden avulla ne voivat käsitellä saamiaan parametreja tai omia tietojaan. Esimerkiksi yhden rivin lukemiseen tiedostosta käytämme jäsenfunktiota `readline`.

Python on itse asiassa täysverinen olio-ohjelmointikieli, vaikka tällä kurssilla sitä käytetään ensisijaisesti proseduraaliseen ohjelmointiin. Aiheesta löytyy tarkempaa tietoa esimerkiksi Python Software Foundationin tutoriaalin osassa 9.

Luokka/olio ja lista sopivat yhteen

Usein on tarvetta käyttää listan alkioina toisia listoja tai luokan instansseja, olioita, pelkkien lukujen tai merkkijonojen sijaan. Tarkastellaan seuraavaa esimerkkiä, jossa listaan tallennetaan opiskelijoiden tietoja olioina.

Esimerkki 7.8. Listaan lisätään luokasta luotuja muuttujia eli olioita

```
# Tiedosto: opiskelijat.py
# Määritellään Opiskelija-luokka
class OPISKELIJA:
    etunimi = ""
    sukunimi = ""
    opiskelijanumero = "0"

# Määritellään opiskelijat-lista
opiskelijat = []

# Luodaan Opiskelija-oliota
opiskelija1 = OPISKELIJA()
opiskelija1.etunimi = "Brian"
opiskelija1.sukunimi = "Kottarainen"
opiskelija1.opiskelijanumero = "0123456"

opiskelija2 = OPISKELIJA()
opiskelija2.etunimi = "Susanna"
opiskelija2.sukunimi = "Matikainen"
opiskelija2.opiskelijanumero = "0338765"

# Lisätään opiskelijat listaan
opiskelijat.append(opiskelija1)
opiskelijat.append(opiskelija2)

# Tulostetaan opiskelijoiden tiedot
for opiskelija in opiskelijat:
    print("Nimi:", opiskelija.etunimi, opiskelija.sukunimi)
    print("Opiskelijanumero:", opiskelija.opiskelijanumero)
```

Tuloste

```
>>>
Nimi: Brian Kottarainen
Opiskelijanumero: 0123456
Nimi: Susanna Matikainen
Opiskelijanumero: 0338765
>>>
```

Kuinka se toimii

Ohjelma on jokseenkin entisen kertausta siihen asti kunnes lisäämme listaan sisältöä. Tällä kertaa lisäämme listaan vastikään `OPISKELIJA`-luokasta luodut `opiskelija1` ja `opiskelija2` -oliot.

Tämän jälkeen tulostamme listamme sisällön. Saamme listan jokaisen alkion tilapäissäiliöön `opiskelija`, joka on siis `OPISKELIJA`-luokan instanssi, joten voimme kysyä siltä jäsenmuuttujia `etunimi`, `sukunimi` ja `opiskelijanumero` ja tulostaa niitä ruudulle.

Tuple-rakenne

Kuten aiemmin kävi ilmi, tuple muistuttaa erittäin paljon listaa. Emme paneudu tupleen sen tarkemmin, mutta perusasiat on hyvä tietää, sillä muuttujasta on helppo tehdä vahingossa listan sijaan tuplen ja tästä seuraavat erot on hyvä tunnistaa.

Tuplet ovat rakenteeltaan samanlaisia kuin listat yhdellä merkittävällä erolla: ne ovat alkioiltaan kiinteitä. Tämä tarkoittaa sitä, että tupleilla ei ole metodeja eikä muitakaan tapoja manipuloida niitä. Eli tuplen sisältö on muuntumaton alustamisen jälkeen. Tupleja käytetään usein kun halutaan varmistaa, että esimerkiksi funktiolle annettu sarjamainen parametri ei tule muuttumaan missään vaiheessa.

Esimerkki 7.9. Tuplen käyttäminen

```
# 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”.

Tuplen ja listan eroja käytännössä

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
```

Osaamistavoitteet

Ymmärrät tietorakenteet lista ja luokka/olio ja osaa käyttää niitä ja listan jäsenfunktioita. Osaat käyttää listaa aliohjelmien kanssa ja tiedät tuplen ja listan erot. Osaat koota tähän asti oppaassa käsitellyistä asioista valikkopohjaisen ohjelman, jossa on kaikki perustoiminnot kunnossa esimerkin 7.10 mukaisesti.

Luvun asiat kokoava esimerkki

Luvun kokoava esimerkki 7.10. kokoaa yhteen tähän asti tässä oppaassa läpikäydyt asiat. Ohjelman käyttöliittymä näkyy `valikko()`-aliohjelmassa, ja tyypillisesti valikon valinnat käyvät ilmi kirjoitettavan ohjelman toimeksiannosta. Valikon rakenne heijastuu pääohjelman valintarakenteeseen ja tyypillisesti jokaiseen valikon kohtaan liittyvä toiminnallisuus toteutetaan omana aliohjelmanaan. Tässä ohjelmassa on tiedostonkäsittelyyn liittyvät `tallenna()` ja `lue()` aliohjelmat sekä käyttäjältä tiedot kysyvä `kysy()`-aliohjelma. Tiedot näiden kaikkien aliohjelmien välillä välitetään lista-parametrilla ja paluuarvolla. Koska useissa aliohjelmissa käytetään samaa luokkaa ja tiedoston kenttäerotinta, on ne määritelty globaaleina tunnuksina päätasolla ohjelman alussa. Globaalien tunnusten kohdalla kannattaa muistaa, etteivät ne muutu ohjelman suorituksen aikana vaan muuttuvia tietoja sisältävät muuttujat ovat paikallisia muuttujia aliohjelmissa ja tieto kulkee aliohjelmien välillä parametreilla sekä paluuarvoilla.

Perusohjelmaan tulee seuraavissa luvuissa vielä tärkeitä laajennuksia ja tarkennuksia, mutta perusohjelman perustoiminnallisuus näkyy tässä esimerkissä 7.10. Ohjelman teko tyhjästä näkyy lukuun liittyvällä ohjelmointivideolla 2. Tässä ohjelmassa ei näy videolla oleva `tulostus`-aliohjelma, koska oleelliset toiminnot näkyvät `tallenna`-aliohjelmassa.

Itse ohjelma on seuraavalla sivulla, jotta koko ohjelma mahtuu yhdelle sivulle.

Esimerkki 7.10. Valikkopohjainen ohjelma listalla, luokalla ja olioilla

```
# Tiedosto: valikko_ohjelma.py
EROTIN = ';'   # Globaali kiintoarvo

class HENKILO: # Globaali luokka
    nimi = ""
    pituus = 0
########################################################################

def valikko():
    print("Valitse toiminto:")
    print("1) Kysy tiedot")
    print("2) Tallenna tiedot")
    print("3) Lue tiedot")
    print("0) Lopetus")
    valinta = int(input("Valintasi: "))
    return valinta

def kysy(lista):
    hlo = HENKILO()
    hlo.nimi = input("Anna nimi: ")
    hlo.pituus = int(input("Anna pituus: "))
    lista.append(hlo)
    return lista

def tallenna(nimi, lista):
    tiedosto = open(nimi, 'w', encoding="utf-8")
    for hlo in lista:
        tiedosto.write("{0:s}{1:s}{2:d}\n".format(hlo.nimi, EROTIN, hlo.pituus))
    tiedosto.close()
    return None

def lue(nimi, lista):
    tiedosto = open(nimi, 'r', encoding="utf-8")
    while (True):
        rivi = tiedosto.readline()
        if (len(rivi) == 0):
            break
        rivi = rivi[:-1]
        sarake = rivi.split(EROTIN)
        hlo = HENKILO()
        hlo.nimi = sarake[0]
        hlo.pituus = int(sarake[1])
        lista.append(hlo)
    tiedosto.close()
    return lista

def paaohjelma():
    tiedot = []
    tiedosto = "L07Demo.txt"
    while (True):
        toiminta = valikko()
        if (toiminta == 1):
            tiedot = kysy(tiedot)
        elif (toiminta == 2):
            tallenna(tiedosto, tiedot)
        elif (toiminta == 3):
            tiedot = lue(tiedosto, tiedot)
        elif (toiminta == 0):
            print("Kiitos ohjelman käytöstä.")
            tiedot.clear()
            break
        else:
            print("Tuntematon valinta, yritä uudestaan.")
        print()
    return None

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

Luku 8: Kirjastot ja moduulit

Funktioista puhuttaessa näimme, kuinka pystymme käyttämään uudelleen aikaisemmin muualla kirjoitettua koodia. Entäpä jos haluaisimme käyttää aikaisemmin tekemäämme funktiota jossain toisessa ohjelmassa eli tiedostossa? Pythonissa tämä voidaan toteuttaa käyttämällä moduuleja. Moduuli on periaatteessa lähdekooditiedosto, joka sisältää funktioita, joita voidaan tuoda käytettäväksi sisällyttämisen (eng. `import`) avulla. Todellisuudessa olemme 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. Ohjelmointiympäristön mukana toimitettavista lisätoimintoja tarjoavista moduuleista käytetään usein nimitystä kirjastomoduuli (erityisesti Python) tai funktiokirjasto (C, C++).

Kuten sanottu, käyttääksemme kirjastomoduuleita – valmiita muiden tekemiä 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 8.1. math-moduuli

```
# Tiedosto: math_moduuli.py

import math

print("Tehdään muutama korkeampaa matematiikkaa vaativa operaatio:")
print("e toiseen 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 olettaa.

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 (eli kiintoarvoa) `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 tällainen 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 ohjelmia. Niiden poistaminen on sallittua; jos tiedostoa ei ole olemassa, tekee tulkki uuden käännöksen, kunhan alkuperäinen lähdekooditiedosto (.py) on saatavilla.

from...import -sisällytyskäsky

Joissain tapauksissa saatamme törmätä tilanteeseen, jossa haluamme saada funktiot ja muuttujat suoraan käytettäviksemme ilman pistenotaatiota. Eli siis päästä esimerkiksi `math`-kirjaston funktioihin käsiksi ilman `math`-etuliitettä käyttämällä pelkkää funktionimeä `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 kaikki `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 `sin`iin, niin voisimme tehdä sen käskyllä `from math import sin`.

Tähän sisällytystapaan liittyy yksi vakava riski, ylikirjoittumisvaara. Jos kaksi moduulia sisältää samannimisen funktion, kirjoitetaan ensin sisällytetyn moduulin funktio yli 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()
```

johtaisi seuraavaan tulosteeseen

```
>>>
"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 jo tehnyt niitä. Jokaista funktioita sisältävää Python-lähdekooditiedostoa 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 8.2 Oma tiedosto moduulina

```
# Tiedosto: omamoduli.py

def terve():
    print("Tämä tulostus tulee moduulin omamoduli 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 vakiot (kiintoarvot) `versio` ja `sana`, joista ensimmäinen kertoo `omamoduli`-moduulin versionumeron ja toinen sisältää vain satunnaisen merkkijonon. Seuraavaksi teemme ohjelman, jolla kokeilemme moduulin toimintaa:

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

Tuloste

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

Kuinka se toimii

Huomaa, että nyt sisällytimme moduulin "`omamoduli`" `import`-käskyllä, joten joudumme käyttämään pistenotaatiota. Luonnollisesti `from…import` –rakenteella olisimme päässeet siitä eroon, mutta edellisen keskustelun mukaisesti emme tee näin välttääksemme 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

Aiemmin esitellyn `math`–moduulin avulla pääsemme käsiksi erilaisiin matemaattisiin apufunktioihin, joilla voimme laskea mm. trigonometrisia arvoja ja 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 esitellään kaikki kirjastomoduulit sekä niiden sisältämät funktiot ja hyödylliset vakiot, 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. Tällä kurssilla näistä sopivin on yleensä randint-funktio, joka palauttaa siis satunnaisen kokonaisluvun annetulta väliltä. Satunnaisuuteen liittyen kannattaa pitää mielessä, että kyseessä on pseudosatunnaisluku eli kyseessä ei ole "aito" satunnaisluku vaan se tulee tietyn laskennan perusteella kuten Pythonin dokumentaatiossa kerrotaan. Tämän oppaan kannalta satunnaisuus on riittävää, mutta tällä moduulilla ei siis kannatta lähteä jakamaan rahaa esim. lauantai-illan Lotossa.

```
>>> import random

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

>>> # Satunnainen joukko
>>> random.sample(range(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 kellonajan ja päivämäärän sekä muotoilla niitä eri tavoin. Moduuli tuntee englanninkieliset nimet kuukausille ja viikonpäiville, sekä mahdollistaa monipuoliset ja nopeat laskutoimitukset kalenteripäivämäärillä. Osa moduulin toiminnoista ottaa huomioon myös aikavyöhykkeet. Datatiedostoissa olevat aikatiedot mahdollistavat datan tehokkaan luokittelun eri ryhmiin. Esimerkiksi energialaitokselta saatavat sähkönkulutustiedot perustuvat tunnin tarkkuudella kirjattuihin sähkömittarin lukemiin joka päivä, joten sähkönkulutustieto-tiedostossa on ensin päivämäärä ja kellonaika sekä sen jälkeen paljon erilaista tietoa sähkönkäyttöön liittyen.

Tiivistetysti aikatietoja voidaan käyttää kolmeen asiaan: tiedon muotoiluun, laskemiseen ja yksittäisten tietojen hyödyntämiseen. Kuten yllä tuli jo esille, aikaan liittyy aikavyöhykkeet ja esim. Suomessa aika annetaan yleensä Itä-Euroopan vyöhykeajassa, mutta joskus käytetään myös Koordinoitua yleisaikaa (aiemmin Greenwich Mean Time eli GMT ja nyt UTC). Näistä ei yleensä tarvitse välittää, paitsi matkustaessa, mutta joskus niihin törmää datan kanssa ja `datetime`-moduuli auttaa silloin. Aikatieto esitetään tekstitiedostossa merkkijonona, mutta `datetime`-moduuli perustuu aika-olioihin, joita käyttämällä tiedon muotoilu helpottuu merkittävästi. Aika-olion käyttö mahdollistaa myös laskemisen, josta tyypillinen esimerkki on iän laskenta syntymäpäivän ja halutun päivämäärän välillä. Ja yksittäisiä tietoja voidaan hyödyntää erityisesti datan luokittelussa, sillä aika-olion jäsenmuuttujat ja –funktiot osaavat kertoa esim. ko. ajan kuukauden, viikon, viikonpäivän.

Lähtökohtana on moduulin käyttöönotto ja päivämäärän asettaminen eli

```
>>> # luodaan päiväys, aloitetaan kirjaston käyttöönotolla
>>> import datetime

>>> # Tämä päivä kokonaisuudessaan, eli aika nyt sekunnit mukaan lukien
>>> nyt = datetime.datetime.now()
>>> nyt
datetime.datetime(2010, 4, 21, 12, 43, 25, 987000)
```

`datetime`-moduuli sisältää `datetime`-olion, jolla on `now`-jäsenfunktio, joka palauttaa päivämäärän ja kellonajan yhtenä oliona, ts. `datetime.datetime.now()`. Kuten yllä näkyy, tämä olio sisältää tiedot hyvässä tallessa jäsenmuuttujissa. Mikäli olion sisältämiä tietoja halutaan tulostaa, kannattaa käyttää `strftime`-jäsenfunktiota, jolla päivämäärän ja kellonajan voi muotoilla merkkijonoksi tarkasti haluamallaan tavalla. Muotoilumerkeistä tärkeimmät näkyvät alla Taulukko 8.1:ssä.

```
>>> # 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.'
```

`strftime`-funktiolle on myös käänteinen funktio `strptime`, joka osaa tehdä merkkijonona annetusta ajasta aika-olion yllä olevan `now()`-funktion tavoin.

```
>>> datetime.datetime.strptime("30.10.2019 12:58", "%d.%m.%Y %H:%M")
datetime.datetime(2019, 10, 30, 12, 58)
```

Laskenta onnistuu `datetime`-moduulilla seuraavalla tavalla. Esimerkiksi jos tiedät kaksi ajankohtaa kuten nykyisen päivämäärän ja henkilön syntymäpäivän, voit laskea niiden välissä olevan ajan pituuden vähennyslaskuna. Alla oleva `datetime.date.today()` antaa meille nykyisen päivän ilman tunteja ja minuutteja.

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

Toinen esimerkki laskennasta on siirtymä eli ero (delta) aikojen välillä. Tämä onnistuu `timedelta`-funktiolla alla olevan mukaisesti. Huomaa, että siirtymänä kannattaa käyttää päiviä, sekunteja tai mikrosekunteja, sillä minuutit, tunnit ja viikot muutetaan näihin yksiköihin automaattisesti. Siirtymä voi olla positiivinen tai negatiivinen luku.

```
>>> tanaan = datetime.datetime.now() # 30.10.2019
>>> toissa_paivana = tanaan + datetime.timedelta(days=-2)
>>> toissa_paivana.strftime("%A %d.%m.%Y")
'Monday 28.10.2019'
>>>
```

Yllä olevissa esimerkeissä näkyy erilaisia tapoja käyttää jäsenfunktioita ja esim. `datetime.datetime.now()` ja `datetime.timedelta(days=-2)` näyttävät erilaisilta. Käskyjen ensimmäinen `datetime` on kirjaston nimi ja tässä kirjastossa on kaksi rinnakkaista luokkaa, `datetime` ja `timedelta`. Näistä `datetime`-oliolla on lisäksi `now()`-jäsenfunktio (joka palauttaa `datetime`-olion). Tämä voi tuntua sekavalta, mutta nämä liittyvät läheisesti aiemmin puhuttuun nimiavaruus-asiaan. Analogisesti voimme ajatella, että LUTissa on Viipuri-sali ja kaikki tietävät, että se on pääoven vieressä. Sali 2310 taas löytyy 2-rakennuksesta 3 kerroksesta ja riittävän vanhat tietävät, että tuo sali oli aiemmin nimeltään 10-sali. Toisin sanoen nämä kirjasto-luokka-funktio ja vastaavat ketjut rakentuvat loogisesti ja ne, jotka käyttävät niitä tarpeeksi paljon oppivat ne samalla tavoin kuin esim. rakennusten huoneiden nimeämislogiikan. Ja ennen kuin muistaa huoneiden sijainnit, kannattaa käyttää huonekarttaa apuna. Ohjelmoinnissa kannattaa vastaavasti etsiä toimivia esimerkkejä ja käyttää niitä apuna kun pitää tehdä itse vastaavia asioita.

Edellisissä esimerkeissä näkyi useita tapoja saada kiinnostavia tietoja aika-oliosta kuten esimerkiksi päivä, kuukausi ja vuosi (%d, %m, %Y). Muotoilumerkeillä saadaan tiedot merkkijonoon, mutta usein käytettäviä tietoja saa myös jäsenmuuttujista. Esimerkiksi `datetime`-oliolla on jäsenmuuttujat `year`, `month`, `day`, `hour`, `minute`, `second` sekä jäsenfunktio `weekday()`, joka palauttaa päivän järjestysnumeron (maanantai 0…sunnuntai 6).

Alla olevassa taulukossa näkyy joukko yleisimpiä muotoilumerkkejä ja sitä täydellisempi luettelo löytyy esim. Python dokumentaatiosta (F1 eli Help-tiedosto) kohdasta “datetime — Basic date and time types”. Jos tarvitset jotain muuta vastaavaa tietoa, kannattaa tarkistaa Python-dokumentaatiosta olisiko ko. tieto saatavilla jossain jo valmiina.

Taulukko 8.1 Päivämäärä-tietojen muotoilumerkkejä, tiivistelmä Python Help:stä

Directive	Meaning	Example
`%a`	Weekday as locale’s abbreviated name.	Sun, Mon, …, Sat (en_US)
`%A`	Weekday as locale’s full name.	Sunday, Monday, …, Saturday (en_US)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6
`%d`	Day of the month as a zero-padded decimal number.	01, 02, …, 31
`%b`	Month as locale’s abbreviated name.	Jan, Feb, …, Dec (en_US)
`%B`	Month as locale’s full name.	January, February, …, December (en_US)
`%m`	Month as a zero-padded decimal number.	01, 02, …, 12
`%y`	Year without century as a zero-padded decimal number.	00, 01, …, 99
`%Y`	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12
`%M`	Minute as a zero-padded decimal number.	00, 01, …, 59
`%S`	Second as a zero-padded decimal number.	00, 01, …, 59
`%z`	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive).	(empty), +0000, -0400, +1030, +063415, -030712.345216
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, EST, CST
`%j`	Day of the year as a zero-padded decimal number.	001, 002, …, 366
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, …, 53
`%%`	A literal `'%'` character.	%

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.perf_counter()      # Odotetaan hetki
16146.506199272
>>> time.perf_counter()      # Odotetaan hetki
16152.503509738
```

Esimerkki ei kerro itessään kerro mitä tapahtuu, paitsi että kyseessä on taas yksinkertainen funktiokutsu, joka palauttaa kellonajan desimaalilukuna. Toistamalla itse työn yllä olevan esimerkin huomaat, miten uusi luku on edellistä suurempi eli aikaa on kulunut.

Windows-järjestelmässä `perf_counter` käyttää Windowsin rajapinnassa olevaa palvelua ja funktio palauttaa liukulukuna tiedon kellon ajasta kutsuhetkellä. Ja kun näit kutsuja on peräkkäin, paljastaa kutsujen erotus kuluneen ajan. Eli tällä funktioilla voidaan ottaa talteen ajanhetkiä ja laskea niiden erotus, jonka avulla voidaan sitten arvioida esimerkiksi eri funktioiden käyttämiä aikoja eli mikä on hidas ja mikä nopea.

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 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”. Standardikirjastossa on myös oma luku Python Profilers, jossa käydään läpi Pythonin tarjoamia ratkaisuja ohjelmien profilointiin eli suorituskyvyn arviointiin.

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 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.lut.fi")
sisalto = sivu.read() # luetaan sivun sisältö
sivu.close()
tekstina = sisalto.decode("utf-8") # parsitaan merkistö oikeanlaiseksi
print(tekstina)
```

Tämä tulostaa LUT:n 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. Jos siis näin käy, tiedät varautua asiaan ja sallia uuden yhteyden. Luonnollisesti, jos ohjelma ei saa yhteyttä verkkoon, ei funktiokaan saa mitään haettua.

fractions-moduuli

Pythonin matemaattisen taidot eivät rajoitu pelkästään kokonais- ja desimaalilukujen käsittelyyn, vaan mukaan voidaan ottaa 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.

Muita moduuleja

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

Ulkoisten kirjastojen lisääminen Pythoniin

Pythonin mukana tulee paljon hyödyllisiä kirjastoja ja ne saa käyttöön `import`-käskyllä. Luonnollisesti on olemassa myös kirjastoja, jotka eivät tule Pythonin perusasennuksessa mukana ja ne pitää ensin asentaa ennen kuin niitä voi käyttää `import`-käskyllä. Esimerkkejä tällaisista kirjastoista ovat seuraavat kolme pakettia:

1. **numpy** – Numerical Python, erityisesti matriisit/taulukot.
2. **matplotlib** – graafien ja diagrammien piirtämiseen tehty kirjasto
3. **svgwrite** – vektorigrafiikan piirtämiseen tehty kirjasto, jolla pystyy tekemään svg-tiedostoformaatin mukaisia piirustuksia (Scalable Vector Graphics).

Nämä kirjastot toimivat esimerkkeinä siitä, miten Pythonin käyttömahdollisuuksia voi entisestään laajentaa ja tehostaa ja asiasta kiinnostuneet voivat etsiä lisäinformaatiota yllä olevilla hakusanoilla Internetistä. numpy-kirjaston matriiseihin tutustutaan tarkemmin luvussa 10, mutta muutoin näiden kirjastojen käyttö jää oman harrastuksen varaan.

Mainitut kirjastot voi asentaa Pythoniin sen omalla asennustyökalulla PIP. Koska numpy-kirjasto on matplotlib-kirjaston sisällä, ei sitä tarvitse asentaa erikseen. Alla on askeleet Pythonin asennukseen, jonka jälkeen siihen lisätään kirjastot. Mikäli Pythonin asennus on kunnossa, ei sitä tarvitse toistaa vaan voit siirtyä suoraan kirjastojen asennukseen PIP:llä.

Tee Pythonin perusasennus koneelle asennusohjelman ohjeiden mukaisesti ja huomaa seuraavat asiat

1. Perusasennuspaketti, esim. https://www.python.org/downloads/release/python-372/
2. Asennuksessa tulee olla mukana IDLE, PIP, dokumentaatio, launcher ja ympäristömuuttujat tulee päivittää, erityisesti Python tulee lisätä polku/PATH-muuttujaan

Onnistuneen Python-perusasennuksen jälkeen lisätään kirjastot PIP:llä:

1. Käynnistä Windowsin Komento-ikkunaa (cmd) ja mene Pythonin asennushakemistoon
2. Tarkista, että asennus on mennyt oikein ja seuraaviin käskyihin tulee järkevät versionumerot eikä ilmoitus, ettei ohjelmaa löydy tms. virheilmoitus. Huomaa, että ”version” sanan edessä on 2 kpl tavuviivoja eli --
   a. python --version
   b. pip --version
3. Asenna ensin matplotlib ja sen jälkeen svgwrite alla olevilla käskyillä
   a. pip install matplotlib==3.0.2
   b. pip install svgwrite==1.2.1

Molempien PIP-käskyjen pitäisi tuottaa useita rivejä tietoa asennuksen etenemisestä ja onnistumisesta. Huomaa, että em. versionumerot 3.7.2, 3.0.2 ja 1.2.1 eivät ole satunnaisia vaan em. ohjelmistojen yhteensopivia versioita. Koska matplotlib sisältää numpy-kirjaston, ei sitä tarvitse asentaa erikseen.

Osaamistavoitteet

Osaat käyttää Pythonin yleisempi kirjastomoduuleita ja osaat tehdä oman moduulin sekä käyttää sitä. Keskeisiä kirjastomoduuleja tällä kurssilla ovat etenkin math, random ja datetime.

Luvut asiat kokoava esimerkki

Esimerkki 8.3. Oma kirjasto ja datetime-kirjaston peruskäskyjä

Pääohjelma

```
########################################################################
# (c) LUT 2020 L08Paaohjelma.py un
# Tämä esimerkki on tarkoitettu omatoimisen oppimisen tueksi ohjelmoinnin
# opiskeluun. Muu käyttö kielletty.
########################################################################

import datetime
import L08Kirjasto

def paaohjelma():
    lista = [] # Alustuksia ja alkutoimenpiteitä
    tulokset = None
    nimi = input("Anna luettavan datatiedoston nimi: ")
    alku = input("Anna ensimmäinen huomioitava päivä: ")
    loppu = input("Anna viimeinen huomioitava päivä: ")
    pvAlku = datetime.datetime.strptime(alku, "%d.%m.%Y")
    pvLoppu = datetime.datetime.strptime(loppu, "%d.%m.%Y")
    lista = L08Kirjasto.lue(lista, nimi, pvAlku, pvLoppu)
    tulokset = L08Kirjasto.analysoi(lista, tulokset)
    L08Kirjasto.tulosta(tulokset)
    return None

paaohjelma()

########################################################################
# eof
```

Kirjasto

```
########################################################################
# (c) LUT 2020 L08Kirjasto.py un Kirjasto, kiintoarvot, datetime
# Tämä esimerkki on tarkoitettu omatoimisen oppimisen tueksi ohjelmoinnin
# opiskeluun. Muu käyttö kielletty.
########################################################################
# Datatiedoston rivi: ”21-01-2017;3001;14624;10,81;16;507;346;70;26;1826”

import datetime # Kirjastojen import tiedoston alussa

PAIVIA = 7 # Kiintoarvot
PAIVAT = ["Ma", "Ti", "Ke", "To", "Pe", "La", "Su"]
EROTIN = ';'

class DATA: # Luokat
    aika = None
    askelia = 0

class TULOKSET:
    maxAika = None
    maxAskelia = 0
    keskiarvo = 0
    viikko = []
```

```
def lue(lista, nimi, alku, loppu):
    tiedosto = open(nimi, 'r')

    while (True): # Tiedoston lukeminen ja tallennus olioina listaan
        rivi = tiedosto.readline()
        if (len(rivi) == 0):
            break
        sarakkeet = rivi[:-1].split(EROTIN)
        aika = datetime.datetime.strptime(sarakkeet[0], "%d-%m-%Y")
        if ((aika >= alku) and (aika <= loppu)): # Haluttu aikaväli
            data = DATA()
            data.aika = aika
            data.askelia = int(sarakkeet[2]) # askeltiedot 2. sarake
            lista.append(data)
    tiedosto.close()
    return lista

def analysoi(lista, tulokset):
    viikko = []
    summa = 0
    maxAika = lista[0].aika
    maxAskelia = lista[0].askelia
    paivia = (lista[-1].aika - lista[0].aika).days

    for i in range(PAIVIA):
        viikko.append(0)

    for alkio in lista: # Analyysi
        summa += alkio.askelia # Keskiarvoon tarvitaan summa
        viikko[alkio.aika.weekday()] += alkio.askelia # Jakauma päiville
        if (maxAskelia < alkio.askelia): # Suurin askelmäärä ja päivä
            maxAika = alkio.aika
            maxAskelia = alkio.askelia

    tulokset = TULOKSET()
    tulokset.maxAika = maxAika
    tulokset.maxAskelia = maxAskelia
    tulokset.keskiarvo = int(summa / paivia)
    tulokset.viikko = viikko

    return tulokset

def tulosta(tulokset):
    print("Suurin askelmäärä {0:d} tuli {1:%d.%m.%Y}.".format(
                            tulokset.maxAskelia, tulokset.maxAika))
    print("Keskimäärin askelia kertyi {0:d} päivässä.".format(
                            tulokset.keskiarvo))

    print("Viikonpäivittäin askelet jakautuivat seuraavasti:")
    for pv in range(PAIVIA):
        print("{0:s};{1:d}".format(PAIVAT[pv], tulokset.viikko[pv]))

    print("Ohjelman suoritus päättyi {0:%d.%m.%Y} kello {0:%H:%M}.".
                             format(datetime.datetime.now()))
    return None
########################################################################
# eof
```

Luku 9: 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. Usein järkevä lopputulos on kertoa käyttäjälle tapahtuneesta ja lopettaa ohjelman suoritus hallitusti. Näin käyttäjä tietää, mikä meni pieleen ja voi yrittää ratkaista asian sopivalla tavalla. Aloitetaan katsomalla ensin miten virhetilanteita voi joissain tapauksissa estää ennakolta.

Virhetilanteiden estäminen ennakolta

Jossain tapauksissa virhe-tilanne voidaan havaita ehtolauseella ja estää siten varsinaisen virheen tapahtuminen kokonaan. Esimerkiksi aliohjelman alussa voidaan tarkistaa parametrin arvon järkevyys ja palata aliohjelmasta takaisin heti, jos se on tilanteen kannalta järkevin toimintamalli. Esimerkiksi aliohjelma voi odottaa parametrinä tulostettavaa listaa ja jos lista onkin tyhjä, ei aliohjelmassa ole mitään tehtävissä vaan sieltä voidaan palata heti ja ilmoittaa käyttäjälle ongelmasta esimerkin 9.1 mukaisesti.

Esimerkki 9.1. Virhetilanteen tunnistaminen ennakolta

```
def tulosta(nimet):
    if (nimet == None):
        print("Ei nimiä tulostettavaksi, selvitä ne ensin.")
        return None
    ...
    return nimet
```

Esimerkissä 9.1 parametrinä oleva lista-muuttuja `nimet` on alustettu `None`-arvolla, jotta virheentarkistus on mahdollista. Ennen nimien tulostamista ne pitäisi luonnollisesti selvittää, mutta jos näin ei jostain syystä ole tehty, voidaan `tulosta`-aliohjelmaan päätyä tyhjän listan kanssa. Esimerkissä palautetaan `None`-muuttuja kutsuvaan ohjelmaan virheen merkkinä kun taas onnistunut aliohjelmakutsu palauttaa lista-muuttujan onnistumisen merkiksi, jotta kutsuva ohjelma voi reagoida tilanteeseen tarpeen mukaan. Tälläisen virheentarkistuksen voi tehdä aliohjelmassa tai kutsuvassa ohjelmassa. Mikäli aliohjelma on monien ohjelmien käyttämässä kirjastossa, palvelee aliohjelmassa oleva virheentunnistus niitä kaikkia ja näin yleiskäyttöisestä aliohjelmakirjastosta saadaan luotettavampi ja mahdolliset virhetilanteet pystytään estämään ennakolta.

Virheistä yleisesti

Kaikki tähän mennessä itse ohjelmia kirjoittaneet ovat varmasti nähneet virheilmoituksia. 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 eli poikkeuksen ja lopettaa. Voimmeko ottaa nämä poikkeukset kiinni ohjelmassa ja toimia tilanteeseen sopivalla tavalla?

try…except-rakenne

`try...except` on olemassa virheiden/poikkeusten kiinniottamista ja niiden käsittelyä 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/poikkeuksen nimeltään `ValueError`, joka käytännössä tarkoittaa sitä, että tulkki yritti muuttaa tekstiä numeroksi onnistumatta siinä, koska arvo oli ei ollut sopiva. 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 tehdä `try...except` –rakenteen. Käytännössä tämä tapahtuu siten, että laitamme `try`-osioon sen koodin, jonka suoritamme kun kaikki menee normaalisti, ja `except`-osioon sen koodin, joka suoritetaan poikkeustapauksessa eli virheen tapahtuessa. Käytännössä kyseessä on poikkeustilanteiden käsittelyyn sovitettu valintarakenne ja näin ollen joissain tapauksissa virhetilanteita voidaan hoitaa myös tavallisella valintarakenteella.

Esimerkki 9.2. Virheen käsitteleminen

```
# 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 virheelle alttiin lauseen/koodin `try`-osioon ja virheen tapahtuessa ajettavan koodin `except`-osioon. `except`–osiolle voidaan määritellä käsiteltävä virheluokka – tässä tapauksessa `ValueError` – tai sitten voimme jättää luokan pois, jolloin `except`-osio suoritetaan aina virheen tyypistä riippumatta. Jos esimerkin tapauksessa keskeyttäisimme suorituksen näppäinyhdistelmällä `Ctrl-C`, joka aiheuttaa näppäimistökeskeytyksen, saisimme vastaavan 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 lisätä 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

```
# Tiedosto: kaato2.py
try:
    luku = int(input("Anna luku: "))
    print("Annoit luvun", luku)
except:
    print("Ohjelmassa tapahtui virhe")
```

Saisimme kommentin “Ohjelmassa tapahtui virhe” riippumatta siitä, mikä virhe ohjelman ajon aikana tapahtui. Tämä on käyttäjän ja ohjelman parantamisen kannalta ongelmallinen ratkaisu, koska nyt emme tiedä mikä meni vikaan, emmekä voi selvittää syytä virheeseen. Lisäksi tässä 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")
```

Poikkeus `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. Jos tunnistamattomien poikkeusten riski on ilmeinen, voi virheen kuvauksen ottaa talteen alla näkyvällä ’`as virhe`’ rakenteella ja tulostaa tämän jälkeen `virhe`-muuttujan sisältämän ilmoituksen käyttäjälle.

```
except Exception as virhe:
    print("Tunnistamaton virhe: '{0:s}'.".format(virhe))
```

Kannattaa huomata, että mikäli toipumismekanismit ovat useille eri virheluokalle samat, voidaan niiden toiminta yhdistää samaan `except`-osioon seuraavalla tavalla:

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

Tämä 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. Näin ollen monipuolisen poikkeustenkäsittelyn voisi rakentaa seuraavan rungon ympärille:

```
try:
    # Tarkkailtava(t) operaatio(t), tyypillisesti input, jakolasku tms.
except ValueError:
    # Poikkeustenkäsittelijä poikkeukselle ValueError
except KeyboardInterrupt:
    # Poikkeustenkäsittelijä poikkeukselle KeyboardInterrupt
except Exception:
    # Poikkeustenkäsittelijä muille poikkeuksille
```

`try`...`except`-rakenteeseen voidaan liittää myös `else`-osio, joka ajetaan siinä tapauksessa, ettei yhtäkään virhettä synny:

Esimerkki 9.3. else-rakenne

```
# 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

Entäpä jos haluamme 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 9.4. Finally-osio

```
# Tiedosto: finally.py
import sys
import time

try:
    tiedosto = open("uutinen.txt", "r", encoding="utf-8")
    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. Jos keskeytämme ajon `KeyboardInterrupt`illa (`Ctrl-C`), saamme normaalin tulkin virheilmoituksen.

Huomionarvoista on kuitenkin se, että juuri ennen virheen esiintymistä suoritamme `finally`-osion ja suljemme 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. Huomaa, että `finally`-osuus suoritetaan **aina**, myös vaikka virhettä ei sattuisikaan ja että `sys.exit(-1)` palauttaa kutsuvalle ohjelmalle eli käyttöjärjestelmälle virhekoodin -1. Palautettavat virhekoodit mahdollistavat tiedon siirtämisen kutsuvaan ohjelmaan, mutta niiden suhteen kannattaa olla tarkkana – esim. Viope odottaa aina paluuarvoa 0 onnistuneen lopetuksen merkiksi. Ja koska `finally`-suoritetaan aina, ei `try`-osassa tarvitse olla erikseen tiedoston sulkemista, kuten normaalissa `except`-osiossa pitäisi.

Kuinka poikkeustenkäsittely kannattaa toteuttaa?

Tässä vaiheessa poikkeustenkäsittelyn kokonaiskuva rupeaa pikkuhiljaa hahmottumaan ja samalla sen aiheuttama työmäärä saattaa ruveta mietityttämään. `try-except`-rakenne rupeaa muistuttamaan valintarakennetta monine haaroineen ja `else`-osioineen, olemme tutustuneet jo puoleen tusinaa erilaisia poikkeuksia kuten `Exception`, `OSError`, `ValueError`, `TypeError`, `NameError`, `IndexError`, `KeyboardInterrupt` ja `SystemExit`, kun Pythonin koko poikkeushierarkia sisältää 64 poikkeusta. Herää siis kysymys, että mahtuuko ohjelmaan enää mitään muuta kuin

poikkeustenkäsittelyä? Tässä kohdin on hyvä muistaa, että kyseessä on *poikkeusten* käsittely ja etenkin pienissä itselle tehdyissä ohjelmissa sen voi jättää sivurooliin. Kaupallisissa ohjelmistoissa tilanne on toinen ja esim. tuhansien pankki- ja tankkausautomaattien pitäminen toiminnassa keskeytyksettä 24/7 on tavoite, johon kannattaa panostaa oikeasti ja toteuttaa poikkeustilanteiden käsittely kattavasti.

Tämän oppaan tavoite on auttaa ymmärtämään tarve poikkeustenkäsittelylle sekä osata toteuttaa poikkeustenkäsittely muutamien tärkeimpien käskyjen kohdalla. Käytännössä tämä tarkoittaa poikkeustenkäsittelyn toteuttamista aina tiedostonkäsittelyn yhteydessä siten, että käyttäjä saa tietää ongelmasta sekä siitä, missä se tapahtui ennen kuin ohjelma lopetetaan hallitusti. Näin käyttäjällä on realistinen mahdollisuus lähteä selvittämään ongelman syitä ja yrittää eliminoida ne. Tämän oppaan poikkeustenkäsittelyn tavoitetaso näkyy kokoavassa esimerkissä 9.5 alla ja se perustuu seuraaviin lähtökohtiin:

1. Poikkeustenkäsittely on toteutettava aina tiedostonkäsittelyn yhteydessä.
2. Poikkeustenkäsittelijä sijoitetaan lukemisen/kirjoittamisen tekevään aliohjelmaan.
3. Poikkeustenkäsittelijän sisällä on kaikki tiedostonkäsittelyyn liittyvät käskyt alkaen `open`-käskystä ja päättyen `close`-käskyyn eikä näiden välissä ole tiedoston käsittelyyn liittymättömiä käskyjä.
4. Poikkeustenkäsittelijä käsittelee `Exception`-poikkeuksen, ilmoittaa minkä nimisen tiedoston käsittely epäonnistui ja lopettaa ohjelman suorituksen `sys.exit(0)` -käskyllä.

Todellisuudessa poikkeustenkäsittelyn laajuus ja tarkkuus tulee miettiä lähtien liikkeelle ohjelman käyttötarkoituksesta ottaen huomioon käytettävissä olevat resurssit.

Osaamistavoitteet

Ymmärrät virheiden ennalta estämisen idean ja miksi tietyissä tapauksissa ohjelmissa tarvitaan poikkeustenkäsittelyä. Ymmärrät mikä ero on `try...except` ja `try...finally` –rakenteilla sekä tunnistat puolen tusinaa yleisintä poikkeusta. Osaat toteuttaa poikkeustenkäsittelyn tiedostonkäsittelyn yhteydessä `try... except` –rakenteella ja sinulla on hyvät lähtökohdat lähteä suunnittelemaan ja toteuttamaan kattavaa poikkeustenkäsittelyä Pythonilla tehtyihin perusohjelmiin.

Luvun asiat kokoava esimerkki

Esimerkki 9.5. Turvallista tiedostonkäsittelyä

```
# Tiedosto: try_try.py
# Poikkeustenkäsittely: tiedoston avaaminen, lukeminen ja kirjoittaminen

import sys

def lueTiedosto(nimi):
    print("Luetaan tiedosto '{0:s}'.".format(nimi))
    try:
        tiedosto = open(nimi, "r")

        while (True):
            rivi = tiedosto.readline()
            if (len(rivi) == 0):
                break
            print(rivi[:-1])

        tiedosto.close()
    except Exception:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(nimi))
        sys.exit(0)

    print("Tiedoston lukeminen onnistui.")
    return None

def kirjoitaTiedosto(nimi): # 1 kokonaisluku per rivi, lopussa tyhjä rivi
    print("Kirjoitetaan tiedosto '{0:s}'.".format(nimi))
    try:
        tiedosto = open(nimi, "w")

        for i in range(10):
            tiedosto.write(str(i)+'\n')

        tiedosto.close()
    except Exception:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(nimi))
        sys.exit(0)

    print("Tiedoston kirjoittaminen onnistui.")
    return None

def paaohjelma():
    nimi = "L09.txt"
    kirjoitaTiedosto(nimi)
    lueTiedosto(nimi)
    print("Kiitos ohjelman käytöstä.")
    return None

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

Luku 10: Data analytiikka ja uusia tietorakenteita

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

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

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

Sanakirja (eng. Dictionary)

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

Sanakirjan märittely muistuttaa muita sarjamuuttujia:

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

Keskeisiä eroja ovat kaksoispiste avainten ja arvojen välissä, tietueet eli avain-arvo -parit erotellaan toisistaan pilkuilla, ja sanakirja eli kaikki parit merkitään aaltosulkeiden {} väliin kun tuple käytti kaarisulkeita () ja lista hakasulkuja []. Avaimet eivät ole missään tietyssä järjestyksessä sanakirjan sisällä vaan jos haluat käyttää jotain tiettyä järjestystä, on sinun huolehdittava siitä jo alustusvaiheessa sekä ylläpidettävä tätä järjestystä käsin. Vaihtoehtona on myös käyttää collections-kirjastosta löytyvää sanakirjan kehittyneempää muotoa, OrderedDict, joka säilyttää siihen laitettujen alkioiden järjestyksen. Sanakirjan käyttö esitellään Esimerkki 10.1:ssä.

Esimerkki 10.1 Sanakirjan käyttäminen

```
# Tiedosto: sanakirja.py

# Sanakirjan indeksi voidaan esittää näin; sulkujen sisällä heittojen
# välissä välilyönti ei ole merkitsevä merkki.

sahkoposti = {"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", sahkoposti["167-671"])

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

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

for avain in sahkoposti:
    print("Veljeksen {0} sähköposti on {1}".format(avain, sahkoposti[avain]))

if ("176-761" in sahkoposti):
    print("\n176-761:n osoite on {0}".format(sahkoposti["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. Sanakirjasta haetaan arvo siihen liittyvällä avaimella, esim. yllä ‘167-671’, ja sanakirjaan lisätään arvo sijoittamalla uuteen avaimeen liittyvä arvo, esim. yllä ‘617-716’. 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 johtuen niiden erilaisesta rakenteesta. Mutta ei hätää, Pythonista löytyy `sorted`-funktio, joka hoitaa homman molemmille.

Esimerkki 10.2 Tuplen järjestäminen

```
# 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 käsittelyn jälkeen. `sorted` toimii siis vähän kuten `len`, eli se palauttaa jotain, joka on tässä tapauksessa järjestyksessä olevat numerot. On syytä huomata, että `sorted` ei tee muutoksia alkuperäiseen muuttujaan. Jos muutokset halutaan saada talteen muuttujaan, kannattaa lajittelun tulos tallentaa uuteen muuttujaan:

```
lajiteltu = sorted(tietorakenne)
# Vertaa listaan, alla oleva muuttaa lista:n järjestyksen
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 kentän mukaan (indeksillä nolla). 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 Pythonin tapa ilmaista *anonyymi funktio*, mutta tästä sinun ei tarvitse tietää tämän enempää. Asiaan palataan myöhemmillä ohjelmointikursseilla. Riittää, että hahmotat kuinka `opiskelijat` tuple/sanakirja lajitellaan tietyn kentän mukaan. Alla on vielä toinen esimerkki asiaan liittyen, jossa sanakirja lajitellaan ensin avaimen mukaan ja sitten arvon mukaan.

Esimerkki 10.3 Lisää järjestelyä

```
# lajittelu.py
autot = {2019:23, 2017:31, 2016:42, 2018:38, 2015:29}

print("Järjestettynä vuosiluvun mukaan nousevasti.")
lajiteltu = sorted(autot)
print("Vuosi: Autoja")
for vuosi in lajiteltu:
    print("{0:d}: {1:d}".format(vuosi, autot[vuosi]))

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

Tuloste

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

Kuinka se toimii

Sanakirjan järjestäminen tapahtuu `sorted`-funktiolla. `sorted`-funktio tekee uuden listan, jonka se laittaa haluttuun järjestykseen ja alkuperäinen sanakirja jää koskemattomaksi. Sanakirjan lajittelu avaimen mukaan on suoraviivaista eli `sorted`-funktiolle riittää välittää parametrina pelkästään lajiteltava sanakirja. Toisessa esimerkissä lajittelu tehdään arvojen perusteella, jolloin sanakirja jaetaan osiin `items`-funktiolla ja `key`-parametrilla määritetään lajiteltava tieto. Lajiteltava tieto on `lambda`-funktio `autot` ja oleellinen tieto on hakasuluissa oleva luku 1 eli lajitellaan listan toisen alkion perusteella. Jos arvon 1 tilalle laittaa arvon 0, tapahtuu lajittelu avaimen perusteella (0. alkio listassa).

Tietojen käsittelyn kannalta lajittelu on yksi keskeinen toimenpide ja esimerkiksi taulukkolaskenta-ohjelmat tarjoavat siihen paljon erilaisia vaihtoehtoja. Myös Pythonissa lajittelua voi tehdä monella tapaa edellä esitettyjen `sorted` ja `lambda`-funktioiden avulla. Ne eivät kuitenkaan ole perusasioita vaan lähtökohta omaan maailmaansa, johon voi tutustua tarkemmin esimerkiksi Python dokumentaation ”Sorting HOW TO” –osuuden avulla. Lähtökohtaisesti tämän oppaan taso on listojen lajittelu `sort`-jäsenfunktion avulla.

Huomioita sarjallisten muuttujien vertailusta

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

Yksinkertaisimmassa tapauksessa 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 lyhempi. Tämän takia esimerkiksi isot kirjaimet ovat aina pieniä kirjaimia arvollisesti pienempiä. Alla olevasta taulukosta näet joitain esimerkkejä näiden sääntöjen käytännön tulkinnoista; kaikki vertailut tuottavat tuloksen `True`:

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

Eri tietotyyppien vertailu keskenään ei ole sallittua. Listaa ei siis voida verrata kokonaislukuun, eikä merkkijonoa 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.

Matriisi eli numpy-moduulin taulukko

numpy-moduuli on numeerisen laskennan laajennus Pythoniin eli Numerical Python –paketti, joka tarjoaa työkaluja isojen moniulotteisten taulukoiden ja matriisien käsittelyyn. Tässä oppaassa katsomme vain 2-ulotteista taulukkoa, matriisia, joka on tyypillinen tapa esittää dataa taulukkolaskentaohjelmien riveinä ja sarakkeina. Pythonissa ei ole omaa matriisi-tietotyyppiä, mutta vastaavan rakenteen pystyy tekemään lisäämällä listoja listaan. numpy-moduulin asennus käytiin läpi luvussa 8 ja seuraavien ohjelmien suorittaminen edellyttää sitä.

Esimerkki 10.4 numpy-kirjaston matriisin peruskäyttö

```
# Tiedosto: numpy_perusteet.py
import numpy
RIVEJA = 3
SARAKKEITA = 3

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

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

# Matriisin alkioiden käsittely indekseillä alustamisen jälkeen
for rivi in range(RIVEJA):
    for sarake in range(SARAKKEITA):
        matriisiB[rivi][sarake] = rivi+sarake

# Laskentaa matriiseilla numpyn avulla
matriisiC = matriisiA + matriisiB

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

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

Tuloste

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

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

Kuinka se toimii

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

numpy tarjoaa valmiita työkaluja matriisien käsittelyyn, joista tässä näkyy vain yhteenlasku sekä tulostus. Lopussa oleva `delete()`-funktio mahdollistaa matriisin alkioiden, rivien ja sarakkeiden poistamisen eikä sitä tarvitse käyttää ohjelman lopussa automaattisen muistinhallinnan vuoksi. Yllä matriisi tulostetaan yhdellä print-käskyllä numpyn toteutuksen mukaisesti ja lisäksi matriisin tiedot tulostetaan puolipisteillä eroteltuna, jolloin data on helppo kopioida esim. Exceliin visualisointia varten.

Osaamistavoitteet

Tässä luvussa tutustuimme kahteen uuteen tietorakenteeseen, jotka olivat sanakirja, dictionary, ja matriisi eli numpy-moduulin moniulotteinen taulukko. Lisäksi tutustuimme sanakirjan ja tuplen lajitteluun, joka on listan `sort`-jäsenfunktiolla tapahtuvaa lajittelua haastavampi tehtävä tuplen ja sanakirjan sisäisen rakenteen vuoksi. Käytännössä Python hoitaa lajittelun, kunhan käytössä on oikeat operaatiot. Laajasti ottaen tämän luvun asiat liittyvät data analytiikkaan ja ne luovat pohjaa datan analysoinnille käsillä olevan tarpeen mukaisesti.

Luvun asiat kokoava esimerkki seuraavalla sivulla

Esimerkki 10.5. Moniulotteista ohjelmointia

```
# (c) LUT 2020 L10numpy.py un Kirjastot, kiintoarvot, luokat
# Tämä esimerkki on tarkoitettu omatoimisen oppimisen tueksi ohjelmoinnin
# opiskeluun. Muu käyttö kielletty.
##########################################################################
# Datatiedoston rivi: ”21-01-2017;3001;14624;10,81;16;507;346;70;26;1826”

import datetime # Kirjastojen import tiedoston alussa
import numpy

KUUKAUSIA = 12 # Kiintoarvot
PAIVIA = 7

class DATA: # Luokat
    pvm = ""
    askelia = 0

def lue(nimi, lista):
    try:
        tiedosto = open(nimi, "r")
        while (True): # Tiedoston lukeminen ja tallennus olioina listaan
            rivi = tiedosto.readline()
            if (len(rivi) == 0):
                break
            sarakkeet = rivi[:-1].split(';')
            data = DATA()
            data.pvm = datetime.datetime.strptime(sarakkeet[0], "%d-%m-%Y")
            data.askelia = int(sarakkeet[2]) # askeltiedot 2. sarake
            lista.append(data)
        tiedosto.close()
    except Exception:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(nimi))
        sys.exit(0)
    return lista

def analysoi(lista, matriisi):
    for alkio in lista: # Analyysi: tiedot listalta matriisiin
        kk = alkio.pvm.month - 1 # month-arvot ovat 1-12
        pv = alkio.pvm.weekday() # weekday() palauttaa 0-6
        matriisi[kk][pv] += alkio.askelia
    return matriisi

def tulosta(matriisi):
    for pv in range(PAIVIA): # Otsikkorivin tulostus
        print(';' + str(pv+1), end='')
    print(';')

    for kk in range(KUUKAUSIA): # Tulokset excel-visualisointia varten
        print(kk+1, end=';')
        for pv in range(PAIVIA):
            print(matriisi[kk][pv], end=';')
        print()
    return None

def paaohjelma():
    lista = [] # Alustuksia ja alkutoimenpiteitä
    matriisi = numpy.zeros((KUUKAUSIA, PAIVIA), int)
    nimi = input("Anna luettavan datatiedoston nimi: ")
    lista = lue(nimi, lista)
    matriisi = analysoi(lista, matriisi)
    tulosta(matriisi)
    return None

paaohjelma()
# eof
```

Luku 11: Algoritmi, pseudokoodi ja rekursio

Usein ohjelmointimaailmassa on tapana aloittaa ohjelmointi puoliohjelmoinnilla, joka ei varsinaisesti ole mikään varsinainen ohjelmointikieli, mutta soveltuu kuvaamaan niistä useimpia. Tällaista kuvausta sanotaan usein pseudokoodiksi. Lisäksi tietynlainen tapa ratkaista ongelma, kuten esimerkiksi alkioiden järjestely tai suurten kokonaislukujen kertolasku, on niin spesifi, 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. Ja yksi tyypillinen esimerkki algoritmista on rekursiivinen ohjelma, jonka erityispiirteet on syytä huomioida ohjelmia tehdessä.

Pseudokielet ovat ainutlaatuisia siksi, että niiltä puuttuu kokonaan kielioppi. Ne eivät varsinaisesti ole humaanisia kirjakieliä kuten esimerkiksi englanti, saksa tai suomi saatikka 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 – esim. säiliöitä ja askeltajia – yksinkertaistamaan esitystä.

Algoritmeista ja pseudokoodeista löytyy lisätietoja mm. Wikipediasta hakusanoilla ”algoritmi” ja ”pseudokoodi”. Moneen ongelmaan myös Google tarjoaa 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 Lisayslajittelu(lista t)
   n = listan t alkioiden lukumäärä

   # Käydään taulukon jokainen alkio läpi ensimmäistä lukuun ottamatta
   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))
         # Tässä vaihdetaan alkioiden t[j] ja t[j-1] arvot keskenään
         j = j – 1 # while-rakenteen edellyttämä indeksin muutos
       end while
   end for
```

Mutta kuinka tästä pääsee 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 käydä läpi listaa sekä löytää tiedolle uuden paikan, 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.

Pseudokoodi auttaa ohjelman toiminnan hahmottamisessa

Yleisesti ottaen pseudokoodilla kirjoitettujen algoritmien lukemisen osaaminen helpottaa ohjelmointia myös vaikeita asioita toteutettaessa. Tämän taidon, niin kuin monet muutkin ohjelmoinnin taidot, oppii ainoastaan harjoittelemalla. Pseudokielen luku- ja kirjoitustaito on asia, joka helpottaa lähdekoodin suunnittelua huomattavasti sekä mahdollistaa ei-natiivikielisten esimerkkien hyödyntämisen.

Käytännön ongelmasta ohjelmaksi

Ensimmäinen esimerkkimme oli aika suoraviivainen, koska lähtökohtanamme oli valmis pseudokoodi. Toisessa algoritmiesimerkissä 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 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):
    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

```
# 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... Kenties tehtävänannossa oli jopa määrittelyvirhe?

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*.

Rekursiivinen ohjelma

Rekursio on tehokas tapa ratkaista tiettyjä ohjelmointitehtäviä. Tässä vaiheessa opasta tehdyissä ohjelmissa rupeaa näkymään rekursiivisia ratkaisuja, joten käymme rekursion perusteet ja oikean tavan sen toteuttamiseen tässä algoritmien yhteydessä. Rekursio ei ole tämän oppaan ydinasioihin ja rekursiota ei käytetä muulla kuin tässä luvussa.

Erään määritelmän mukaan ”Rekursiivinen algoritmi on algoritmi, jonka toiminta perustuu rekursion käyttöön.” Vaikka tällainen määritelmä on tyypillisesti hyödytön, sopii se rekursioon oikein hyvin, silllä Tieteen termipankin määritelmän mukaan ”[rekursio on] kielen ominaisuus, että sama rakenne voidaan toistaa periaatteessa rajattoman monta kertaa.” Puuttumatta kielitieteeseen sen enempää matematiikassa rekursio mahdollistaa funktion määrittelyn niin, että ”funktion arvo tietyssä pisteessä riippuu funktion arvosta edellisessä pisteessä.” Käytännössä kertoman laskenta on tyypillinen esimerkki rekursiosta ja alla oleva kaava esittää luvun *n* kertoman laskentakaavan matemaattikkojen esitystavalla:

$$n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$$

Näin ollen esimerkiksi luvun 5 kertoma lasketaan seuraavalla tavalla:

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

Tämä esitystapa näyttää jo tutulta ja kertoma voidaankin laskea helposti esimerkin 11.2 mukaisesti toistorakenteella.

Esimerkki 11.2. Kertoman laskenta silmukalla

```
# Kertoman laskenta while-toistorakenteen avulla

kertoma = 1
alku = 5
n = alku
while (n > 0):
    kertoma = kertoma * n
    n = n - 1
print("Luvun {0:d} kertoma on {1:d}.".format(alku, kertoma))
```

Näin saimme laskettua kertoman, mutta tässä ei näy rekursiota. Yritetään uudestaan ja nyt kertoma pitää laskea rekursiivisella algoritmilla. Tieteen termipankin mukaan toistoja voi olla rajattoman monta, mutta ohjelmoija alkaa tällöin epäillä ikisilmukkaa ja siksi ohjelmassa pitää olla kaksi vaihtoehtoa:

1. Saman ohjelman kutsuminen uudestaan eli rekursio.
2. Ohjelman lopettaminen hallitusti eli rekursion päättäminen.

Ohjelman itsensä kutsuminen tarkoittaa sitä, että ohjelman sisällä on saman aliohjelman kutsu. Tämä kuulostaa oudolta ja siksi tämä ohjelmatyyppi on saanut oman nimen – rekursiivinen ohjelma. Rekursiivinen aliohjelma kertoman laskemiseksi näkyy esimerkissä 11.3.

Esimerkki 11.3. Rekursiivinen aliohjelma

```
# Rekursiivinen aliohjelma eli aliohjelma kutsuu itseään

def kertoma(x):
    if (x > 0):                    # Lopetusehtoa ei saavutettu vielä
        return (x * kertoma(x-1)) # Kutsuu itseään, parametri muuttuu
    else:            # lopetuksen paluuarvo, tyypillisesti kiinteä arvo
        return 1

def paaohjelma():
    luku = int(input("Minkä luvun kertoman haluat laskea: "))
    print(kertoma(luku)) # ensimmäinen rekursiivisen ohjelman kutsu
    return None

paaohjelma()
```

Esimerkissä 11.3 näkyy, miten `kertoma`-aliohjelma kutsuu valintarakenteen ensimmäisessä haarassa itseään ja toisessa haarassa ohjelman suoritus lopetetaan palauttamalla luku 1. Keskeinen asia on, että kutsuttaessa aliohjelmaa uudestaan sitä ei kutsuta samalla arvolla vaan **parametrin arvo on muuttunut**. Toisella kierroksella parametrin arvo on yhden pienempi, kolmannella kierroksella kaksi pienempi jne. kunnes parametrin arvo on lopulta 0 ja rekursiivisen kutsun sijaan ohjelma palauttaa arvon 1. Tämän seurauksena saadaan suoritettua haluttu kertolasku, 1 * 2 * 3 * … * n, ja saadaan laskettua n:n kertoma n!. Itse laskenta tapahtuu samalla tavalla esimerkkien 11.2 toistorakenteella ja 11.3 rekursiivisella aliohjelmalla, mutta ohjelmien rakenteet ovat erilaisia.

Rekursion osalta esimerkissä 11.3 on suora rekursio eli aliohjelma kutsuu itse itseään suoraan. Toinen vaihtoehto on epäsuora rekursio, jolloin tyypillisesti kaksi aliohjelmaa kutsuu toisiaan vuorotellen. Suora rekursio on tyypillisesti harkinnan ja suunnittelun tulos, kun taas epäsuoraan rekursioon päädytään usein huomaamatta ja ohjelman suorituskyky kärsii kun ohjelma käyttää aikaa aliohjelmakutsujen ketjuttamiseen.

Tärkein syy rekursiivisen aliohjelman läpikäyntiin tässä oppaassa on pystyä välttämään hallitsematon rekursio. Oikein toteutettuna rekursiivinen algoritmi on tyypillisesti lyhyt ja helppo ymmärtää kokeneelle ohjelmoijalla, mutta kokemattomat ohjelmoijat päätyvät usein rekursioon sitä suunnittelematta ja tällöin ohjelman suorituskyky voi kärsiä. Rekursion keskeisimpiä asioita on sen hallittu lopettaminen. Todellisuudessa tämä pätee kaikkiin aliohjelmiin eli kaikki aliohjelmat tulee lopettaa `return` -käskyyn tyypillisesti paluuarvon kanssa. Näin paluu aliohjelmasta tapahtuu kutsun jälkeiseen kohtaan, aliohjelman varaamat resurssit palautetaan käyttöjärjestelmän käyttöön ja aliohjelman suoritus päättyy suunnitellusti. Tyypillisesti rekursio on keskein aihe Tietorakenteet ja algoritmit -kursseilla.

Osaamistavoitteet

Ymmärrät pseudokoodin käsitteen ja osaat muuntaa ongelman algoritmiksi sekä algoritmin koodiksi. Ymmärrät mitä rekursio tarkoittaa ja pystyt välttämään rekursion käytön silloin kun se ei ole järkevää.

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? Asian selvittämiseksi tutustumme tässä kappaleessa bitteihin ja tiedon esittämiseen erilaisten merkkitaulukoiden avulla. Lisäksi esittelemme vaihtoehdon tekstimuotoiselle tiedon tallentamiselle, binaaritiedostot, katsomme miksi tietokone ei aina pyöristä lukuja niin kuin matematiikan opettaja opetti ja miten tietokone käsittelee sisäisesti aika-tietoa kuten päivämääriä.

Tietokoneen sisällä kaikki tieto käsitellään ja tallennetaan binaariarvoina, eli kaksikantalukuina, jotka saavat arvoja 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 jonoihin (eli kahdeksan nollaa tai ykköstä, esim. 10001001), jota sanomme tavuksi. Yksi tavu taas on 2-kantainen esitysmuoto numeroarvolle väliltä 0-255. Kuulostaa hienolta, mutta katsotaan kuitenkin esimerkin avulla, kuinka noista bittijonoista, ts. binaariluvuista, saadaan meille tuttuja 10-kantaisia kokonaislukuja.

Esimerkki 12.1. Bittilukujen laskeminen, binaariluvusta kokonaisluvuksi

```
# Tiedosto: bin2int.py
def bittiluku():
    bittijono = input("Anna binaariluku: ")

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

    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 10-kantaisena", tulos)
    return None

bittiluku()
```

Tuloste

```
>>>
Anna binääriluku: 1101
Bittijonosi on 4 bittiä pitkä.
Bittijonosi on 10-kantaisena 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 10-kantaisen esityksen bittiarvolle 101101, saisimme seuraavan 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 = **45**					

Toisaalta, voimme myös laskea bittiesityksen kokonaisluvuille seuraavalla ohjelmalla:

Esimerkki 12.2. Bittilukujen laskeminen, kokonaisluvusta binaariluvuksi

```
# 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 10-kantainen luku on 2-kantaisena", tulos)
```

Tuloste

```
>>>
Anna kokonaisluku: 74
Antamasi 10-kantainen luku on 2-kantaisena 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 sen hetkisen 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 eli merkkien 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.

Taulukko 12.1 ASCII-taulukko

Numero	**ASCII-merkki**	**Numero**	**ASCII-merkki**	**Numero**	**ASCII-merkki**
32	välilyönti	64	@	96	`
33	!	65	A	97	a
34	”	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Yläpuolella olevaan taulukkoon on kerätty alkuperäisen ASCII-taulukon numerot ja niitä vastaavat merkit. Ensimmäiset 32 merkkiä (0-31) on varattu ohjauskoodeille ja muille ei-näkyville merkeille. Jos luemme levyltä bittisarjan 001000001, tarkoittaa se kokonaislukuna arvoa 65. Jos taas tulkitsemme tämän numeroarvon ASCII-taulukon avulla, huomaamme lukeneemme tiedostosta ison A-kirjaimen. Jos tiedostossa vastaavasti lukisi ”01011100 01110101 01010100 01101111”, voitaisiin se lukea arvoina ”97 117 84 111”, eli ”auTo”.

Meitä suomalaisia ajatellen alkuperäisessä 7 bitistä, eli 128 merkistä, koostuvassa ASCII-taulukossa on yksi suuri ongelma – se ei tunne skandinaavisia merkkejä. Alkuperäinen ASCII-taulukko suunniteltiin nimenomaisesti englanninkieliselle aakkostolle, joten alkuperäinen ratkaisu ei sisältänyt esim. Ä, Ö eikä Å -kirjaimia. Tämän vuoksi ASCII-taulukkoa on myöhemmin jouduttu laajentamaan 256 merkkiin (8 bittiä) sekä luomaan joukko uusia taulukoita. Lisäksi käyttöjärjestelmiin on toteutettu mahdollisuus valita, mitä merkkitaulukkoa ohjelmien lukemisessa ja tulkitsemisessa käytetään.

Modernissa tietotekniikassa siirrytään kohti universaaleja merkkitaulukoita, joista tärkein on 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 – jos ei pysyvästi – niin hetkeksi ainakin (ainahan ulkoavaruudesta voi löytyä miljoonia uusia kieliä...). 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 varmempi käyttää merkkitaulukkoa 1252, jolloin ohjelma alkoi alla olevalla määrityksellä. Nykyisissä versioissa on toimiva UTF-8 koodaus eikä alla olevan tyylisiä määrityksiä enää tarvita.

```
# -*- 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)
'푑'
>>>
```

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

Binaaritiedostot eli sananen 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 12.3. pickle-moduuli, binääritallennus

```
# Tiedosto: p_dump.py

import pickle
import sys

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

    pickle.dump(lista, tiedosto)
    tiedosto.close()
except Exception:
    print("Tiedoston käsittelyssä virhe, lopetetaan.")
    sys.exit(0)
```

Koska pickle-moduuli tallentaa tiedon binaarimuotoisena, 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. Tietokone pystyy kuitenkin käsittelemään sitä tehokkaasti.

Esimerkki 12.4. pickle-moduuli, tallenteen lataaminen

```
# Tiedosto: p_dump.py

import pickle
import sys

try:
    tiedosto = open("testi.data","rb")
    luettu = pickle.load(tiedosto)
    tiedosto.close()
except Exception:
    print("Tiedoston käsittelyssä virhe, lopetetaan.")
    sys.exit(0)

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 ne pitää ensin muuttaa merkkijonoiksi. `pickle`-moduuli osaakin käsitellä kaikkia tässä oppaassa esiteltyjä tietomuotoja. Kannattaa myös muistaa, että `pickle`ä käyttäessämme joudumme avaaman tiedoston binääritilaan avauskytkimillä `"rb"` (read-binary) ja `"wb"` (write-binary).

Pari sanaa pyöristämisestä

Olet varmaan tähän mennessä huomannut, ettei Python käsittele liukulukuja aina samalla tavalla kuin sinä. Lähtökohtana on, että tietokoneen muistiin ei mahdu numeroita äärettömällä tarkkuudella ja siksi desimaaliosia sisältävillä numeroilla laskennassa tulee joskus vastaan ”hassuja” tilanteita kuten seuraava:

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

Tämä johtuu siitä, että liukuluvut on määritelty tietyillä standardeilla (yleensä IEEE-754) ja niitä käsitellään sovittujen sääntöjen mukaan. Tästä löytyy tarkempaa tietoa esimerkiksi Wikipediasta hakusanalla ”liukuluku” ja englanninkielinen artikkeli on suomenkielistä laajempi.

Toinen mielenkiintoinen asia on pyöristäminen etenkin kun meille suomalaisille on itsestään selvää, että vitonen pyöristyy ylöspäin. Tämä on selkeä toimintapa, mutta tuo mukanaan ongelmia. Ajatellaanpa seuraavaa: Kahvilassa on myynnissä pulla hintaan 2,5€ja kakkupala hintaan 3,5€ Erittäin vanha kirjanpitojärjestelmä osaa laskea vain kokonaisluvuilla, joten se pyöristää aina numerot kokonaisluvuksi. Näin ollen kun eräänä päivänä saadaan myytyä 10 pullaa ja 10 kakkupalaa, on kirjapidossa tulona 70€mutta kassassa on vain 60€rahaa. Tämä ei kuulosta hyvältä.

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 kannattaa huomata, että maailmassa on peruskoululaitoksia, joissa 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 esimerkiksi Wikipediasta hakusanalla ”pyöristäminen”.

Ajan esitystapa tietokoneessa

Muistat varmaan datetime-moduulin ja miten sen avulla saadaan selville tietokoneen systeemin kellonaika, jotta aika voidaan esittää ymmärrettävässä muodossa esim. päivämääränä ja kellonaikana? Sehän onnistui esim. näin:

```
>>> datetime.now().strftime("Tänään on %d.%m.%Y ja kello on %H:%M.")
'Tänään on 21.04.2010 ja kello on 12:43.'
```

Tämä on siis tuttua asiaa, mutta tietokoneen ratkaisu ajan käsittelyyn jäi tuolloin selvittämättä. Vastaus on itse asiassa triviaali, sillä tietokoneissa aikaa käsitellään numeroina ja isompi numero tarkoittaa pidempää aikaa lähtökohdasta ja pienempi aika lyhyempää aikaa. Unix-puolella lähtökohtana on Unix Epoch -aika, joka on 1.1.1970 klo 00:00:00 eli aikaa esittävän reaaliluvun arvo on tällöin Unix-järjestelmässä 0 (nolla). Luku kasvaa siitä sekunti kerrallaan ja jos halutaan käsitellä tuota ajankohtaa aiempia tapahtumia, ovat luvut silloin negatiivisia. Tämä kuulostaa insinööriratkaisulta, mutta näin on ollut jo noin 50 vuotta, joten ratkaisua voi väittää toimivaksi.

Unix Epoch aika ei ole ainoa tapa ajan esittämiseen ja esim. MS-Windows käyttää nykyään NT time epoch'ia, jossa lähtökohtana on 1.1.1601. Tällä erolla ei ole merkitystä niin kauan kuin ohjelmissa ei viitata suoraan näihin aika-numeroihin vaan pyydetään aika yllä olevalla tavalla järjestelmäpalveluista, jotka palauttavat ajan oikeassa muodossa. Oman järjestelmän 0-hetken voit selvittää seuraavalla käskyllä:

```
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

Asiaa tarkemmin pohtiessa on selvää, että meillä on ongelma 19.1.2038 kun 32 bittisissä Unix-järjestelmissä tapahtuu ylivuoto aikamuuttujan kohdalla. Tämä tilanne vastaa vuoden 2000 tapahtumia niissä ohjelmistoissa, joissa vuosi esitettiin 2 numeron tarkkuudella. Eli vuodesta (19)99 tuli tuolloin (20)00 ja kun 00 on pienempi kuin 99 niin ohjelmistoissa saattoi tapahtua odottamattomia asioita. Onneksi useimmat Unix-pohjaiset järjestelmät ovat olleet jo pitkään 64-bittisiä, joten tätä ongelmaa joudutaan odottamaan vielä muutama vuosi. Vai onkohan jossain vielä 32 bittisiä Unix-ohjelmistoja käytössä?

Osaamistavoitteet

Ymmärrät kuinka tieto on tallennettu tietokoneen muistiin ja kuinka se muutetaan ihmisen ymmärtämään muotoon. Ymmärrät kuinka `pickle`-moduulin avulla voidaan käsitellä binaaridataa. Ymmärrät liukulukuihin ja pyöristämisen liittyviä erilaisia haasteita ohjelmoinnissa. Lisäksi ymmärrät tietokoneen tavan käsitellä aikaa numeroina ja siitä seuraavat haasteet.

Luku 13: Käyttöliittymistä

Tietokoneiden ohjelmointia tehtiin vielä 1960 luvulla reikäkorttien avulla. Reikäkorttiin perustuva automaattinen tietojenkäsittely kehitettiin Yhdysvaltain väestölaskentaa varten 1890 kutomakoneiden reikäkorttiohjauksen pohjalta. Automaattinen tietojenkäsittely näytti siinä määrin lupaavalta alalta, että sitä varten perustettiin yhtiö, jonka nimi oli vuodesta 1924 alkaen IBM. Alkuaikoina reikäkortteja käytettiin tiedon koodaamiseen eli tieto, numerot ja merkit, lävistettiin kartongista valmistettuun pahviin reikinä, joiden paikka määritteli halutun tiedon (http://suomentietokonemuseo.fi/vanhat/fin/laite_fin.htm). Reikäkortteja käytettiin siis alussa kutomakoneiden ohjaukseen ja tiedon tallettamiseen, mutta myöhemmin myös tietokoneohjelmia koodattiin reikäkorteilla.

Siirtyminen reikäkorteista ohjelmien koodaamiseen näppäimistön avulla helpotti työtä merkittävästi. Tosin koodaaminen ilman näyttöä oli haastavaa ja se tehtiin rivieditorilla, jossa merkkejä voitiin syöttää yhdelle riville kerrallaan syöttömoodissa tai rivejä voitiin muokata kokonaisina riveinä komentomoodissa. Nykyään koko ohjelma on tyypillisesti nähtävillä näytöllä, mutta rivieditoreita löytyy vielä. Esimerkiksi Unix/Linux käyttöjärjestelmissä on tyypillisesti mukana rivieditori ed, josta kehitettiin näyttöjä hyödyntäviä editoreita kuten vi (visual editor) ja myöhemmin emacs.

Näppäimistön käyttö muutti ohjelmointia merkittävästi ja vaikka graafiset käyttöliittymät vallitsevat nykyään markkinoita, ohjelmat perustuvat vielä tyypillisesti tekstitiedostoihin. 1970-luvulla kehitetty Unix-järjestelmä perustuu merkkipohjaisiin työkaluohjelmiin, vaikka nykyään useimmat niistä ovat saaneet graafisia käyttöliittymiä tai niihin perustuvia seuraajia. Merkkipohjaisille käyttöliittymille on kuitenkin vielä oma käyttäjäkuntansa ja sulautetuissa järjestelmissä on usein paljon koodia ilman käyttäjälle näkyvää käyttöliittymää. Esimerkiksi auton lukkiutumattomat jarrujärjestelmät perustuvat ohjelmistoihin, vaikkei käyttäjä pääsekään vaikuttamaan niihin oman käyttöliittymän kautta.

Tässä luvussa esittelemme lyhyesti graafisten käyttöliittymien perusteet ja perinteisen komentorivikäyttöliittymän. Katsomme ensin miltä graafiseen käyttöliittymään perustuva ohjelma näyttää Pythonilla tehtynä. Sen jälkeen tutustumme lyhyesti komentorivikäyttöliittymään ja miten komentoriviparametrit saadaan hoidettua Pythonilla.

Graafisten käyttöliittymien perusteet

Nykyisin useimmat ohjelmat julkaistaan graafisiin käyttöliittymiin perustuvissa käyttöjärjestelmissä. 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 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 on luoda lähdekooditiedosto, joka tallennetaan ja ajetaan sellaisenaan tulkista.

Seuraavilla kahdella esimerkillä tutustumme siihen, kuinka yksinkertaisen graafisen ikkunan tekeminen onnistuu. Aloitetaan tekemällä pohja, johon käyttöliittymä rakennetaan.

Graafinen käyttöliittymä

Esimerkki 13.1. Perusikkuna

```
# Tiedosto: gui.py
from tkinter import *

pohja = Tk()
pohja.mainloop()
```

Esimerkkikoodin tuottama tulos

Kun ajamme esimerkin koodin, tuottaa tulkki alla olevan tyhjän käyttöliittymäikkunan:

Kuva 13.1: Esimerkin 13.1. tuottama käyttöliittymäikkuna. Ikkunan ulkoasu vaihtelee käyttöjärjestelmän ja ikkunamanagerin mukaan

Kuinka koodi toimii

Ensimmäinen esimerkkikoodi on varsin lyhyt eikä sisällä paljoakaan toimintoja. Ensimmäisellä rivillä otamme käyttöön Tkinter-kirjaston. Tällä kertaa sisällytys toteutetaan aiemmasta poikkeavalla `from...import *` -syntaksilla johtuen siitä, että se on tämän kirjaston yhteydessä näppärämpi ratkaisu.

Toisella rivillä luomme käyttöliittymän perustan tekemällä muuttujasta `pohja` Tk()-luokan juuri- eli pohjatason (`root`). Tähän pohjatasoon lisäämme jatkossa kaikki haluamamme komponentit ja toiminnot, mutta tällä kertaa jätämme sen tyhjäksi. Kolmannella rivillä käynnistämme Tkinter-käyttöliittymän kutsumalla pohjatasoa sen käynnistysfunktiolla `mainloop` ja ohjelma lähtee käyntiin tuottaen tyhjän ikkunan. Tyhjä ikkuna johtuu siitä, ettei ohjelman pohjatasolle ole sijoitettu muita komponentteja. Jos olisimme laittaneet sille vaikka painonapin, olisi ruutuun ilmestynyt 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 seuraavaksi kuinka pohjatasolle lisätään tekstikenttiä ja syöttölaatikoita. Huomaa, että seuraava esimerkki ei ole tämän kurssin ydinasioita, sillä toteutuksessa on käytetty olio-ohjelmointia, jossa on mukana luokan määrittely jäsenmuuttujineen ja -funktioineen. Tämä esimerkki on siis tarkoitettu niille, joita

asia kiinnostaa.

Esimerkki 13.2. Kehittyneempi graafinen ohjelma

```
# Tiedosto: tk_gui.py
from tkinter import *

class LASKIN():
    luku1 = ""
    luku2 = ""
    vastaus = ""
    def __init__(self):
        pohja = Tk()
        pohja.title('Laskin')
        Label(pohja, text='1. luku').pack()
        self.luku1 = Entry(pohja)
        self.luku1.pack()
        Label(pohja, text='2. luku').pack()
        self.luku2 = Entry(pohja)
        self.luku2.pack()
        Label(pohja,text='Summa').pack()
        self.vastaus = Entry(pohja)
        self.vastaus.pack()
        Button(pohja,  text='Summaa', command=self.summaa).pack()
    def summaa(self):
        eka = float(self.luku1.get())
        toka = float(self.luku2.get())
        summa = eka + toka
        self.vastaus.delete(0, END)
        self.vastaus.insert(0, str(summa))

kalkulaattori = LASKIN()
```

Kuva 13.2: Esimerkin 13.2 näytölle tuottama ikkuna eli Laskin-ohjelma

Esimerkkikoodin tuottama tulos

Kun ajamme esimerkin koodin, tuottaa tulkki kuvassa 13.2 näkyvän käyttöliittymäikkunan.

Kuinka koodi toimii

Tässä lähdekoodissa tuotamme ensimmäisen käyttöliittymän, joka tekee jotain. Alkuun suoritamme samanlaiset alustustoimenpiteet kuten aiemmin eli otamme käyttöön Tkinter-moduulin. Seuraavaksi määrittelemme luokan `LASKIN`, joka hoitaa niin laskimen toiminnallisuuden kuin graafisen käyttöliittymänkin.

Luokalla on kolme jäsenmuuttujaa: `luku1`, `luku2` ja `vastaus`. Aluksi nämä alustetaan tyhjiksi ja luokan alustusvaiheessa näihin sijoitetaan syöttölaatikot. Alustuksen aluksi

luodaan jälleen pohja, johon kaikki ohjelmamme komponentit sijoitetaan. Tällä kertaa määrittelemme ikkunalle otsikon "Laskin". Seuraavaksi lisäämme tekstikentän ja paketoimme sen samalla ohjelmaan mukaan. Tekstikenttä soveltuu staattisen tekstin esittämiseen ja parametreina annamme pohjan, johon tekstikenttä lisätään ja toisena parametrina itse lisättävän tekstin. Tämän jälkeen sijoitamme jäsenmuuttujaan `luku1` syöttökentän `Entry(pohja)` määritteen avulla. Tämä luo siis laatikon, johon käyttäjä voi syöttää tietoa. Tämän jälkeen paketoimme sen mukaan ohjelmaamme. Toistamme samat vaiheet vielä kahteen kertaan, jotta saamme kaikki tarvittavat komponentit mukaan. Viimeiseksi lisäämme mukaan nappulan (`Button`), jolla itse yhteenlasku tapahtuu. Nappulalle annamme parametrina `pohjan`, jotta se menee oikeaan paikkaan; tekstin, joka tulee nappulaan; ja kolmantena `command`-parametrina viitteen luokan omaan `summaa` jäsenfunktioon. Näin laskin kutsuu tätä funktiota aina kun nappulaa painetaan.

`summaa` jäsenfunktiossa kysymme luokkamme jäsenmuuttujilta (jotka ovat tyyppiä `Entry`) niiden sisältöä metodilla `get`. Tällä tavoin saatu sisältö on tekstiä, joten se täytyy muuntaa `float`illa liukulukumuotoon. Tilapäissäiliöön `summa` tallennetaan lukujen yhteenlaskun tulos. Tämän jälkeen `vastaus`-tekstikentän sisältö tuhotaan alusta loppuun ja sen tilalle lisätään merkkijonoksi muutettu yhteenlaskun summa.

Itse ohjelma käynnistyy luomalla uusi olio `LASKIN`-luokasta tiedoston viimeisellä rivillä.

Huomioita Tkinter-moduulista

Koska Tkinter-moduuli on hyvin laaja ja toimintatavoiltaan jonkin verran tavallisesta Python-ohjelmakoodista poikkeava, ei tässä oppaassa käsitellä aihetta 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

Komentorivikäyttöliittymä ja komentoriviparametrit

Komentorivikäyttöliittymä tarkoittaa sitä, että ohjelma käynnistetään käyttöjärjestelmän komentorivilta ohjelman nimellä. Alla esimerkki tästä Windows'n komentoikkunasta:

```
C:\Python372>komentorivi.py
Tämä ohjelma käynnistetään komentoriviltä.
```

Jos saman ohjelman ajaa IDLEn editori-ikkunasta, aukeaa interaktiivinen komentorivitulkki-ikkuna, johon tulee alla olevat tulosteet. Seuraava vaihe ei onnistu tässä ympäristössä, joten kannattaa siirtyä Windows'n komentorivikäyttöliittymään.

```
>>>
================= RESTART: C:\Python381\komentorivi.py ==================
Tämä ohjelma käynnistetään komentoriviltä.
>>>
```

Komentoriviparametrit noudattavat samaa ideaa kuin aliohjelman parametrit. Aliohjelmalle parametrit annetaan nimen perässä olevissa suluissa, esim. `print("eka", 2)`, jolloin parametreina ovat merkkijono `eka` ja luku `2`. Jos ajamme edellä olevan ohjelman IDLEssä, emme saa välitettyä sille tietoa, koska ohjelma ei kysy niitä erikseen tai lue niitä mistään. Komentirivillä voimme antaa parametrit ohjelman nimen jälkeen seuraavan esimerkin mukaisesti:

```
C:\Python372>komentorivi.py eka 2
Tämä ohjelma käynnistetään komentoriviltä.
C:\Python372\komentorivi.py
eka
2
```

Nyt ohjelma tulostaa saman kommentin kuin aiemminkin, mutta sen jälkeen se tulostaa kaikki komentoriviparametrit jokaisen omalle rivilleen. Tämä komentoriviohjelma on nähtävissä alla olevassa esimerkissä 13.3. Ohjelma sisältää perustulosteen lisäksi `sys`-kirjaston käyttöönoton, josta löytyy jäsenmuuttuja `argv` eli argumenttivektori, jossa on kaikki komentorivillä olleet sanat listassa merkkijonoina. Käyttöjärjestelmä siis ottaa komentoriville kirjoitetun merkkijonon talteen ja pilkkoo sen välilyönneistä erillisiksi merkkijonoiksi listaan, mikä kuulostaa tyypilliseltä `split()`:ltä, ja kaikki komentoriviparametrit voi käydä läpi `for`-lauseella. Tässä on syytä muistaa, että listalla on kaikki merkkijonot mukaan lukien ensimmäisenä oleva ohjelman nimi ja kaikki ovat merkkijonoja eli ne pitää muuttaa sopivaksi tietotyypiksi, virheentarkistusten kanssa, ennen käyttöä. Mutta tästä eteenpäin ohjelmointi on tuttua asiaa, ja esimerkissä uutta on vain `sys`-kirjaston `argv`-vektorin käyttö sekä komentoriviparametrien idea.

Esimerkki 13.3. Komentorivillä olevat parametrit tulostava ohjelma

```
import sys
print("Tämä ohjelma käynnistetään komentoriviltä.")
for parametri in sys.argv:
    print(parametri)
```

Jos edellä oleva ohjelma ei toimi komentorivillä, kannattaa tarkistaa onko python.exe-ohjelman sisältävä hakemisto lisätty käyttöjärjestelmän PATH-muuttujaan. Pythonin asennuksesta puhuttiin luvussa 8, joten sieltä löytyy tarkempia ohjeita tähän liittyen.

Osaamistavoitteet

Ymmärrät graafisen käyttöliittymän perusteet ja tiedät miten edetä jatkossa niiden parissa, jos tarvetta ilmenee. Samoin komentoriviohjelma ja komentoriviparametrit ovat tuttuja niin käsitteinä kuin toiminta-ajatuksena ja pystyt perehtymään niihin tarkemmin tarpeen mukaan.

Loppusanat

Olemme tämän oppaan avulla tutustuneet 13 luvussa ohjelmoinnin perusteisiin Python-kielellä sekä joihinkin ohjelmistokehityksen perusajatuksiin. Tähän pisteeseen päästyämme voimme jo alkaa puhumaan ohjelmistotuotannosta sekä laajemmasta 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 ohjelmointia. Tässä oppaassa käsiteltyjen perusasioiden ymmärtäminen on merkki siitä, että pystyt halutessasi tekemään 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 käyttäjäkunta kasvaa koko ajan.

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/. Eikä koskaan kannata unohtaa Pythonin omia – laajoja – dokumentaatiota, jotka löytyvät osoitteesta http://docs.python.org/py3k.

Oppaassa käsiteltiin kuutta eri muuttujan roolia. Lisää luettavaa aiheesta löytyy Jorma Sajaniemen sivuilta osoitteesta http://saja.kapsi.fi/var_roles/.

Lähdeluettelo

Python – ohjelmointiopas, versio 1.2, Kasurinen, Jussi, 2008.

Dive Into Python 3, Pilgrim, Mark, 2010.

Python-materiaalia suomen kielellä.
http://wiki.python.org/moin/FinnishLanguage

Python v3.x documentation, Python Software Foundation, 2010.
http://docs.python.org/py3k

The Roles of Variables, Sajaniemi, Jorma, 2008.
Lähde: http://saja.kapsi.fi/var_roles/

The Python Wiki
http://wiki.python.org/moin/

Wikipedia
Algoritmit, liukuluvut, pyöristys, pseudokoodi yms.

Liite 1: Lyhyt ohje referenssikirjastoon

Tässä liitteessä tutustumme lyhyesti Python-dokumenttien keskeiseen osioon, Python Software Foundationin referenssikirjastoon. Kyseinen kirjasto sisältää kaiken tarpeellisen tiedon Python-ympäristöstä, sen toiminnoista, funktioista, moduuleista sekä operaattoreista.

Alla olevaan taulukkoon on listattu ne luvut, joista luultavimmin löytyy lisätietoa tässä oppaassa läpikäydyistä asioista. Mikäli haluat tutustua luvun sisältöön, mene Internetissä osoitteeseen http://docs.python.org/py3k/library/index.html (Python versio 3) ja etsi vastaava luku. Verkossa olevat dokumentit ovat englanninkielisiä.

Luku	**Sisältö**
2	Tietoa Python-ympäristön sisäänrakennetuista funktioista ja käskyistä kuten input(), print(), import sekä len()
5.2	Tietoa Boolen-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 on 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 ja 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. 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 on 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 ehtorakenteena 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 kuuluu yleisesti 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-rakenteeseen kuuluu oma koodilohko.
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 ohjelmoija voi 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 moduuli tarkoittaa 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.
paluuarvo	Paluuarvo 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 käsitellä 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 tähän mennessä paras tai soveltuvin 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 alkiota, 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 haluttu toistoehto saavutetaan. Pythonissa toistorakenteita ovat while ja for -rakenteet.
tulkki	Tulkki on ohjelma, jolla voidaan suorittaa lähdekooditiedostoja. Tulkki lukee lähdekooditiedostoja ja rivejä 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.
Exception	Yleinen poikkeus kattaen useimmat poikkeukset pl. SysExit ja KeyBoardInterrupt	Tässä oppaassa poikkeustenkäsittely riittää tehdä tähän poikkeukseen liittyen.
EOFError	input() ei saanut luettavaa arvoa.	Lopetit ohjelman suorituksen CTRL-D-yhdistelmällä komentokehotteessa.
FileNotFoundError	Avattavaa tiedostoa ei löydy.	Yritit avata tiedoston luettavaksi, mutta tiedostoa ei löytynyt.
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.	Käytit nimeä, jota ei ole määritelty, esim. kirjoitit funktion nimen väärin, tai käytit funktiota ennen sen määrittelyä.
OSError	Käyttöjärjestelmävirhe liittyen esim. tiedostoon tai hakemistoon.	Yritit esim. avata tiedostoa, jota ei ole, tai levy on täysi eikä kirjoittaminen onnistu.
RuntimeError	Tapahtui yleinen virhe.	Python-tulkki ei osaa kertoa, mikä virhe tapahtui, mutta jotain meni vikaan.

SyntaxError	Koodin synktaksissa on virhe.	Olet luultavasti sisentänyt jonkin rivin väärin tai koodistasi puuttuu pilkkuja tai heittomerkkejä.
SystemError	Tulkin sisäisissä tiedostoissa tapahtui virhe.	Python-tulkin asennus on mennyt jostain syystä rikki. Aja asennuspaketista korjausasennus.
SystemExit	Ohjelma lopetti toimintansa.	Lopetit ohjelman funktiolla sys.exit().
TypeError	Tietotyypeissä on yhteensopivuusongelma	Koetit muuttaa merkkijonon numeroarvoksi, 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 muitakin virhetiloja, jotka eivät ole tyypillisiä tämän oppaan aihealueilla. 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 LUT yliopiston Ohjelmoinnin perusteet -kurssin tueksi. Tyylioppaan tarkoitus on orientoida opiskelija käyttämään oikeita rakenteita oikeissa tarkoituksissa ja oikealla tyylillä kirjoitettuna.

Pythonin luojan 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 ymmärrettävyyttä sekä pitämään lähdekoodin kieliasu yhtenäisenä. Kannattaa kuitenkin pitää mielessä, että jossain tilanteissa tietyt tyylisäännöt tai tyylioppaat eivät vain yksinkertaisesti päde. Tällöin tulee pyrkiä pitämään linja yhtenäisenä ja huolehtia Python-koodin tärkeimmästä ominaisuudesta eli sen ymmärrettävyydestä ja selkeydestä.

Sisennyksestä

Käytä sisennyksissä koodin tasolta toiselle aina neljää (4) välilyöntiä per taso. Ainoan poikkeuksen tähän tekee vanhojen lähdekoodien ylläpito ja korjaus, kun ne perustuvat aiemmin voimassa olleeseen 8 välilyönnin standardiin.

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 luokkamäärittelyt toisistaan tyhjällä 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.

- Välittömästti 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),
  Boolen-arvot (and, or, not).
  ```
- Erottele kommentit #-merkistä välilyönnillä
  ```
  # Tämä on kommentti.
  ```

- 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)
```

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 kommentit aina englanniksi mikäli on pienikin mahdollisuus, 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 koodiosion kanssa. Monirivisessä kommentissa käytä '#'-merkin jälkeen ainakin yhtä välilyöntiä. Kappaleenvaihto monirivisessä kommentissa merkitään tyhjällä kommenttirivillä, jolla on ainoastaan "#"-merkki.

Koodinsisäinen kommentointi

Koodinsisäinen kommentointi tarkoittaa koodirivin jatkoksi kirjoitettua kommenttia. 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 tietyn kaavan mukaisesti.

Nimeämislogiikasta

Käytä muuttujien, funktioiden ja rakenteiden nimeämiseen jotain seuraavista tavoista:

```
- muuttujannimi (pelkkiä pieniä kirjaimia)
- muuttujan_nimi_alaviivalla_erotettuna
- IsotKirjaimet (muuttujan nimen sanat alkavat isolla kirjaimella)
```

Huom: 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 (esim. ”except_” tai ”_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 luokka-rakenteiden käyttöön.

Huomioita

Edellä oli 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

Alla on esimerkkejä tämän oppaan mukaan mukaisella tyylillä kirjoitetuista ohjelmointirakenteista. Huomaa, että kukin esimerkki keskittyy otsikon mukaiseen asiaan ja laajaa ohjelmaa kirjoitettaessa useiden esimerkkien asiat tulee toteuttaa samassa ohjelmassa.

Tiedon tulostaminen näytölle

```
# Normaali tekstin tulostus
print("Tekstiä.")
print('Lisää "tekstiä"')
print("Rivi 1\nRivi 2")
print("Rivi 1", "Rivi 2", sep="\n")
print("Rivin alku", end=' ')
print("ja samalle riville tuleva rivin loppu.")
# Määritetään muuttujat ja tulostetaan niiden sisältö ruudulle
nopeus = 13
pii = 3.14159
kaupunki = "Lappeenranta"
print(str(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

```
# Esimerkki 1. Pelkkää tulostusta sisältävä aliohjelma.
def valikko():
    print("Valikko")
    print("1) Avaa tiedosto")
    print("2) Tallenna tiedosto")
    print("3) Järjestä tiedoston sisältö")
    print("0) Lopeta")
    return None
# Kutsutaan aliohjelmaa sen nimellä
valikko()

# Esimerkki 2. Aliohjelma parametrilla sekä sitä kutsuva pääohjelma.
# Aliohjelma parametrilla ja paluuarvolla eli funktio. Itse funktio
# palauttaa parametrinsa itseisarvon
def itseisarvo(luku):
    if (luku < 0):
        luku = luku * -1
    return luku

# Luodaan kokonaislukulista ja tulostetaan listan alkiot ja niiden
# itseisarvot itseisarvo()-funktion avulla pääohjelmassa.
def paaohjelma():
    luvut = [1, -3, 0, -192299, 3097]
    for alkio in luvut:
        print("Luvun", alkio, "itseisarvo on", itseisarvo(alkio))
    return None

paaohjelma()
```

Tiedostoon kirjoittaminen

```
# Otetaan sys mukaan suorituksen keskeyttämiseksi 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 OSError:
        print("Tiedostoon kirjoittaminen ei onnistunut.")
        tiedosto.close()
        sys.exit(1)
except OSError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(1)
tiedosto.close()
print("Tiedostoon kirjoittaminen suoritettu onnistuneesti.")
```

Tiedostosta lukeminen

```
# Avataan tiedosto lukemista varten ja luetaan sen sisältämät rivit
import sys
try:
    tiedosto = open("luvut.txt", "r", encoding="utf-8")
    while (True):
        rivi = tiedosto.readline()
        if (len(rivi) == 0):
            break
        rivi = rivi[:-1]
        print(rivi)
except OSError:
    print("Tiedostoa ei voitu avata.")
    sys.exit(1)
tiedosto.close()
print("Tiedoston lukeminen suoritettu onnistuneesti.")
```

Poikkeukset

```
# Kysytään kaksi 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)
```

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 alkioittain erilaisilla for-silmukoilla
for alkio in lista:
    print(alkio)
for i in range(0, len(lista)):
    print(lista[i])
```

Luokka ja lista -tietorakenteiden yhteistoiminta

```
# Luodaan AUTO-luokka, kaksi autoa ja autokauppa-lista
class AUTO:
    merkki = ""
    vuosimalli = ""

auto1 = AUTO()
auto2 = AUTO()
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) + ".")
```

Ohjelmatiedoston rakenne

```
# Alkukommentit aloittavat tiedoston.
# Kirjastot sisällytetään tarvittaessa globaaleina rakenteina.
# Kiintoarvot määritellään globaaleina tunnuksina.
# Luokat määritellään globaaleina rakenteina.
# Aliohjelmat määritellään globaaleina tunnuksina.
# Huom. Globaalit muuttujat ovat kiellettyjä!

def valikko():  # Aliohjelmat määritellään ennen pääohjelmaa.
    print("Valikko")
    print("1) Kysy nimi")
    print("2) Tulosta nimi")
    print("0) Lopeta")
    return (int(input("Valintasi: ")))

def paaohjelma():  # Pääohjelma pidetään yksinkertaisena.
    while (True):
        valinta = valikko()
        if (valinta == 1):
            nimi = input("Anna nimi: ")
        elif (valinta == 2):
            print(nimi)
        elif (valinta == 0):
            break
        else:
            print("Tuntematon valinta, yritä uudestaan.")
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()  # Päätasolla on vain pääohjelmakutsu.
# eof
```