PYTHON

-

OHJELMOINTIOPAS,

VERSIO 1.1

Lappeenrannan teknillinen yliopisto 2007
Jussi Kasurinen
ISBN 978-952-214-440-9 ISSN 1459-3092

10

PYTHON – OHJELMOINTIOPAS
versio 1.1.

Jussi Kasurinen

Käsikirjat 10
Manuals 10

PYTHON – OHJELMOINTIOPAS
versio 1.1.

Jussi Kasurinen

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan osasto
PL 20
53851 Lappeenranta

ISBN 978-952-214-440-9
ISSN 1459-3092

Lappeenranta 2007

Alkuperäinen käännös ja lokalisointi, täydentävä materiaali sekä taitto:

Jussi Kasurinen

Toteutuksen ohjaus sekä tarkastus:

Uolevi Nikula

Lappeenrannan teknillinen yliopisto, Tietojenkäsittelytekniikan laboratorio.

Lappeenranta 26.9.2007

Alkusanat

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

Viikoittaisten otsikkojen alapuolella lukee, mistä teoksesta teksti on alun perin käännetty suomeksi. Lisäksi lähdeteoksen vaihtuessa on otsikkorivillä merkki, josta eteenpäin oleva teksti on kyseessä olevasta lähteestä. Merkintä ’BoP’ tarkoittaa teosta Byte of Python, ‘PSF’ Foundationin dokumentaatioita ja ‘LwP’ How to think like a Computer Scientistiä. Lisäksi käytössä on merkintä “LTY” joka tarkoittaa, että kyseinen teksti on tätä opasta varten kirjoitettua alkuperäistekstiä.

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

Aiheiden jako kokonaisuuksiin noudattaa ensisijaisesti Lappeenrannan teknillisen yliopiston syksyn 2007 kurssin ”Ohjelmoinnin Perusteet” viikoittaista jakoa. Lisäksi joissain luvuissa tekstin sekaan on lisätty aiheita, jotka eivät suoranaisesti liity viikoittaiseen aiheeseen, mutta ovat hyödyllistä lisätietoa. Lisäksi huomioi, että oppaan lopussa olevissa liitteissä on paljon hyödyllistä lisätietoa esimerkiksi tulkin virheilmoituksista, sanastosta sekä ohjelmointityyleistä.

Kuten kurssillakin, myös teoriaosioissa sekä materiaalissa oletetaan, että käyttäjä tekee tehtäviä IDLE-kehitysympäristön avulla WindowsXP-työasemalla, ja käyttää Python-tulkin versiota 2.5.1. Lisäksi Python-tulkkiin tulisi olla asennettuna seuraavat laajennusmoduulit: NumPy, Imaging Library sekä py2exe. Mikäli tarvitset lisäohjeita tarvittavien ohjelmien asennuksesta, löytyy liitteestä 1 vaiheittaiset ohjeet ohjelmointiympäristön asennukseen.

Tämä opas on täydennetty ja paranneltu versio aiemmin ilmestyneestä ensimmäisestä Python-ohjelmointioppaasta ja on tarkoitettu korvaamaan kyseinen teos.

SISÄLLYSLUETTELO

Luku 1: Ensiaskeleet Pythoniin

Alkuperäislähde Byte of Python luku 3

Esittely

BoP

Ensimmäisessä osiossa näet, kuinka perinteinen ‘Hello World’ -ohjelma voidaan toteuttaa Pythonilla. Samalla opit kuinka voit kirjoittaa, ajaa ja tallentaa kirjoittamiasi koodeja Python –ympäristössä.

Pythonia käyttäessäsi voit ajaa ohjelmia kahdella tavalla; käyttäen komentorivitulkkia tai ajamalla lähdekooditiedostoja. Seuraavaksi näet kuinka nämä menetelmät toimivat.

Komentorivitulkin käyttäminen

Avaa komentorivikehoite valitsemalla Käynnistä-valikosta vaihtoehto ”Suorita…” ja kirjoita aukeaavaan ikkunaan **command** ja paina **Enter**. Nyt eteesi pitäisi aueta mustapohjainen teksi-ikkuna; olet komentorivikehoitteessa. Käynnistä tulkki kirjoittamalla käsky **python** komentoriville ja paina **Enter**. Tämän jälkeen kirjoita `print 'Hello World'` ja paina **Enter**. Sinun tulisi nähdä tulostunut teksti `Hello World`.

Windows-käyttäjät voivat ajaa tulkin komentoriviltä mikäli `PATH`-muuttuja on määritelty oikein. Normaalisti tämä tapahtuu automaattisesti asennuksen yhteydessä joten luultavimmin sinun ei tästä tarvitse itse huolehtia. Mikäli kuitenkin tulee ongelmia, vaihtoehto Windows-käyttäjille onkin ajaa Python-koodi IDLE:llä. IDLE on lyhenne sanoista Integrated DeveLopment Environment, ja sen haku- ja asennusohjeet läpikäytiin edellisessä luvussa. Ohjelma löytyy käynnistysvalikosta polun Start -> All Programs -> Python 2.5 -> IDLE (Python GUI) kautta. IDLE-ympäristössä interaktiivinen shell-ikkuna toimii samalla tavoin kuin komentorivitulkki. IDLE on saatavilla myös Linux- ja MacOS-järjestelmille.

Huomaa, että jatkossa esimerkkien merkintä >>> tarkoittaa komentorivitulkille syötettyä Python-käskyä.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
```

```
>>> print "Hello World!"
Hello World!
>>>
```

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

Kuinka suljen komentorivitulkin

Jos käytit Pythonia Windowsin komentorivikehotteesta, sulkeutuu tulkki painamalla ensin **Ctrl-z** ja tämän jälkeen **Enter.** Jos käytät IDLEn shell-ikkunaa tai Linuxin komentoriviä, sulkeutuu tulkki painamalla **Ctrl-d.**

Lähdekooditiedoston käyttäminen

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

Avaa valitsemasi editori ja kirjoita alla olevan esimerkin mukainen koodi. Tämän jälkeen tallenna koodi tiedostoon helloworld.py.

Esimerkki 1.2. Lähdekooditiedoston käyttäminen

```
# -*- coding: cp1252 -*-
#Ensimmäinen ohjelma
print "Hello World!"
```

Tämän jälkeen aja ohjelma. Jos käytät IDLEä, onnistuu tämä editointi-ikkunan valikosta Run-> Run Module. Tämä voidaan toteuttaa myös pikavalintanäppäimellä F5. Muussa tapauksessa avaa komentorivikehote ja kirjoita käsky **python *`hellowor ld.py`***.

Tuloste

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

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

Jos taas koodisi aiheutti virheen, tarkasta että kirjoitit koodisi *täsmälleen* samoin kuin esimerkissä ja aja koodisi uudestaan. Erityisesti huomioi se, että Python näkee isot ja pienet kirjaimet eri merkkeinä. Tämä tarkoittaa sitä, että esimerkiksi 'Print' ei ole sama

asia kuin ‘print’. Varmista myös, että et epähuomiossa laittanut välilyöntejä tai muutakaan sisennystä rivien eteen, tästä puhumme lisää seuraavissa luvuissa.

Miten se toimii

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

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

Huomautus

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

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

Yhteenveto

Tässä vaiheessa sinun pitäisi osata kirjoittaa yksinkertainen Python-koodi, käyttää editoria sekä ajaa koodinpätkiä niin komentorivitulkilla kuin lähdekooditiedostostakin. Nyt kun olet Python-ohjelmoija, niin siirrytään eteenpäin ja jatketaan Python-ohjelmointiin tutustumista.

Luku 2: Muuttujat, tiedon vastaanottaminen, loogiset lausekkeet

Alkuperäislähde, Byte of Python luvut 4 ja 5 sekä BSF tutorial, luku 3

Perusteet

BoP

Pelkkä ‘Hello World’ ei ole kovinkaan kattava ohjelma. Luultavasti haluaisit tehdä ohjelmallasi myös jotain muuta, kuten ottaa vastaan syötteitä, muunnella niitä ja saada aikaan jonkinlaisia vastauksia. Kaikki tämä onnistuu Pythonissa käyttämällä apunamme vakioita ja muuttujia.

Vakiomuuttuja

Esimerkkejä vakioista voi olla vaikkapa numeroarvot `5`, `1.23`, `9.25e-3` tai merkkijono kuten `'Tämä on merkkijono'` tai `"Tämäkin on merkkijono!"`. Näitä sanotaan vakioiksi, koska ne todellakin ovat *vakioita* – niitä käytetään sanantarkasti eivätkä ne koskaan voi olla arvoltaan mitään muuta kuin juuri se arvo, joka niille on annettu.

Numerot

Pythonista löytyy neljä erilaista numerotyyppiä: kokonaisluvut, pitkät kokonaisluvut, liukuluvut (kansanomaisesti desimaaliluvut) sekä kompleksiluvut.

- Esimerkiksi `2` tai `-5` ovat kokonaislukuja, koska ne eivät sisällä desimaaliosaa.
- Pitkät kokonaisluvut ovat käytännössä ainoastaan erittäin suurikokoisiksi kasvamaan kykeneviä kokonaislukuja.
- Desimaalilukuja ovat esimerkiksi `3.23` and `52.3E-4`. Merkki `E` tarkoittaa kymmenpotenssia. Tässä tapauksessa, `52.3E-4` on siis $52.3 * 10^{-4}$.
- Kompleksilukuja ovat vaikkapa `(-5+4j)` ja `(2.3 - 4.6j)`

Merkkijonot

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

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

- **Käyttäen sitaattimerkkiä (')**

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

- **Käyttäen lainausmerkkiä (")**

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

- **Käyttäen kolmea sitaatti- tai lainausmerkkiä (''', """)**

  Voit määritellä useamman rivin pituisia merkkijonoja käyttämällä kolmea sitaattimerkkiä. Kolmen sitaattimerkin sisällä voit käyttää vapaasti myös yllä olevia sitaattimerkkejä. Esimerkiksi näin:

```
'''Tämä on monirivinen teksti, tässä ensimmäinen rivi.
Tämä on toinen rivi.
"Kuka olet?", hän kysyi.
Mies vastasi "Doe, John Doe."
'''
```

- **Ohjausmerkit**

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

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

Entä jos haluat tulostaa useammalle riville? Voit käyttää kolmen sitaattimerkin tapaa joka juuri esiteltiin, tai sitten vaihtoehtoisesti käyttää rivinvaihtomerkkiä (`\n`). Rivinvaihtomerkki tulee näkyviin tekstiin normaalisti kauttaviiva-n-yhdistelmänä, mutta tulkissa tulostuu rivinvaihtona. Esimerkiksi `"Tämä tulee ensimmäiselle riville. \n Tämä tulee toiselle riville."` Toinen vastaava hyödyllinen merkki on sisennysmerkki (\t), joka vastaa tabulaattorimerkkiä ja jolla voimme tasata kappaleiden reunoja. Ohjausmerkeistä on hyvä tietää lisäksi se, että yksittäinen kenoviiva rivin päässä tarkoittaa sitä, että merkkijono jatkuu seuraavalla rivillä. Tämä aiheuttaa sen, että tulkki ei lisää rivin päähän rivinvaihtoa vaan jatkaa tulostusta samalle riville. Esimerkiksi,

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

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

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

- **“Raa’at” merkkijonot**

  Jos haluat merkitä, että jokin merkkirivi ei sisällä erikois- tai ohjausmerkkejä vaan on tarkoitettu tulostettavaksi merkilleen kuten on kirjoitettu, voidaan se esittää raakarivi-merkillä. Merkkijonon eteen laitetaan etuliite ‘r’ tai ‘R’, jolloin tulkki ohittaa ohjausmerkit ja tulostaa ne sellaisenaan ruudulle. Esimerkki raa’asta merkkijonosta olisi vaikkapa

  ```
  r"Rivinvaihto merkitään merkkiyhdistelmällä \n".
  ```

- **Unicode-merkkijonot**

  Unicode on kansainvälinen standardi, jonka avulla voidaan ilmaista paikallisia erikoismerkkejä ja ei-länsimaisia aakkostoja. Jos haluat kirjoittaa Hindiä tai arabialaisilla kirjaimilla, joudut valitsemaan editorin joka tukee unicode-moodia. Python osaa operoida kyseisillä merkeillä, kun niitä sisältäviin merkkiriveihin lisätään etuliitte ’u’ tai ’U’.

  Unicode-operaattoria tarvitaan lähinnä silloin, kun työskennellään sellaisten tiedostojen kanssa, jotka sisältävät laajennetun ASCII-taulukon ulkopuolisia merkkejä tai ovat kirjoitettu ei-länsimaisilla aakkosilla.

- **Merkkijonot ovat vakioita**

  Tämä tarkoittaa sitä, että kun olet kerran luonut merkkijonon, et voi muuttaa suoraan sen sisältöä. Vaikka tämä vaikuttaa ongelmalliselta, ei se itse asiassa ole sitä. Tämä ei aiheuta juurikaan rajoituksia, ja myöhemmässä vaiheessa näytämme useampaan otteeseen miksi asia on niin.

- **Merkkijonojen yhdistäminen**

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

Muuttujat

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

Muuttujien nimeäminen

Muuttujat ovat esimerkki tunnisteista. Tunniste tarkoittaa nimeä, jolla yksilöidään jokin tietty asia. 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 (iso tai pieni), alaviivoja tai numeroita (0-9).
- Skandinaaviset merkit (å,ä,ö,Å,Ä,Ö) eivät kelpaa muuttujien nimiin.
- Nimet ovat aakkoskoosta riippuvaisia (eng. case sensitive), eli isot ja pienet kirjaimet ovat tulkille eri merkkejä. Siksi nimet ”omanimi” ja ”omaNimi” **eivät** tarkoita samaa muuttujan nimeä.
- Kelvollisia nimiä ovat muun muassa `i`, `__mun_nimi`, `nimi_23` ja `a1b2_c3`.
- Epäkelpoja nimiä taas ovat esimerkiksi `2asiaa`, `taa on muuttuja, jäljellä` ja `-mun-nimi`.

Muuttujien tietotyypit

Muuttujille voidaan antaa mitä tahansa Pythonin tuntemia tietyyppejä, kuten merkkijonoja tai numeroita. Kannattaa kuitenkin huomata, että jotkin operaatiot eivät ole mahdollisia keskenään, kuten esimerkiksi kokonaisluvusta ei voi vähentää merkkijonoa. Palaamme näihin rajoituksiin myöhemmissä luvuissa.

Kuinka kirjoitan Pythonilla ohjelman

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

1. Avaa mieleisesi koodieditori.
2. Kirjoita alla olevan esimerkin mukainen koodi, muista kirjainkoot.
3. Tallenna tiedosto nimellä muuttuja.py. Vaikka editorisi ei tätä automaattisesti tekisikään, on hyvä muistaa Python-koodin pääte **.py**, jotta käyttöjärjestelmä –ja jotkin editorit - tunnistavat tiedostosi Python-koodiksi.
4. Aja koodisi komentoriviltä komennolla **python *muuttuja.py*** tai käytä IDLEn komentoa ’Run Module’ (F5).

Esimerkki 2.1 Muuttujien käyttäminen ja vakioarvot

```
# -*- coding: cp1252 -*-
# Tiedosto: muuttuja.py

luku = 5
print luku
luku = luku + 1
print luku

teksti = '''Tämä on monirivinen teksti.
Tämä tulee toiselle riville.'''
print teksti
```

Tuloste

```
>>>
5
6
Tämä on monirivinen teksti.
Tämä tulee toiselle riville.
>>>
```

Kuinka se toimii

Ohjelmasi toimii seuraavasti: Ensin me määrittelemme muuttujalle `luku` vakioarvon `5` käyttäen sijoitusoperaattoria `(=)`. Tätä riviä sanotaan käskyksi, koska rivillä on määräys siitä, että jotain pitäisi tehdä. Tässä tapauksessa siis sijoittaa arvo `5` muuttujaan `luku`.

Seuraavalla rivillä olevalla toisella käskyllä me tulostamme muuttujan `luku` arvon ruudulle käskyllä `print`. Sijoitusoperaatio toimii loogisesti aina siten, että se arvo, johon sijoitetaan on operaattorin vasemmalla puolella ja se, mitä sijoitetaan oikealla.

Tämän jälkeen me kasvatamme luvun arvoa yhdellä ja tallennamme sen takaisin muuttujaan `luku`. Tulostuskäsky `print` antaakin – kuten olettaa saattoi - tällä kertaa arvon 6.

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

Huomautus muuttujista

LTY

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 huoletta tallentaa tilalle merkkijono. Joissain tapauksissa Pythonin syntaksi tosin vaatii sen, että käytettävä muuttuja on jo aiemmin määritelty joksikin soveltuvaksi arvoksi. Useimmiten tähän törmätään kun käytetään muuttujia toistorakenteiden lippumuuttujina.

BoP

Loogiset ja fyysiset rivit

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

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

Jos kuitenkin haluat käyttää kirjoitusasua, jossa sijoitat enemmän kuin yhden loogisen rivin fyysiselle riville, voit toteuttaa sen puolipisteiden (;) avulla. Puolipiste toimii erotinmerkkinä kahden loogisen lauseen rajalla mahdollistaen peräkkäisten lauseiden kirjoittamisen. Esimerkiksi

```
i = 5
print i
```

on täsmälleen sama kuin

```
i = 5;
print i;
```

joka taas on edelleen sama kuin

```
i = 5; print i;
```

tai

```
i = 5; print i
```

Kuitenkin, **on erittäin suositeltavaa, että kirjoitat koodin sellaiseen muotoon, jossa yksi fyysinen rivi koodia tarkoittaa yhtä loogista käskyä.** Käytä useampaa fyysistä riviä yhtä loogista riviä kohti ainoastaan mikäli rivistä on tulossa erittäin pitkä. Taka-ajatuksena tässä on se, että koodi pyritään pitämään mahdollisimman helppolukuisena ja yksinkertaisena tulkita tai tarvittaessa korjata. Puolipisteen erottamia loogisia rivejä tulee välttää niin kauan kuin se on mahdollista. Itse asiassa puolipiste-syntaksille ei ole mitään ehdotonta tarvetta, mutta se on olemassa, jotta sitä voitaisi tarvittaessa käyttää.

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

```
s = 'Tämä on merkkijono. \
Merkkijono jatkuu täällä.'
print s
```

Tämä tulostaa seuraavan vastauksen:

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

Samoin,

```
print \
i
```

on sama kuin

```
print i
```

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

Sisennys

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

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

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

```
i = 5
  print 'Arvo on', i # Virhe! Huomaa ylimääräiset välilyönnit alussa
print 'Arvo siis on', i
```

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

```
  File "whitespace.py", line 4
    print 'Arvo on’, i # Virhe! Huomaa ylimääräiset välilyönnit alussa
SyntaxError: invalid syntax
```

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

Molemmissa tapauksissa syy on toisen rivin alussa olevassa ylimääräisessä välilyönnissä. Syntax error tarkoittaa sitä, että Python-koodissa on jotain, mikä on täysin kieliopin vastaista, eikä tulkki pysty edes sanomaan mitä siitä pitäisi korjata. Käytännössä sinun tulee muistaa tästä osiosta lähinnä se, että **et voi aloittaa satunnaisesta paikasta koodiriviä, vaan sinun on noudatettava sisennyksien ja osioiden kanssa loogista rakennetta**. Rakenteita, joissa luodaan uusia osioita, käsitellään muutaman viikon kuluttua.

Ohjeita sisennysten hallintaan

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

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

Kommenttirivit

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

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

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

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

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

Tulostaisi se seuraavanlaisen tekstin

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

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

Yhteenveto

Nyt olemme läpikäyneet tärkeimmät yksityiskohdat Pythonin kanssa operoinnista ja voimme siirtyä eteenpäin mielenkiintoisempiin aiheisiin. Seuraavaksi alamme läpikäymään yleisimpiä operaattoreita, jotka liittyvät muuttujilla operointiin. Aikaisemmin olet jo tutustunut niistä muutamaan, (+) ja (=) operaattoreihin.

Operaattorit ja lausekkeet

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

Operaattorit

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

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

Taulukko 2.1 Laskentaoperaattorit

Operaattori	Nimi	Selite	Esimerkki
=	Sijoitus	Sijoittaa annetun arvon kohdemuuttujalle	`luku = 5` sijoittaa muuttujalle luku arvon 5. Operaattori toimii ainoastaan mikäli kohteena on muuttuja.
+	Plus	Laskee yhteen kaksi operandia	`3 + 5` antaa arvon `8`. `'a' + 'b'` antaa arvon `'ab'`.
-	Miinus	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` (kokonaislukujen jako palauttaa kokonaisluvun). `4.0/3` tai `4/3.0` antaa arvon `1.3333333333333333`
//	Tasajako	Palauttaa tiedon kuinka monesti y menee x:ään	`4 // 3.0` antaa arvon `1.0`
%	Jakojäännös	Palauttaa x:n jakojäännöksen y:stä.	`8%3 antaa 2`. `-25.5%2.25` antaa `1.5`.

Taulukko 2.2 Loogiset- eli vertailuoperaattorit

Operaattori	Nimi	Selite	Esimerkki
<	Pienempi kuin	Palauttaa tiedon siitä onko x vähemmän kuin y. Vertailu palauttaa arvon 0 (`False`) tai 1 (`True`).	`5 < 3` palauttaa arvon `0` (eli. `False`) ja `3 < 5` palauttaa arvon `1` (eli. `True`). Vertailuja voidaan myös ketjuttaa: 3 < 5 < 7 palauttaa arvon `True`.
>	Suurempi kuin	Palauttaa tiedon onko x enemmän kuin y.	`5 > 3` palauttaa arvon `True`. Jos molemmat operandit ovat numeroita, ne muutetaan ne automaattisesti vertailukelpoisiksi eli samantyyppisiksi. Merkkijonoista verrataan ensimmäisiä (vasemmanpuolimmaisia) kirjaimia ASCII-taulukon mukaisessa järjestyksessä.
<=	Vähemmän, tai yhtä suuri	Palauttaa tiedon onko x pienempi tai yhtä suuri kuin y.	`x = 3; y = 6; x <= y` palauttaa arvon `True`.
>=	Suurempi, tai yhtä suuri	Palauttaa tiedon onko x suurempi tai yhtä suuri kuin y.	x = 4; y = 3; x >= 3 palauttaa arvon `True`.
==	Yhtä suuri kuin	Testaa onko operandit yhtä suuria.	`x = 2; y = 2; x == y` palauttaa arvon `True`. `x = 'str'; y = 'stR'; x == y` palauttaa arvon `False`. `x = 'str'; y = 'str'; x == y` palauttaa arvon `True`.
!=	Erisuuri kuin	Testaa onko operandit erisuuria.	`x = 2; y = 3; x != y` palauttaa arvon `True`.

Taulukko 2.3 Boolean-operaattorit

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

Taulukko 2.4 Bittioperaattorit

Operaattori	Nimi	Selite	Esimerkki
<<	Siirto vasempaan	Siirtää luvun bittejä annetun numeroarvon verran vasemmalle. (Jokainen luku on ilmaistu muistissa bitteinä.)	`2 << 2` palauttaa `8`. `2` on ilmaistu käyttäen arvoa `10`. Vasemmalle siirtyminen 2 bitin verran antaa rivin `1000`, joka taas on arvona 8.
>>	Siirto oikeaan	Siirtää luvun bittejä oikealle numeroarvon verran.	`11 >> 1` palauttaa `5`. `11` on bitteinä `1011`, josta siirryttäessä 1 bitti oikeaan tulee `101` joka taas lukuina on `5`.
&	Bittijonon AND	Bittijonon AND yksittäisille biteille.	`5 & 3 palauttaa arvon 1`.
|	Bittijonon OR	Bittijonon OR yksittäisille biteille.	`5 | 3` palauttaa arvon `7`
^	Bittijonon XOR	XOR(valikoiva OR) yksittäisille biteille.	`5 ^ 3` antaa arvon `6`
~	Bittijonon kääntö	Bittijonon käännös arvolle x on –(x+1)	`5` palauttaa arvon `-6`.

Operaattorien suoritusjärjestys

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

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

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

.

Taulukko 2.2 Operaattorien suoritusjärjestys

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

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

Laskentajärjestyksestä

Oletusarvoisesti suoritusjärjestys toteutuu taulukon mukaisesti. Kuten edellä kuitenkin mainittiin, voit muuttaa järjestystä käyttämällä sulkuja. Suluilla työskennellään 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`.

Liitännäisyys

Operaattorit arvioidaan yleisellä tasolla vasemmalta oikealle, mikäli niiden suoritusjärjestys on samaa tasoa. Esimerkiksi `2 + 3 + 4` on käytännössä sama kuin `(2 + 3) + 4`. Kannattaa kuitenkin muistaa, että jotkin operaattorit, kuten sijoitusoperaattori, toimivat oikealta vasemmalle. Tämä tarkoittaa siis sitä, että lauseke `a = b = c` tulkitaan olevan `a = (b = c)`.

Lausekkeet

Lausekkeiden käyttö

Esimerkki 2.2. Lausekkeiden käyttö

```
# -*- coding: cp1252 -*-
# Tiedosto: expression.py

pituus = 5
leveys = 2

pinta = pituus * leveys
print 'Pinta-ala on', pinta
print 'Kehä on', 2 * (pituus + leveys)
```

Tuloste

```
>>>
Pinta-ala on 10
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-käskyn sisällä.

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

Tiedon lukeminen käyttäjältä

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

Syötteen ottaminen

Syötteen käyttäminen

Esimerkki 2.2. Arvojen vastaanottaminen käyttäjältä

```
# -*- coding: cp1252 -*-
# Tiedosto: expression.py

pituus = input('Anna kappaleen pituus: ')
leveys = 2

pinta = pituus * leveys
print 'Pinta-ala on', pinta
print 'Kehä on', 2 * (pituus + leveys)
```

Tuloste

```
>>>
Anna kappaleen pituus: 6
Pinta-ala on 12
Kehä on 16
>>>
```

Kuinka se toimii

Ensimmäisellä rivillä sijoitamme muuttujalle pituus arvoksi input()-funktion tuloksen. Input-funktio saa syötteenä merkkijonon, kuten nyt ”Anna kappaleen pituus: ”, ja tulostaa tämän merkkijonon ruudulle toimintaohjeeksi käyttäjälle. Käyttäjä syöttää mieleisensä luvun shell-ikkunaan tai komentorivitulkille, jonka jälkeen annettu luku tallentuu muuttujaan pituus. Seuraavaksi tulostamme ja laskemme pinta-alan ja kehän pituuden edellisen esimerkin tavoin.

Huomautus

Input()-funktion kanssa tulee olla tarkkana, koska sen käytössä on joitakin rajoituksia. Ensinnäkin se osaa luontevasti vastaanottaa ainoastaan lukuarvoja, kuten kokonaislukuja

tai liukulukuja. Merkkijonoja input osaa käsitellä ainoastaan jos ne syötetään sille sitaattien sisällä. Input()-funktion etu on kuitenkin se, että se osaa tallettaa annetun arvon oikeanlaisena tietotyyppinä eikä erillisiä tyyppimuunnoksia tarvita.

Merkkijonojen kanssa työskennellessä käytetään vastaavaa, mutta paljon avoimempaa funktiota raw_input(). Kuten raaoissa merkkijonoissa, raw_input() tallentaa saamansa arvon muuttujaan merkkijonona eikä se käsittele erikoismerkkejä tai ohjausmerkkejä vaan tallentaa ne merkkimuotoisina annettuun jonoon.

Esimerkki 2.3. Raw_input()-funktion käyttäminen

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: input.py

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

Tuloste

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

Kuinka se toimii

Tässä versiossa tärkeää on huomata se ero, että merkkijono ”Kumiankka” pystyttiin antamaan ilman sitaatteja, mikä taas ei toimisi input()-funktiolla. Toisaalta taas raw_input()-funktiolle annetut luvut tallentuvat merkkijonoina, eikä niillä voitaisi laskea ilman tyyppimuunnoksia.

Tyyppimuunnoksista ja merkkijonoilla operoinnista puhumme enemmän ensi viikolla, siihen asti riittää, että tiedät kuinka input()-funktiolla voidaan ottaa käyttäjältä syötteinä lukuja.

Pythonin käyttö laskimena

PSF

Tulkkia voi käyttää myös kuten yksinkertaista laskinta. Annat tulkille operandit ja operaattorin ja tulkki tulostaa sinulle vastauksen. Voit myös kokeilla komentorivitulkin (tai 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
>>> # Kokonaisluvuilla jakolasku tuottaa kokonaislukuvastauksen.
... 7/3
2
>>> 7%3
1
```

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

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

Yhtäaikainen sijoitus toimii myös normaalisti:

```
>>> x = y = z = 0  # Nollaa x, y ja z
>>> x
0
>>> y
0
>>> z
0
```

Kokonaislukuja, pitkiä 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
```

Myös kompleksiluvuille on tuki Pythonissa. Imaginääriosa erotetaan luvusta jälkiliitteellä `“j”` tai `“J”`. Kompleksiluvut, joiden reaaliosa on erisuuri kuin 0 kirjoitetaan muotoon "`(`*`real`*`+`*`imag`*`j)`", tai luodaan komennolla "`complex(`*`real, imag`*`)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Kompleksiluvun osat ovat aina liukulukuja ja niitä voidaan käsitellä erikseen viittaamalla niihin `luku.real` ja `luku.imag` – muodoilla.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Tyyppimuunnosfunktiot (`float()`, `int()` and `long()`) eivät toimi kompleksilukujen kanssa. Kuten normaalisti, kompleksiluku on mahdotonta esittää normaalina reaalilukuna.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>>
```

Luku 3: Merkkijonot ja tyyppimuunnokset

Alkuperäislähde BSF tutorial, luku 3 sekä How to think like a computer scientist; Learning with Python, luku 3.2

Merkkijonot ja leikkaukset PSF

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

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

Kuten aiemmin mainittiin, merkkijonot voivat levittäytyä useille reveille monin eri tavoin. Fyysistä rivinvaihtoa voidaan merkitä ohitusmerkillä, tai käyttää kolmea sitaattimerkkiä:

```
heippa = "Tämä on erittäin pitkä merkkijono joka\n\
      jatkuu useille riveille ihan kuten aiemmissa esimerkeissä.\n\
      Huomaa, että tehdäksesi tällaisen rivin joudut huolehtimaan, \n\
      että se on sisennetty eri tasolle kuin sijoitusoperaattori."

print heippa
```

Huomioi että tarvitset edelleen kenoviivan merkitsemään loogisen rivin jatkumista, vaikka rivi itsessään päättyisikin rivinvaihtoon. Tämä esimerkki tulostaa seuraavanlaisen tuloksen:

```
Tämä on erittäin pitkä merkkijono joka
jatkuu useille riveille ihan kuten aiemmissa esimerkeissä.
Huomaa, että tehdäksesi tällaisen rivin joudut huolehtimaan,
että se on sisennetty eri tasolle kuin sijoitusoperaattori.
```

Jos käyttäisimme raakamerkkijonoa, rivinvaihtomerkkiä “\n” ei tulkittaisi rivinvaihdoksi, vaan se näkyisi tulostuksessa normaalisti merkkimuodossaan.

Merkkijonoja voidaan yhdistellä ‘+’ operattorilla ja toistaa ‘*’-operaattorilla.:

```
>>> sana = 'Apu' + 'VA'
>>> sana
'ApuVA'
>>> '<' + sana*5 + '>'
'<ApuVAApuVAApuVAApuVAApuVA>'
```

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

```
>>> sana[4]
'A'
>>> sana[0:2]
'Ap'
>>> sana[2:4]
'uV'
>>> sana[0:5:2]
'AuA'
```

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

```
>>> sana[:2]    # Ensimmäiset kaksi kirjainta
'Ap'
>>> word[2:]    # Kaikki muut kirjaimet paitsi kaksi ensimmäistä
'uVA'
```

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

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

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

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

```
>>> 'x' + sana[1:]
'xpuVA'
>>> 'Splat' + sana[4]
>>>
'SplatA'
```

Tai vaihtoehtoisesti

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

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

```
>>> sana[:2] + sana[2:]
'ApuVA'
>>> sana[:3] + sana[3:]
'ApuVA'
```

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

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

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

```
>>> sana[100]

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    sana[100]
IndexError: string index out of range
>>>
```

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

```
>>> sana[-1]     # Viimeinen merkki
'A'
>>> sana[-2]     # Toiseksi viimeinen merkki
'V'

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

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

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

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

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

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

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

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

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

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

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

Tyyppimuunnokset

LwP

Pythonin mukana tulee sisäänrakennettuna funktiot, joiden avulla muuttujan tyyppi voidaan muutta yhdestä toiseksi jos se on yksikäsitteisesti mahdollista toteuttaa. Esimerkiksi int() muuttaa annetun syötteen kokonaisluvuksi ja str() merkkijonoksi. Lisäksi liukuluvulle ja pitkille kokonaisluvuille on olemassa omat tyyppimuunnosfunktiot:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

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

```
>>> int(3.99999)
3
>>> int(“-267”)
-267
```

Float() muuttaa kokonaislukuja ja numeerisia merkkijonoja liukuluvuiksi:

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

Str() muuttaa annettuja syötteitä merkkijonoiksi:

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

Huomautus tyyppimuunnoksista

LTY

Tyyppimuunnoksiin liittyy kuitenkin yksi tekninen yksityiskohta, joka tulee huomioida niitä käytettäessä:

```
>>> luku = "2323" #Alustetaan luku merkkijonona
>>> int(luku)
2323
>>> luku +1 #Muuttuja ei tyyppimuunnoksesta huolimatta ole luku
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in -toplevel-
    luku +1
TypeError: cannot concatenate 'str' and 'int' objects
>>> luku = int(luku) #Tallennetaan muutos
>>> luku + 1 #Nyt lukuun voidaan lisätä
2324
>>>
```

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

Muotoiltu tulostus

Python-ohjelmointikielessä on myös lisätoimintoja, joiden avulla pystymme tarkentamaan sitä, kuinka ohjelma tulostaa merkkijonoja. Tätä sanotaan formatoiduksi, eli muotoilluksi tulostukseksi.

Joissain tapauksissa saatamme haluta, että tulostettava merkkijono noudattaa jotain tiettyä kaavaa huolimatta siitä, millainen varsinainen tulos on. Esimerkiksi ”nelinumeroinen luku ” tai ”kaksi-desimaalinen liukuluku” ovat hyviä esimerkkejä tästä. Teknisesti tämä toteutetaan siten, että print-käskylla annettavaan tulostussyötteeseen sijoitetaan muotoilumerkit %-merkillä erotettuna, ja syötteen perään sijoitetaan muuttujalista. Esimerkiksi muoto

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

Muuttuu nyt muotoon

```
print "%s ajaa maitoautoa." %(sana)
```

Siitä huolimatta molempien tuloste on identtinen, eli

```
maitomies ajaa maitoautoa.
```

Tulosteeseen sijoitettu merkki %s tarkoittaa, että tämän merkin paikalle tullaan sijoittamaan merkkijono (s, string). Vastaavasti, jos olisimme halunneet sijoittaa paikalle vaikka kokonaisluvun, olisi muotoilumerkkinä ollut %i (integer, myös %d):

```
luku = 13
sana = "Maitomies"

print "%s käy reitillään läpi %i taloa."%(sana, luku)
```

Tällä saisimme vastaukseksi tulosteen

```
Maitomies käy reitillään läpi 13 taloa.
```

Merkki	Tyyppi	Merkki	Tyyppi
%i	Kokonaisluku	%s	Merkkijono
%d	Kokonaisluku	%c	Yksittäinen merkki
%f	Liukuluku	%u	Liukuluku

Huomioi, että muuttujalistaan merkitään muuttujat siinä järjestyksessä, kuin ne tulostuksessa halutaan esiintyvän. Jos käytämme keskenään yhteensopimattomia muotoilumerkkejä, kuten %i ja listalla on merkkijono, aiheutamme tulkille virheen.

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

```
luku = 13.234266223435345134636643523542342 34

print "%.1f %.2f %.4f %.7f"%(luku,luku,luku,luku)
```

Tämä antaa tuloksena seuraavanlaisen tulosteen:

```
13.2 13.23 13.2343 13.2342662
```

Voimme siis säädellä vapaasti, kuinka desimaalia luvusta näytetään. Kannattaa lisäksi huomata, että kyseinen menetelmä osaa lisäksi pyöristää luvut oikein toisin kuin esimerkiksi kokonaislukujako. Jos yritämme samaa kokonaisluvuilla, saamme seuraavanlaisen tuloksen:

```
luku = 13

print "%.1i %.2i %.4i %.7i"%(luku,luku,luku,luku)
```

Tulostaa

```
13 13 0013 0000013
```

eli tässä tapauksessa ohjelma ainoastaan tulostaa lisää etunollia. Vastaavasti, jos lisäämme pisteen eteen luvun, kerromme kuinka monta merkkiä tilaa luvulle on vähintään varattava:

```
luku = 13

print "%1i %2i %4i %7i"%(luku,luku,luku,luku)
```

Tulostaa

```
13 13   13      13
```

Kuten huomaamme, tämä ei koske tilannetta, jossa luku on pidempi kuin vähintään varattava tila. Luonnollisesti lukuja voidaan myös yhdistää:

```
luku = 13.234266223435345134636643523542342 34
print "%3.1f %4.1f %5.1f %6.1f"%(luku,luku,luku,luku)
```

Antaa tuloksena tulosteen:

```
13.2 13.2  13.2   13.2
```

Luku 4: Koodin haarautuminen

Alkuperäislähde Byte of Python, luku 6

Ohjausrakenteet

BoP

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

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

If-rakenne

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

If-rakenteen käyttäminen

Esimerkki 4.1: Käytetään if -rakennetta

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: if.py

numero = 23
arvaus = input('Anna kokonaisluku : ')

if arvaus == numero:
        print 'Arvasit oikein!' # Tästä alkaa uusi osio, huomaa sisennys
        print "Peli päättyy tähän" # Osio päättyy tähän
elif arvaus < numero:
```

```
        print 'Luku on suurempi kuin arvaus' # Toinen osio
        #Tähän voisit jatkaa koodia joka ajetaan kun arvaus on
        # yläkanttiin.
else:
        print 'Luku on pienempi kuin arvaus'
        # Tähän voisit jatkaa koodia joka ajetaan kun arvaus on
        # alakanttiin.

print "Tämä tulostuu aina koska tämä on if-rakenteen jälkeen"
```

Tuloste

```
>>>
Anna kokonaisluku : 10
Luku on suurempi kuin arvaus
Tämä tulostuu aina koska tämä on if-rakenteen jälkeen
>>>
Anna kokonaisluku : 30
Luku on pienempi kuin arvaus
Tämä tulostuu aina koska tämä on if-rakenteen jälkeen
>>>
Anna kokonaisluku : 23
Arvasit oikein!
Peli päättyy tähän
Tämä tulostuu aina koska tämä on if-rakenteen jälkeen
>>>
```

Kuinka se toimii

Tämä ohjelma vastaanottaa arvauksia käyttäjältä ja kertoo, onko käyttäjän arvaus yli vai alle määritellyn vastauksen, joka on kiintoarvo muuttujassa `numero`. Funktio input() huolehtii siitä, että se kysyy käyttäjältä lukua, ja tallentaa sen muuttujaan `arvaus`.

Tämän jälkeen ohjelma etenee if-rakenteeseen ja ensimmäisenä testaa, onko annettu muuttujan arvo arvaus sama kuin määritelty numero. Jos on, siirtyy ohjelma kaksoispisteen jälkeen alkavaan sisennyksellä eroteltuun osioon, jossa käyttäjää onnitellaan. Jos taas arvaus ei ole oikein, siirrytään if-rakenteessa eteenpäin elif-rakenteeseen, jossa kokeillaan onko arvaus vastausta pienempi. Jos tämä väittämä pitää paikkansa, niin siirrymme kaksoispisteen jälkeen alkavaan sisennettyyn osioon, jossa käyttäjälle kerrotaan tulos. Jos luku ei myöskään ole pienempi, siirrytään eteenpäin rakenteessa, jossa vastaan tulee lopetusrakenne else, joka ajetaan aina jos if tai yksikään elif-rakenteen vaihtoehto ei ole kelvannut. Tässä tapauksessa luku voi ainoastaan olla pienempi, ja tämä kerrotaan käyttäjälle. If-rakenteen päätyttyä ajo jatkuu normaalisti seuraavalta if-rakenteen tason riviltä.

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

```
if True:
        print 'Kyllä, arvo True on aina totta.'
```

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

Esimerkki 4.1: If-elif-else-rakenne, kertaus LTY

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

vari = "punainen"

if vari == "sininen":
   print "Väri on sininen"

elif vari == "vihreä":
   print "Väri on vihreä"

elif vari == "punainen":
   print "Väri on punainen"

else:
   print "En tunnista väriä."
```

Kuinka se toimii

Tässä esimerkissä käytämme yksinkertaisella tavalla if-elif-else-rakennetta. Rakenne kertoo käyttäjälle, minkä värin hän on muuttujaan `vari` valinnut. Jos `vari` olisi arvoltaan ”*sininen*”, katkeaisi rakenteen läpikäynti ensimmäiseen osioon. Koska `vari` ei ole ”*sininen*”, siirrytään ensimmäiseen elif (else-if, muutoin-jos)-rakenteeseen. Tässä testaamme onko arvo ”*vihreä*”. Jos muuttuja `vari` olisi ”*vihreä*”, loppuisi toisto tähän osioon.

Koska elif-osioita voidaan ketjuttaa peräkkäin useita kappaleita yhden if-osion perään, kokeilemme toisella elif-osiolla onko muuttujan `vari` arvo ”*punainen*”. Koska tällä kertaa osumme oikeaan, lopetetaan if-rakenteen läpikäynti tähän. Lopussa olemme vielä määritelleet else-osion, joka suoritettaisi siinä tapauksessa jos yksikään if- tai elif-osio ei olisi toteutunut. Tässä tapauksessa näin olisi päässyt käymään vaikkapa silloin, jos muuttujan `vari` arvoksi olisi annettu ”*keltainen*”.

Huomioita if-rakenteesta

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

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

Tulostaisi vastauksen

```
Otit kolmipyörän.
```

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

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

Tulostaisi vastauksen

```
Et ottanut rahapalkintoa.
```

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

```
palkinto = "kolmipyörä"
if palkinto == "rahapalkinto":
    print "Otit rahat."

elif palkinto == "kolmipyörä":
    print "Otit kolmipyörän."
```

Tulostaisi vastauksen

```
Otit kolmipyörän.
```

Tässä yhteydessä emme tarvitse else-osiota mihinkään - oletetaan vaikka että ainoat meitä kiinnostavat vaihtoehdot ovat *rahat* tai *kolmipyörä* - joten voimme jättää else-osion huomioimatta. Käytännössä tämä tarkoittaisi samaa kuin merkinnän

```
else:
    pass
```

lisääminen rakenteen loppuun, mutta tästä puhumme lisää seuraavassa luvussa.

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

```
luku = 50

if luku < 10:
    tulos = "pienempi kuin 10."
if luku < 100:
    tulos = "pienempi kuin 100."
if luku < 1000:
    tulos = "pienempi kuin 1000."
else:
    tulos = "suurempi 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 kuin 1000."
print tulos
```

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

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

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

Boolean-arvoista

Python tukee loogisten lausekkeiden kanssa työskennellessä Boolean-arvoja, jotka voivat saada joko arvon tosi (`True`) tai epätosi (`False`). Nämä arvot ovat käytännössä monimutkaisempi esitystapa numeroarvoille 1 (`True`) ja 0 (`False`), ja niitä käytetäänkin lähinnä koodin luettavuuden parantamiseen.

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

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

Boolean –arvojen käyttö on loogisissa lausekkeissa luonnollisempaa kuin pelkkien numeroiden, mutta useimmiten niiden välillä ei ole syntaktista eroa. Useimmiten arvot True ja False voidaankin korvata myös numeroilla 1 ja 0.

Luku 5: Toistorakenteet

Alkuperäislähde Byte of Python luku 6 sekä Python Software Foundation Tutorial luku 4

While-rakenne

BoP

While-rakenne mahdollistaa sen, että voit toistuvasti läpikäydä samaa osiota, kunnes ohjelma täyttää sille asetetut ehdot. While-rakenne on esimerkki toistorakenteista, joita Pythonissa on sisäänrakennettuna kaksi erilaista. While-rakenteelle voi myös määritellä else-osion, joka ajetaan jos while-osioa ei koskaan käynnistetä tai kun se on suoritettu loppuun.

While-rakenteen käyttäminen

Esimerkki 5.1 Numeronarvauspeli while-rakenteen avulla

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: while.py

numero = 23
kesken = True
while kesken: #Jatketaan kunnes kesken saa arvon False
    arvaus = input('Anna kokonaisluku : ')

    if arvaus == numero:
            print 'Arvasit oikein!'
            print "Peli päättyy tähän"
            kesken = False #Pelaaja arvasi oikein, silmukka lopetetaan.

    elif arvaus < numero:
            print 'Luku on suurempi kuin arvaus' # Toinen osio

    else:
            print 'Luku on pienempi kuin arvaus'

else:
    print "Silmukka on suoritettu loppuun, muuttuja 'kesken' sai
arvon",kesken

print "Tämä tulostuu loppuun koska 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
Silmukka on suoritettu loppuun, muuttuja kesken sai arvon False
Tämä tulostuu loppuun koska se on while-rakennetta seuraava looginen
rivi.
>>>
```

Kuinka se toimii

Tämä ohjelma kehittää edellisen viikon numeronarvauspeliä eteenpäin ja tällä kertaa poistammekin siitä ärsyttävän piirteen, joka pakotti käyttäjen aina käynnistämään pelin uudelleen jokaisen arvauksen jälkeen. Tämä onnistuukin helposti while-rakenteen avulla.

Ensinnäkin käyttäjältä arvausta pyytävä käsky siirrettiin silmukkaan sisälle. Tämän ansiosta käyttäjältä kysytään jokaisen kierroksen alussa, mitä hän haluaisi tällä kertaa arvata. Lisäksi ennen silmukkaa esittelemme muuttujan `kesken`, joka toimii lippumuuttujana. `Kesken` saa ennen silmukkaa arvon `True`, jonka ansiosta silmukka lähtee pyörimään ja jatkaa pyörimistä kunnes lippumuuttaja saa arvon `False`. Joka kierroksella tulkki tarkastaa, onko ehtomuuttuja vielä voimassa ja jos on, aloittaa uuden kierroksen. Jos taas ei ole – mikä tapahtuu kun käyttäjä arvaa oikein - siirrytään else-osioon. Else-osion vahvuus on siinä, että joskus voi tulla tilanne, jossa toistorakenteen ehto on valmiiksi `False`, jolloin voidaan suoraan siirtyä else-rakenteeseen. Kannattaa myös pitää mielessä, että tämä osio on if-rakenteen tapaan vapaa-ehtoinen, while-rakenne ei vaadi else-rakennetta toimiakseen. Lopuksi tulkki jatkaa eteenpäin while-rakenteesta ja tulostaa viimeisen rivin.

Huomioita while-rakenteesta

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

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

Tulostaisi vastauksen

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

Huomaa, että tässä tapauksessa teemme testauksen edelleen samalla periaatteella kuin aiemmin. Joka kerta kun kierros täyttyy, tarkastamme onko muuttujan `arvo` sisältämä numeroarvo pienempi kuin 4. Kierroksella, jolla tulostamme 3, kasvatamme muuttujan arvoa yhdellä, joten toistoehto ei enää toteudu.

For-rakenne

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

Esimerkki 5.2 for-rakenteen toiminta

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: for.py

for i in range(1, 5):
        print i
else:
        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 [1,2,3,4], jossa siis on neljä jäsentä. For-rakenne käy läpi nämä neljä jäsentä – toistorakenne tapahtuu neljä kertaa - jonka jälkeen ohjelma jatkaa normaalia kulkuaan seuraavalta loogiselta riviltä, joka tulostaa kommentin silmukan päättymisestä.

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

Myös for-rakenteeseen voi myös liittää else-osion, joka toimii samoin kuin while-rakenteessa.

Break-käsky

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

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

Esimerkki 5.3. Toistorakenne break-käskyllä

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: break.py

while True:
        merkkijono = raw_input('Kirjoita jotakin: ')
        if merkkijono == 'lopeta':
                break
        print 'Merkkijono oli', len(merkkijono), "merkkiä pitkä."
else:
    print "Tätä te ette koskaan tule näkemään."
print 'Loppu!'
```

Tuloste

```
>>>
Kirjoita jotakin: Joku porho kehui
Merkkijono oli 16 merkkiä pitkä.
Kirjoita jotakin: repineensä puukaupoilla tuohta
Merkkijono oli 30 merkkiä pitkä.
Kirjoita jotakin: niin paljon, että
Merkkijono oli 17 merkkiä pitkä.
Kirjoita jotakin: heikkolatvaista puistatti.
Merkkijono oli 26 merkkiä pitkä.
Kirjoita jotakin: lopeta
Loppu!
>>>
```

Kuinka se toimii

Ohjelman toiminta on varsin yksinkertainen. `while`-rakenne asetetaan toimimaan niin kauan, kunnes `break`-käsky saaavutetaan, koska muita keinoja lopetusehdon saavuttamiseen ei anneta. Tämän jälkeen käyttäjältä pyydetään toistuvasta syötettä, joka tulostetaan ja tulosteesta kerrotaan sen merkkimäärä, kunnes syötteeksi annetaan

“lopeta”. Kun käyttäjä antaa arvon “lopeta”, täyttyy `break` – käskyä edeltävä if-rakenteen ehto ja toistorakenne katkeaa.

Tehtävästä kannattaa huomioida kaksi asiaa: ensinnäkin se, että vaikka while-rakenteelle annettu ehto on staattisesti `True`, katkeaa toistorakenne break-käskyllä pysyvästi vaikka katkaisuehto ei olekkaan tullut voimaan (eli saanut arvoa `False`). Lisäksi kannattaa huomata, että vaikka esimerkkikoodi sisältää else-osion, ei siihen koskaan mennä. Pythonin syntaksi olettaa, että toistorakenteen katkaiseminen break-käskyllä tarkoittaa, että toistorakenne on täyttänyt sille annetun tehtävän, eikä täten ”muussa tapauksessa” – osiota tule ajaa.

Muista myös, että break-lause toimii myös for-rakenteen kanssa.

Continue-käsky

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

Continue-käskyn käyttäminen

Esimerkki 5.4 Toistorakenne continue-käskyllä

```
# -*- coding: cp1252 -*-
# Tiedostonnimi: continue.py

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

Tuloste

```
>>>
Syötä tekstiä: hei
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 taas syöttää lopetuskäskyn ”lopeta”, ohjelma katkaisee toistorakenteen `break`-käskyllä.

Myös `continue` toimii for-rakenteen kanssa.

Pass-käsky

PSF

Kolmas kontrollirakenteiden kanssa toimiva käsky on `pass`. Se ei varsinaisesti tee mitään vaan on varattu käytettäväksi siinä tapauksessa, jos ohjelmoija tarvitsee syntaksin säilyttämiseksi osion rakenteeseensa, mutta sille ei haluta antaa mitään tehtävää. Näin voi käydä esimerkiksi kun haluamme loogisesti käyttää else-osioa ilman if-osion toiminnallisuuksia. Esimerkiksi silloin, kun haluamme varoittaa käyttäjää virheellisestä muuttujan arvosta, mutta oikein asetetuilla arvoilla emme halua toimenpiteitä, on pass-käsky yksi vaihtoehto.

Esimerkki 5.5 Pass-käsky toistorakenteessa

```
# -*- coding: cp1252 -*-
# Tiedostonnimi: pass.py

for i in range(0,8):
    if i % 2 == 0:
        pass
    else:
        print i,"on pariton luku."
```

Tuloste

```
>>>
1 on pariton luku.
3 on pariton luku.
5 on pariton luku.
7 on pariton luku.
>>>
```

Kuinka se toimii

Käytännössä tämä esimerkki on ainoastaan malli siitä, kuinka pass-lausetta voidaan käyttää apuna joissain tilanteissa. For-silmukka läpikäy luvut 1-7, ja testaa, onko luku kahdella jaollinen, eli parillinen. Jos on, kierroksella ei tehdä mitään – se ohitetaan pass-käskyllä - jos taas ei, tulostetaan luku ja maininta siitä, että se on pariton. Käytännössä pass-käsky voidaan useimmiten myös kiertää, esimerkiksi tässä tapauksessa if-lause voidaan laittaa testaamaan parittomuutta (i % 2 != 0), jolloin else-rakennetta ei tarvita lainkaan. Kuitenkin pass-käskylle löytyy myöhemmin joitakin käyttötilanteita, jossa se on helpoin ja yksinkertaisin tapa ratkaista ongelma.

Range()-funktiosta

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

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

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

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

>>> range(-10, -100, -30)
[-10, -40, -70]
```

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

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


k u m i a n k k a
>>>
```

Huomaa, että pilkku print-käskyn lopussa tarkoittaa sitä, että seuraava tulostus tehdään tämän lauseen perään.

Else-osio toistorakenteessa

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

Esimerkki 5.6 Else-osio toistorakenteessa, alkuluvut

```
# -*- coding: cp1252 -*-
#tiedostonnimi: alkuluku.py

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

Tuloste

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

Luku 6: Funktiot

Alkuperäislähde Byte of Python, luku 7 sekä Python Software Foundation tutorial, luvut 4.6-4.7

Funktiot ja niiden käyttäminen

BoP

Funktiot ovat ”koodinpätkiä”, joita voidaan käyttää uudestaan moneen kertaan. Niillä on mahdollista nimetä koodiin osio, jolle annetaan syötteet, joiden avulla tämä osio – funktio - suorittaa sille määritellyt toimenpiteet ja palauttaa laskemansa vastauksen. Funktion käyttöönottamista sanotaan funktion kutsumiseksi. Arvo, 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. Funktiota, joka hoitaa ohjelman ajamisen sanotaan pääohjelmaksi ja funktioita, jotka ovat yksittäisiä aliongelmia taas aliohjelmiksi. Python ei ole funktiorakenteen kanssa niin tarkka kuin esimerkiksi C-kieli, mutta on hyvän ohjelmointitavan mukaista, että kaikki kirjoitetut käskyt ja loogiset lausekkeet on sijoitettu funktioiden sisään.

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

Funktion määritteleminen

Esimerkki 6.1 Funktion määritteleminen ja kutsuminen

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: function1.py

def sanoTerve(): #Aliohjelma alkaa
        print 'Terve vaan!'
        print 'Tämä tulostus tulee funktion sisältä!' #Aliohjelma loppuu

#Pääohjelma alkaa
sanoTerve() # Funktiota kutsutaan sen omalla nimellä
#Pääohjelma loppuu
```

Tulostus

```
>>>
Terve vaan!
Tämä tulostus tulee funktion sisältä!
>>>
```

Kuinka se toimii

Edellä loimme funktion nimeltä sanoTerve() käyttäen ohjeita, jotka aiemmin kävimme lävitse. Tämän funktion kutsussa ei ole muuttujia, joten sitä voidaan kutsua suoraan omalla nimellään. Kutsussa käytettävät funktiotkaan eivät varsinaisesti ole erityisen monimutkainen asia; ne käytännössä ovat muuttujia, joiden avulla voimme antaa tietoa funktiosta toiseen.

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

```
print ”Toistetaan”
sanoTerve()
print ”Uudelleen”
sanoTerve()
```

Tämän jälkeen tulostus näyttää tällaiselta:

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

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

Funktioita pystyy uudelleen käyttämään myös komentorivitulkissa tai shell-ikunassa. Kirjoittamalla funktion nimen shell-ikkunaan tai komentorivitulkkiin voimme välittömästi uudelleenajaa funktion ilman, että joudumme palaamaan takaisin itse

lähdekooditiedostoon. Erityisen hyödyllistä tämä on silloin, kuin pääohjelma on oma funktionsa jolloin ohjelma voidaan käynnistää kutsumalla pääfunktiota.

Funktiokutsu ja parametrien välitys

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

Parametrin määrittely tapahtuu funktion määrittelyssä olevien sulkujen sisään. Jos halutaan antaa useampia parametreja, tulee ne erotella toisistaan pilkuilla. Funktiokutsussa vastaavasti parametreille annettavat arvot syötetään samassa järjestyksessä. Kannattaa myös huomioida terminologiasta seuraava yksityiskohta: useasti funktiokutsun arvoja sanotaan parametreiksi, mutta niiden saamia arvoja nimitetään usein funktion argumenteiksi. Selvyyden vuoksi puhumme jatkossa pelkästään parametreista.

Parametrien käyttäminen

Esimerkki 6.2 Funktio, jolle annetaan parametreja

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: funktio2.py

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

sanoTerve('Brian','Tite',’4’ ) # Annetaan kutsussa parametreja
```

Tuloste

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

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

Tässä tapauksessa tulkki yksinkertaisesti siirtää muuttujien a,b ja c arvot funktiokutsun parametreiksi.

Nimiavaruudet

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

Paikalliset muuttujat käytännössä

Esimerkki 6.3 Paikallisten muuttujien toimivuus

```
# -*- coding: cp1252 -*-
# Tiedoston nimi: func_local.py

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

x = 50
print 'x on ennen funktiota',x
func(x)
print 'x on funktion kutsumisen jälkeen edelleen', x
```

Tuloste

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

Kuinka se toimii

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

Yhteiset muuttujat

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

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

Esimerkki 6.4 Yhteisen muuttujan käyttäminen

```
# -*- coding: cp1252 -*-
# Tiedostonnimi: func_global.py

def funktio():
        global x

        print 'x on', x
        x = 2
        print 'yhteinen x muutettiin funktiossa arvoksi', x

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

Tuloste

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

Kuinka se toimii

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

Parametrien oletusarvot

Joskus funktion kutsussa saamien parametrien arvot on määritelty valinnaisiksi. Lisäksi osa funktioista on rakennettu siten, että kutsuna parametreilla on oletusarvot tapauksissa, joissa käyttäjä ei muista tai halua antaa arvoa. Tämä voidaan Pythonissa toteuttaa määrittelemällä funktiokutsuun parametreille oletusarvot. Oletusarvojen määrittely ei poikkea paljoakaan esimerkiksi muuttujien arvojen määrittelystä. Oletusarvo syötetään sijoitusoperaattorilla (‘=’) suoraan funktiokutsuun. Alla oleva esimerkki näyttää, kuinka tämä käytännössä tapahtuu:

Esimerkki 6.5 Oletusarvot funktiokutsussa

```
# -*- coding: cp1252 -*-
# Filename: func_oletus.py

def sano(viesti, kerroin = 1):
        if kerroin == 1:
            print "Et antanut kerrointa -> käytetään vakiota 1"
        print viesti * kerroin

sano('Nakkipasteija')
sano('Sinappisiideri', 5)
```

Tuloste

```
>>>
Et antanut kerrointa -> käytetään vakiota 1
Nakkipasteija
SinappisiideriSinappisiideriSinappisiideriSinappisiideriSinappisiideri
>>>
```

Kuinka se toimii

Funktiokutsussa on kaksi osaa: tekstiosa, joka tallentuu parametriin *viesti*, sekä toistomäärä, joka tallennetaan muuttujaan *kerroin*. Toistomäärä on vapaaehtoinen ja mikäli sille ei anneta arvoa, käytetään vakioa 1 sekä ilmoitetaan käyttäjälle asiasta. Funktiokutsussa annetaan käytettävät arvot, joista kaikki ei-oletusarvolliset ovat pakollisia.

Huomautus

Funktiokutsun parametrilistan järjestyksessä on huomioitava muutama tärkeä yksityiskohta. Tärkein huomio näistä on se, että oletusarvolliset parametrit on aina sijoitettava viimeisiksi parametreiksi, jotta funktiokutsu on yksiselitteinen. Koska vakioarvollisia parametreja käsitellään valinnaisina parametreina, ei niiden jälkeen voida enää esitellä uusia ei-vakioparametreja, koska silloin tulkki ei tiedä mikä arvo kuuluu millekin parametrille. Käytännössä tämä siis toimii siten, että

`def func(a, b=5)` on oikein, koska

`func(“sana”)` –kutsussa “sana” tulee aina parametrille a ja b saa aina arvon 5. Sen sijaan muoto

`def func(a=5, b)` on väärin, koska

`func('sana')` voidaan tulkita siten, että kutsussa a saa arvon ”sana” ja b ei saa arvoa lainkaan; tästä aiheutuu virhe. Käyttäjä tarkoittaa luultavasti että a on 5 ja b on ”sana”, mutta koska järjestys on sekaisin, ei tulkki tiedä kuuluisiko arvo ”sana” parametrille a vai b.

Parametrien avainsanat

Jos sinulla on käytössä funktio, joka sisältää monta vakioarvoista parametria, etkä haluaisi antaa kuin yhden tai kaksi parametria, voit käyttää avainsanoja. Avainsanojen avulla pystyt määräämään missä järjestyksessä annat parametrit, sekä sen, mitkä parametrit saavat minkäkin arvon. Käytännössä tämä siis tarkoittaa, että argumenttien sijoitusjärjestys muutetaan sijaintiin perustusvasta avainsanoihin:

Avainsanojen käyttäminen

Esimerkki 6.6 Avainsanat funktiokutsuissa

```
# -*- coding: cp1252 -*-
# Filename: funk_avain.py

def luvut(a, b=5, c=10):
        print 'a on', a, 'ja b on', b, 'ja c on', c
```

```
luvut(3, 7)
luvut(25, c=24)
luvut(c=50, a=100, b=30)
```

Tuloste

```
>>>
a on 3 ja b on 7 ja c on 10
a on 25 ja b on 5 ja c on 24
a on 100 ja b on 30 ja c on 50
>>>
```

Kuinka se toimii

Funktiolla *luvut* on yksi parametri, joka ei saa oletusarvoa funktiokutsussaan, sekä kaksi parametria, joiden oletusarvot ovat 5 ja 10. Ensimmäinen funktiokutsu, `luvut(3, 7)`, kutsuu funktiota normaaliin tapaan antaen a:lle arvon 3, b:lle arvon 7 ja jättäen c:n käyttämään vakioarvoa. Toisessa kutsussa toinen parametri - 24 - on avainsanan avulla ohjattu menemään parametrille c, vaikka se sijaintinsa puolesta olisi kuulunut parametrille b.

Viimeisessä kutsussa on esitelty muoto, jossa järjestys on rikottu täysin avainsanojen avulla. Mikäli avainsanoja käytetään, on funktiokutsun muoto vapaa, kunhan jokaiselle pakolliselle parametrille on avainsanan avulla määritelty arvo.

Paluuarvo

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

Paluu-arvon hyödyntäminen

Esimerkki 6.7 Paluu-arvo käytännössä

```
# -*- coding: cp1252 -*-
# Filename: func_return.py

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

```
a = 100
b = 50
suurempi = maksimi(a,b)
print "Suurempi arvo on",suurempi
```

Tuloste

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

Kuinka se toimii

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

Huomaa, että tässä tapauksessa alifunktion ja päätason ei tarvitse sotkea keskenään nimiavaruuksia johtuen siitä, että päätason muuttuja *suurempi* saa sijoituksena funktio *maksimi*:n palautusarvon. Tämä mahdollistaa tehokkaan koodin tekemisen ilman että joutuisimme käyttämään yhteisiä muuttujia.

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

Funktioiden dokumentaatiorivit ja help()-funktio

LTY

Luodaan ohjelma, joka laskee Fibonaccin lukusarjan lukuja:

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

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

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

Dokumentaatiorivi toimii siten, että kun IDLEssä kirjoitat funktiokutsua kyseisestä rivistä, näet automaattisesti aukenevassa apuruudussa funktiokutsun mallin niin kuin kirjoitit sen itse määrittelyyn, sekä sen alapuolella kirjoittamasi dokumentaatiorivin. Rivin käyttö ei ole pakollista, mutta se helpottaa tuntemattomien apufunktioiden käyttöä ja on hyvän ohjelmointitavan mukaista. Dokumentaatiorivi ei erityisesti vaadi kolmen sitaatin kombinaatiota toimiakseen, mutta se mahdollistaa tehokkaat moniriviset ohjeet.

Dokumentaatiorivi näkyy myös komentorivitulkissa tai shell-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. Tulkki lopetetaan painamalla jättämällä rivi tyhjäksi ja painamalla enter tai kirjoittamalla ”quit” ja painamalla enter.

Luku 7: Ulkoiset tiedostot

Alkuperäislähteet Python Software Foundation tutorial, luku 7.2 sekä Byte of Python, luku 12

Tiedostoista lukeminen ja niihin kirjoittaminen BoP

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

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

Tiedoston avaaminen

Käsiteltäessä tiedostoja olisi meidän hyvä ensin tietää mistä niitä löydämme. Jos käytämme IDLEn Shell-ikkunaa, on oletuskansio paikallinen Pythonin asennuskansio. Jos taas teemme töitä IDLEn editorilla tai käytämme komentorivitulkkia, on oletushakemisto se, mihin lähdekoodi on tallennettu (IDLE) tai se, mistä komentorivitulkki on käynnistetty. Oletuskansion vaihtaminen on myös mahdollista, mutta siihen palaamme myöhemmin. Nyt kun me tiedämme mihin tekemämme muutokset tallentuvat, voimme avata tiedoston, kirjoittaa sinne jotain sekä lukea aikaansaannoksemme:

Tiedostojen käyttäminen

Esimerkki 7.1 Tiedoston avaaminen, kirjoittaminen ja lukeminen

```
# -*- coding: cp1252 -*-
# Filename: tiedtesti.py

teksti = '''\
Balin palapelitehdas
pisti pillit pussiin pian
pelipalojen palattua piloille pelattuina:
        pelipalojen puusta puuttui pinnoite!
'''

tiedosto = open('uutinen.txt', 'w') # avataan kirjoitusta varten
tiedosto.write(teksti) # kirjoitetaan teksti tiedostoon
tiedosto.close() # suljetaan tiedosto

tiedosto = open('uutinen.txt', 'r') # avataan tiedosto lukemista varten
while True:
        rivi = tiedosto.readline() #Luetaan tiedostosta rivi
        if len(rivi) == 0: # Jos rivin pituus on 0, ollaan lopussa
                break
        print rivi,
tiedosto.close() # suljetaan tiedosto toisen ja viimeisen kerran
```

Tuloste

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

Kuinka se toimii

Tällä kertaa esimerkki sisältää paljon uutta asiaa, että tämä tehtävä läpikäydään rivi riviltä. Esimmäiset kaksi riviä ovat normaalit kommenttirivit, samoin ensimmäinen looginen rivi on vanhastaan tuttu, sillä ainoastaan tallennetaan testiteksti muuttujaan. Tämän jälkeen pääsemme ensimmäiselle tiedostoja käsittelevälle riville: `tiedosto = open('uutinen.txt', 'w')`. Ensinnäkin, avaamme tiedoston *uutinen.txt* tiedostokahvaan `tiedosto`, joka saa arvona tiedostojen avaamisessa käytetyn funktion `open()`-palautusarvon. Open-funktiota kutsutaan antamalla sille kaksi parametriä, joista ensimmäinen on avattavan tiedoston nimi, tässä tapauksessa merkkijono `uutinen.txt`. Toinen parametri on tila, johon tiedosto halutaan avata, tässä tapauksessa ‘`w`’, eli kirjoitustila (eng. write). Kirjoitusmoodi ei erikseen tarkasta onko aiempaa tiedostoa nimeltä “uutinen.txt” olemassa; jos tiedosto löytyy, se korvataan uudella tyhjällä tiedostolla, ja jos taas ei ole, sellainen luodaan automaattisesti. Koska ajamme

lähdekoodin tiedostosta tiedtesti.py, on oletuskansio automaattisesti sama kuin se, mihin ko. tiedosto on tallennettu.

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

- Ensinnäkin, tiedostokahva *tiedosto* ei varsinaisesti vielä laske meitä käsiksi tiedoston sisältöön, vaan on ainoastaan ”kulkuyhteys” tiedoston sisälle. Tiedostoa itsessään manipuloidaan kahvan kautta, ja siksi funktiot kuten write tai close merkataan pistenotaation avulla, koska ne ovat tiedostokahvan jäsenfunktioita. Tässä vaiheessa tärkeintä on kuitenkin vain muistaa lisätä piste muuttujannimen ja funktion väliin.
- Toiseksi kannattaa muistaa, että olemme avanneet tiedoston nimenomaisesti kirjoittamista varten. Nyt voimme kirjoittaa vapaasti mitä haluamme, mutta jos tila olisi valittu toisin –kuten esimerkiksi lukutila-, kirjoitusyritys aiheuttaisi virheilmoituksen.
- Kolmas asia on se, että **Python voi kirjoittaa tiedostoihin ainoastaan merkkijonoja**. Tämä tulee ottaa kirjaimellisesti; jos haluat tallentaa lukuarvoja tiedostoihin, joudut ennen kirjoitusta muuttamaan ne merkkijonoiksi. Tämä luonnollisesti onnistuu tyyppimuunnosfunktio str:llä.

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

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

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

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

Avaustiloista

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

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

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

Pythonista löytyy myös muita tiloja mm. binäärisen datan käsittelylle, ja niistä voit halutessasi lukea lisätietoa vaikkapa Python software foundationin dokumenteistä.

Kirjanmerkistä

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

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

Työkaluja tiedostonkäsittelyyn PSF

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

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

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

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

`readlines` on variaatio readline-funktiosta. Se palauttaa koko tiedoston rivit kirjanmerkin sijainnista tiedoston loppuun yhteen listaan tallennettuna, mikä käytännössä tarkoittaa sitä, että funktio kerran ajettaessa palauttaa koko tiedoston sisällön. Tälle funktiolle voidaan antaa parametri *sizehint*, joka määrää kuinka monta bittiä tiedostosta ainakin luetaan, sekä sen päälle riittävästi bittejä 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 onkin myös esimerkissä esitelty toistorakenne. Tämä tapa säästää muistia ja on yksinkertainen sekä nopea. Jos oletetaan, että sinulla on olemassa tiedostokahva f, voit tulostaa sen sisällön helposti komennolla:

```
>>> for rivi in f:
        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.

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

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

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

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

>>> f.read(1)
'd'
```

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

```
>>> f.close()
>>> f.read()
```

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

Tiedostojen käsittelyyn on myös muita työkaluja, mutta niiden käyttö on nyt esiteltyihin nähden marginaalista. Jos haluaisit tutustua niihin tarkemmin, löytyy niistäkin kattava esittely Python Software Foundationin dokumentaatioista.

Huomautus tiedoston avaamisesta

Joissain verkosta löytyvissä esimerkeissä saattaa tiedoston avaaminen tapahtua funktiolla `file`, joka näyttäsi toimivan täysin samalla tavoin kuin nyt esitelty `open`. Tämä ei ole virhe, vaan `open` ja `file` ovat itse asiassa samalla tavalla toimivat funktiot, joista toinen on jätetty toimimaan taaksepäin toimivan yhteensopivuuden varmistamiseksi. Molempia voi käyttää, ne ovat identtisiä ja toimivat samalla tavalla, mutta koodin yksinkertaisuuden nimissä suosittelemme käytettäväksi funktiota `open`.

Luku 8: Tietorakenteet ja komentoriviparametrit

Alkuperäislähde Python Software Foundation tutorial luvut 3.1.4 ja 5, sekä Byte of Python luku 9

Johdanto

BoP

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

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

Lista

Lista on tietorakenne, joka sisältää joukon alkioita. Tämä tarkoittaa, että voimme tallentaa alkiojoukkoja yhteen listaan. Helpointa tämä on mieltää siten, että ajattelemme esimerkiksi ostoslistaa jonka laadit kauppaan lähtiessäsi. Erottelet listan yksittäiset tuotteet – tässä tapauksessa siis alkiot - rivinvaihdoilla tai mahdollisesti ranskalaisilla viivoilla, kun taas Python tekee sen pilkuilla. Molemmissa tapauksissa tulos on kuitenkin sama: rivinvaihdoilla eroteltu lista tuotteista on käytännössä sama asia kuin pilkuilla eroteltu lista alkioita.

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

Listan käyttäminen

Esimerkki 8.1. Listan käyttäminen

```
# -*- coding: cp1252 -*-
# Filename: lista.py

# Ostoslistan määrittely
lista = ['omena', 'mango', 'porkkana', 'turtana']

print 'Tarvitsen vielä', len(lista), 'tuotetta.'

print 'Nämä tuotteet ovat:',
for tuote in lista:
        print tuote,

print '\nTarvitsen myös sihijuomaa.'
lista.append('sihijuoma')
print 'Nyt lista näyttää tältä', lista

print 'Järjestellään lista'
lista.sort()
print 'Järjestelty lista on tämän näköinen:', lista

print 'Ensimmäinen ostettava tuote on', lista[0]
ostettu = lista[0]
del lista[0]
print 'Ostin tuotteen', ostettu
print 'Nyt listalla on jäljellä', lista
```

Tuloste

```
>>>
Tarvitsen vielä 4 tuotetta.
Nämä tuotteet ovat: omena mango porkkana turtana
Tarvitsen myös sihijuomaa.
Nyt lista näyttää tältä ['omena', 'mango', 'porkkana', 'turtana',
'sihijuoma']
Järjestellään lista
Järjestelty lista on tämän näköinen: ['mango', 'omena', 'porkkana',
'sihijuoma', 'turtana']
Ensimmäinen ostettava tuote on mango
Ostin tuotteen mango
Nyt listalla on jäljellä ['omena', 'porkkana', 'sihijuoma', 'turtana']
>>>
```

Kuinka se toimii

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

Huomioi myös, että käytimme hieman tavallisuudesta poikkeavaa `for..in` – toistorakennetta listan läpikäymiseen. Tämä siis tarkoittaa sitä, että lista on eräänlainen sarja (sequence), joten sitä voidaan käyttää yksinään for-lauseen määrittelemisessä. Jos annamme range-funktion tilalle listan, for-lause käykin läpi niin monta kierrosta kuin annetussa listassa oli alkioita. Palaamme tähän asiaan myöhemmin.

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

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

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

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

Esimerkki 8.2. Listan leikkaukset

```
# -*- coding: cp1252 -*-
# Filename: lista_l.py

# Ostoslistan määrittely ja alustus
lista = ['omena', 'mango', 'porkkana', 'turtana'] #huomioi hakasulut []

print 'esine 0 on', lista[0]
print 'esine 1 on', lista[1]
print 'esine 2 on', lista[2]
print 'esine 3 on', lista[3]
print 'esine -1 on', lista[-1]
print 'esine -2 on', lista[-2]
print 'esineet 1-3 ovat', lista[1:3]
print 'esineet 2 eteenpäin ovat', lista[2:]
print 'esineet 1 -> -1 ovat', lista[1:-1]
print 'kaikki esineet yhdessä:', lista[:]
```

Tuloste

```
>>>
esine 0 on omena
esine 1 on mango
esine 2 on porkkana
esine 3 on turtana
esine -1 on turtana
esine -2 on porkkana
esineet 1-3 ovat ['mango', 'porkkana']
esineet 2 eteenpäin ovat ['porkkana', 'turtana']
esineet 1 -> -1 ovat ['mango', 'porkkana']
kaikki esineet yhdessä: ['omena', 'mango', 'porkkana', 'turtana']
>>>
```

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

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

`del` voi myös poistaa koko listan yhdellä kertaa:

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

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

Yleisimpiä listan jäsenfunktioita

PSF

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

append(*x*)
: Lisää alkio x listan loppuun.

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

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

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

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

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

count(*x*)
: Palauttaa numeroarvon i joka kertoo kuinka monta kertaa x esiintyy listalla.

sort()
: Järjestää listan alkiot arvojärjestykseen.

reverse()
: Kääntää listan alkiot ympäri, eli ensimmäinen viimeiseksi jne.

Esimerkki 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]
```

Luokka-rakenne

LTY

Kaikissa tilanteissa aiemmin esittelemämme tietorakenteet kuten lista tai tuple eivät riitä. Joskus saatamme haluta rakentaa uusia tietomuotoja, joiden alkioille on annettu oma nimi. Esimerkiksi jos haluaisimme tallentaa henkilötietoja, olisi varmaan helpoin tapa tallentaa ne tietomuotoon, joka on suunniteltu ihmisten henkilötietojen tallentamiseen?

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

Esimerkki 8.4: Luokka rakenteisena tietomuotona

```
# -*- coding: cp1252 -*-
#Tiedoston nimi: luokka.py

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

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

#Luodaan uusi ihminen nimeltään mies ja
#annetaan hänelle henkilötiedot

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

#Tulostetaan miehen tiedot

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

Tuloste

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

Kuinka se toimii

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

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

Esimerkki 8.5: Luokka rakenteisena tietomuotona, esimerkki 2

```
# -*- coding: cp1252 -*-
#Tiedoston nimi: luokka2.py

#Määritellään koira, annetaan sille
#oletusarvot

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

#Luodaan kaksi uutta koiraa

koira_1 = Koira()
koira_2 = Koira()

#Annetaan toiselle koiralle omat tiedot

koira_1.nimi = "Jäyhä"
koira_1.rotu = "Jököttäjä"
print koira_1.nimi, koira_1.rotu

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

Tuloste

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

Kuinka se toimii

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

Luokan jäsenfunktioista

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

Luokkarakenteeseen voidaan siis liittää jäsenfunktioita, joiden avulla ne voivat käsitellä saamiaan parametreja tai itsensä tietoja. Esimerkiksi aiemmin käyttämällämme `Ihminen`-rakenteella voisi jäsenfunktiona olla *syntymapaiva*, jolla jäsenmuuttujaa `ika` kasvatettaisiin yhdellä, tai *potkut*, jolloin rakenteen jäsenmuuttujan `ammatti` arvoksi asetettaisi ”työtön”. Seuraavassa esimerkissä esittelemme yksinkertaisen jäsenfunktion toteuttamistavan.

Esimerkki 8.6: Luokka-rakenteen jäsenfunktiot

```
# -*- coding: cp1252 -*-
#Tiedoston nimi: luokka.py

class Ihminen:
   etunimi = ""
   sukunimi = ""

   def sano_hei(self):
      print "Terve!"

mies = Ihminen()
mies.etunimi = "Kalevi"
mies.sukunimi = "Karvajalka"

print mies.etunimi, mies.sukunimi,"sanoo"
mies.sano_hei()
```

Tuloste

```
>>>
Kalevi Karvajalka sanoo
Terve!
>>>
```

Kuinka se toimii

Tällä kertaa rakenteen `Ihminen` on hieman tavallisuudesta poikkeava. Kuten huomaamme, on muutama tällä kertaa tarpeeton jäsenmuuttuja jätetty pois, koska emme käytä niitä tässä tehtävässä. Sen sijaan olemme esitelleet rakenteen sisällä jäsenfunktion `sano_hei`. Kyseinen jäsenfunktio määritellään normaalin funktion tavoin, mutta sen argumentteihin on lisätty avainsana `self`. Tämä avainsana liittyy olio-ohjelmointiin eikä sitä tässä vaiheessa käsitellä sen tarkemmin, nyt riittää kun tiedät, että toimiakseen Pythonin jäsenfunktion parametreihin pitää liittää parametri `self`. Jäsenfunktio tulostaa kutsuttuna ruudulle merkkijonon ”Terve!”.

Lisäksi olemme koodissa alustaneet rakenteen jäsenmuuttujat. Kun ajamme koodin, toimii jäsenfunktio aivan kuten normaali funktio: kutsun saatuaan tulkki tulostaa merkkijonon ”Terve!”. Koska jäsenfunktio on määritelty rakenteen määrittelyssä, tarkoittaisi se silloin sitä, että jokainen tämän koodin `Ihminen`-tyyppinen rakenteinen muuttuja sisältäisi jäsenfunktion `sano_hei`.

Huomioita

Kuten sanottu, jäsenfunktiot ja niiden määrittely sisältää pitkälti olio-ohjelmointiin liittyviä aiheita ja asioita, joten niistä emme puhu tässä oppaassa tämän enempää. Jos haluat lukea aiheesta lisää, on esimerkiksi Byte of Pythonin luvussa 12 sekä Python Software Foundationin tutoriaalin osassa 9 paljon lisätietoa aiheesta. Python on itse

asiassa täysverinen olio-ohjelmointikieli, vaikka tällä kurssilla sitä käytetään ensisijaisesti proseduraaliseen ohjelmointiin.

Matriisi

LTY

Pythonin omien muuttujatyyppien lisäksi voimme käyttää erikoisempia muuttujatyyppejä, jotka saamme käyttöömme asentamalla laajennusmoduuleja. Yksi tällainen muuttujatyyppi on NumPy-moduulin mukana tuleva matriisi-muuttujatyyppi.

Matriisia on helpointa luonnehtia 2-ulotteiseksi kiinteäkokoiseksi listaksi. Käytännössä Pythonissa matriisia voidaan käyttää tehokkaasti joko monimutkaisten laskutehtävien toteuttamiseen tai tiedon tallentamiseen taulukoihin. Muissa ohjelmointikielissä matriisilla on myös muita käyttötarkoituksia, kuten tietovaraston tai staattisen tallennusmuuttujan rooli. Ennen kuin voimme käyttää matriiseja, joidumme sisällyttämään numpy-moduulin tulkkiin. Tämä tapahtuu komennolla `import numpy`. Tällä erää riittää kun kirjoitat sisällytyskäskyn, seuraavassa luvussa puhumme tarkemmin siitä mitä käsky oikeastaan tekee.

Luomme matriisin komennolla

```
>>>import numpy
>>>matriisi = array(([1,2,3],[4,5,6],[7,8,9]))
```

Käskyllä määrittelemme muuttujan matriisi, joka on 3*3-kokoinen matriisi. Matriisin rivialkioille annamme arvot yhdestä yhdeksään.

```
>>> matriisi
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> print matriisi
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
```

Kun tarkastelemme luomaamme muuttujaa, voimme havaita, että matriisin sisällä oleviin alkioihin viitataan samalla tavoin kuin listan alkioihin: ensin annamme rivin ja tämän jälkeen sarakkeen numeron. Kannattaa tosin huomata, että sarakkeiden ja rivien järjestysnumerointi alkaa nollasta.

```
>>> matriisi[2][2]
9
>>> matriisi[1][1]
5
>>>
```

Matriisin alkioille voidaan myös suorittaa laskutoimituksia, kuten kertolaskuja.

```
>>> matriisi*10
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
>>>
```

Lisäksi kaksi yhteensopivaa matriisia voidaan kertoa keskenään. Myöhempää käyttöä varten numpystä löytyy myös tehokkaampia matriisilaskuoperaattoreita.

```
>>> matriisi1 = array(([1,2],[3,4]))
>>> matriisi2 = array(([5,6],[7,8]))
>>> matriisi3 = matriisi1*matriisi2
>>> matriisi3
array([[ 5, 12],
       [21, 32]])
```

Matriisin ei myöskään välttämättä tarvitse sisältää pelkästään numeroita, vaan voimme luoda matriiseja myös kirjaimilla.

```
>>> matriisi_kirjain = array((["a","b","c"],["d","e","f"]))
>>> print matriisi_kirjain
[['a' 'b' 'c']
 ['d' 'e' 'f']]
>>>
```

Kirjaimia voidaan myös vaihtaa keskenään. Tällä tavoin matriisia voidaankin helposti käyttää vaikkapa tiedontallennustaulukkona esimerkiksi shakki- tai tammipelille. Jokainen laudan ruutu edustaisi tässä tapauksessa matriisin alkiota.

```
>>> matriisi_kirjain[1][2] = "p"
>>> print matriisi_kirjain
[['a' 'b' 'c']
 ['d' 'e' 'p']]
>>>
```

Numpy-moduulin matriisien tärkein käyttö on kuitenkin tieteellisessä laskennassa. Jos haluat lukea lisää matriisien käytöstä, löytyy numpy-moduulin kotisivuilta http://numpy.scipy.org kattava käyttöopas, jossa esitellään erilaisia tapoja käyttää matriiseja. Sivuilla esitellään myös erilaisia laskumenetelmiä, joita numpy-moduuliin on sisäänrakennettu.

Komentoriviparametrit ja niiden käyttäminen

Komentoriviparametreja käytetään useimmiten helpottamaan ja nopeuttamaan ohjelmien ajamista. Käytännössä komentoriviparametrit toimivatkin siten, että voit syöttää ohjelmallesi arvoja jo ohjelman käynnistyksessä, jolloin vältyt välivaiheiden aiheuttamilta katkoksilta suorituksen yhteydessä. Komentoriviparametrien käyttämistä varten joudut kuitenkin työskentelemään komentorivikehotteessa koska IDLE:n kautta niiden syöttäminen mielekkäästi ei onnistu.

Komentoriviparametrit saat käyttöön lisäämällä lähdekoodisi päätasolle sisällytyskäskyn `import sys`. Tällä komennolla saat käyttöösi perusfunktiokirjaston lisäksi kirjastomoduulin `sys`, jonka kautta pääset käsiksi komentoriviparametreihin. Sisällyttämisestä ja kirjastomoduuleista puhumme enemmän ensi viikolla, nyt riittää kun kirjoitat tämän rivin koodiisi. Tämän jälkeen komentoriviparametrit ovat käytettävissä ohjelmassasi lista `sys.argv` alkioina. Listan ensimmäinen alkio, sys.argv[0] sisältää ajettavan tiedoston nimen ja loput alkiot annettuja komentoriviparametreja. Seuraava esimerkki koskeekin nimenomaisesti komentoriviparametreja. Huomaa, että toimiakseen seuraava esimerkki vaatii, että ympäristömuuttujat on asennettu oikein.

Esimerkki 8.8. Komentoriviparametrien käyttäminen

Lähdekoodi

```
# -*- coding: cp1252 -*-
#Tiedosto: koripa.py
import sys

print "Ajettavan koodin nimi on",sys.argv[0]
print "Annoit",len(sys.argv),"komentoriviparametria."
print "Ne olivat"
for i in sys.argv:
    print i
print "sys.argv on kuin normaali lista:"
print sys.argv
```

Tämän jälkeen ajamme ohjelman komentoriviltä valitsemalla Käynnistä-valikosta kohdan ”Suorita…”. Kirjoita tähän ikkunaan ”command” ja paina enter. Oletetaan että tallensimme tiedoston ”koripa.py” hakemistoon D:\testi\ . Siirrymme siihen kansioon ja suoritamme koodin seuraavalla käskyllä

```
D:\TESTI>python koripa.py eka toka 313 terve
```

Tuloste

```
> Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.


D:\TESTI>python koripa.py eka toka 313 terve
Ajettavan koodin nimi on koripa.py
Annoit 5 komentoriviparametria.
Ne olivat
koripa.py
eka
toka
313
terve
sys.argv on kuin normaali lista:
['koripa.py', 'eka', 'toka', '313', 'terve']

D:\TESTI>
```

Kuinka se toimii

Ensimmäinen huomionarvoinen asia on se, että tällä kertaa tulostus ohjautui komentoriville eikä esimerkiksi IDLE:n interaktiiviseen ikkunaan. Tämä siis tarkoittaa sitä, että työskentely komentoriviparametrien kanssa tapahtuu kokonaisuudessaan ”vanhalla tavalla” komentorivikehotteessa. Mikäli et ole koskaan tutustunut tähän ympäristöön, voit aloittaa siihen tutustumisen vaikkapa etsimällä Internetistä Lisa Lemieuxin tutoriaalin hakusanalla ”DOS tutorial”.

Katsotaan ensimmäisenä, mitä me itse asiassa kirjoitimme riville. Kuten varmaan huomasit, lähdekoodissa lukee nyt aiemmin mainittu `import sys`. Tämä mahdollistaa sen, että yleensä voimme tehdä mitään komentoriviparametreilla. Kun syötimme komentoriville käskyn ”`python koripa.py eka toka 313 terve`” niin itse asiassa ensin kutsuimme Python-tulkkia käskyllä `python` ja annoimme sille ajettavaksi tiedoston `koripa.py`, joka sijaitsi samassa hakemistossa kuin missä jo olimme (D:\TESTI\). Tämän jälkeen annoimme tulkille neljä parametria välitettäväksi ajettavalle koodille; nämä siis olivat ”`eka`”, ”`toka`”, ”`313`” sekä ”`terve`”.

Itse lähdekoodi taas pääsi käsiksi näihin parametreihin, koska tulkki tallensi ne listaan `sys.argv` muotoon `['koripa.py', 'eka', 'toka', '313', 'terve']`. Nyt lähdekoodi voi operoida arvoilla normaalien listaoperaatioiden kautta. Ainoa asia, mikä tekee `sys.argv`-listasta hieman tavallisuudesta poikkeavan onkin se, että tietue [0] ei sisällä ensimmäistä parametriä vaan ajetun tiedoston nimen. Parametrit alkavat tietueesta [1] ja jatkuvat niin kauan kuin niitä on annettu. Tämän vuoksi lista `sys.argv` onkin aina vähintään yhden alkion pituinen, koska lähdekooditiedoston nimi tallennetaan joka tapauksessa paikkaan [0] annettiin parametreja tai ei.

Muita Python-kielen rakenteita

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

Tuple

BoP

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

Tuplejen käyttäminen

Esimerkki 8.3: Tuplen käyttäminen

```
# -*- coding: cp1252 -*-
# 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 funktio len() zoo:sta, saamme arvon 3, jolloin voimme siis päätellä että myös tuple on sarjamuuttuja ja sitä voidaan käyttää for-lauseen ehtona. 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 numero ja tämän jälkeen jäsenalkion numero, eli vaikkapa uusi_zoo`[2][2]`, joka siis on ”pingviini”.

Huomioita tuplesta

Kuten aiemmin 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 keskenään. Listan määrittelyssä käytetään hakasulkuja [], kun taas tuplen yhteydessä käytetään ’tavallisia’ kaarisulkuja (). Muuten niiden määrittelyn syntaksi on täysin identtinen:

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

Sanakirja (eng. Dictionary)

BoP

Sanakirja on hieman poikkeava sarjamuuttuja verrattuna kahteen muuhun malliin, listaan ja tupleen. Sen toimintalogiikka muistuttaa enemmän puhelinluetteloa – tai yllättäen sanakirjaa - kuin kauppalistaa. Sanakirja sisältää kaksi päätyyppiä, **avaimet** (key) ja **arvot** (value). Sanakirjan tapauksessa päätyypeistä avaimen tulee olla ainutlaatuinen, eli sanakirja ei voi sisältää kahta samaa avainta.

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

Sanakirjan määrittely on kuitenkin suhteellisen samanlainen muihin sarjamuuttujiin verrattuna:

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

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

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

Sanakirjan käyttäminen

Esimerkki 8.7. Sanakirjan käyttäminen

```
# -*- coding: cp1252 -*-
# Filename: sanakirja.py


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


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

print "167-671 sähköposti on", sanakirja['167-671']

#Lisätään tietue
sanakirja['617-716'] = 'baggy@beagle.biz'

# Poistetaan tietue
del sanakirja['176-167']

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

for avain, arvo in sanakirja.items():
        print 'Veljeksen %s sähköposti on %s' % (avain, arvo)

if '176-761' in sanakirja: # tai sanakirja.has_key('176-761')
        print "\n176-761:n osoite on %s" % sanakirja['176-761']
```

Tuloste

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

Sanakirjassa on 4 merkintää.

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

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

Kuinka se toimii

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

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

Kehittyneempiä tapoja käyttää listaa

Listaa voidaan käyttää perustoimintojensa lisäksi myös simuloimaan tietotekniikan algoritmeissa tavallisia tietovarastotyyppejä kuten kekoa tai pinoa. Koska listalle voidaan lisätä alkioita mihin tahansa väliin ja listasta voidaan ottaa alkioita mistä tahansa välistä, on lista jäsenfunktioineen varsin tehokas rakenteinen tietomuoto verrattuna mihin tahansa dynaamiseen tietomuotoon millä tahansa ohjelmointikielellä.

Vaikka näistä kehittyneemmistä tavoista käyttää listaa ei välttämättä olekaan perusasioiden opettelussa paljoakaan hyötyä, voit palata tähän kappaleeseen jos joskus haluat opetella enemmän listan käyttömuotoja.

Listan käyttäminen pinona (LIFO)

Listametodit tekevät listan käyttämisestä pinona varsin helppoa. Pinohan on tunnetusti joukko, jossa uusin alkio menee päällimmäiseksi ja jossa ulos tulee pinon päällimmäinen alkio. Pinon päälle voidaan lisätä alkioita metodilla append(), samoin kuin pinosta voidaan ottaa alkio metodilla pop(). Käyttämällä molempia ilman erillisiä argumentteja saamme suoraan pinorakenteen:

```
>>> pino = [3, 4, 5]
>>> pino.append(6)
>>> pino
[3, 4, 5, 6]
>>> pino.pop()
6
>>> pino
[3, 4, 5]
>>> pino.pop()
5
>>> pino
[3, 4]
```

Listan käyttäminen jonona (FIFO)

Listan käyttäminen jonona ei ole sekään kovin vaikea toteuttaa. Jonon toteutus pinosta eroaa siten, että jonoon tullaan viimeiseksi ja jonosta poistuu pisimpään jonottanut (eli siis jonon ensimmäinen alkio). Tämä voidaan toteuttaa vaikkapa pop-metodin argumentilla 0, joka siis poistaa alkion ensimmäisen tietueen:

```
>>> jono = ["Eric", "John", "Michael"]
>>> jono.append("Terry")           # Terry lisätään jonoon
>>> jono.append("Graham")          # Graham lisätään jonoon
>>> jono.pop(0)
'Eric'
>>> jono.pop(0)
'John'
>>> jono
['Michael', 'Terry', 'Graham']
```

Huomioita sarjallisten muuttujien vertailusta PSF

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

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

Alkioita keskenään verrattaessa se alkio, jonka ensimmäinen eroava merkki on ASCII-taulukon arvoltaan suurempi, on myös lopullisesti suurempi, vaikka alkio olisi lyhyempi. Tämän takia esimerkiksi isot kirjaimet ovat aina pieniä kirjaimia arvollisesti pienempiä. Alla olevasta taulukosta näet joitain esimerkkejä näiden sääntöjen käytännön tulkinnoista:

```
(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)
```

Huomaa myös, että erilaisten tietotyyppien vertaileminen keskenään on sallittua. Tässäkin kohtaa on olemassa johdonmukainen, mutta hieman keinotekoinen nyrkkisääntö: tietuetyyppien englanninkieliset nimet päättävät niiden keskinäisen vertailun arvon. Tämän vuoksi lista (list) on aina pienempi kuin merkkijono (string), joka puolestaan on aina pienempi kuin tuple (tuple) jne. Tämä ei kuitenkaan päde numeroarvoihin: kokonaisluku (integer) 3 on samanarvoinen kuin liukuluku (float) 3.00, eli numeroita verrataan keskenään nimenomaisesti numeroarvoina tallennustyypistä huolimatta.

Luku 9: Kirjastot ja moduulit

Alkuperäislähde Byte of Python, luku 8, sekä Python Software Foundation Tutorial luvut 10.6 ja 10.7

Moduulit

BoP

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

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

sys-moduulin käyttäminen

Esimerkki 9.1 sys-moduuli

```
# -*- coding: cp1252 -*-
# Tiedosto: kaytto_sys.py

import sys

print 'Komentoriviparametrit olivat:'
for parametri in sys.argv:
        print parametri

print '\n\nJärjestelmähakemisto PYTHONPATH on', sys.path, '\n'
```

Tuloste

```
> Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.


D:\TESTI>python kaytto_sys.py nama on parametrit
Komentoriviparametrit olivat:
kaytto_sys.py
nama
on
parametrit


Järjestelmähakemisto PYTHONPATH on ['',
'C:\\WINDOWS\\system32\\python24.zip', 'Z:\\', 'C:\\Python24\\DLLs',
'C:\\Python24\\lib', 'C:\\Python24\\lib\\plat-win',
'C:\\Python24\\lib\\lib-tk', 'C:\\Python24', 'C:\\Python24\\lib\\site-
packages', 'C:\\Python24\\lib\\site-packages\\win32',
'C:\\Python24\\lib\\site-packages\\win32\\lib',
'C:\\Python24\\lib\\site-packages\\Pythonwin']

D:\TESTI>
```

Kuinka se toimii

Ensiksi sisällytämme `sys`-moduulin käyttämällä `import` -käskyä. Käytännössä tämä ainoastaan ilmoittaa tulkille, että haluamme käyttää tämän nimistä moduulia. Sys-moduuli sisältääkin erityisesti Python-tulkkiin ja sen toimintaympäristöön liittyviä toimintoja. Tähän viittaa kirjaston nimikin sillä sys on lyhenne sanasta system, eli järjestelmä.

Kun tulkki lukee import-käskyn, se etsii tiedostoa nimeltä “moduulinnimi.py” – eli tässä tapauksessa `sys.py` – kansioista, jotka on määritelty listaan, joka löytyy nimellä `sys.path`. Mikäli kyseinen tiedosto löytyy, tulkki lataa ja alustaa kyseisen tiedoston päätason komennot, jolloin moduulin funktiot ovat käytettävissä myös myöhemmin.

Komentoriviparametrien tallentamiseen käytettävää listamuuttujaa `argv` kutsutaan `sys`–moduulista käyttäen pistenotaatiota - `sys.argv` – jolloin voimme olla varmoja, että emme vahingossa ylikirjoita paikallisen koodin `argv`-muuttujaa. Tähän on olemassa muutamia poikkeuksia riippuen sisällyttämismenetelmästä, mutta niistä puhumme hieman myöhemmin. Yleisesti notaatiotapa kuitenkin vahvistaa sitä käsitystä, että käytämme nimenomaisesti `sys`-moduulin muuttujaa `argv`.

`sys.path` siis sisältää tiedon siitä, mitkä hakemistot Python ymmärtää kotihakemistoikseen. Python myös osaa sisällyttää ja hakea tiedostoja automaattisesti näistä kansioista. Huomioi lisäksi, että listan ensimmäinen tietue on tyhjä; tämä siis tarkoittaa sitä, että oletushakemistojen lisäksi myös se hakemisto, johon

lähdekooditiedosto on tallennettu kelpaa sisällytettävän tiedoston sijainniksi. Muutoin joudut siirtämään tiedostot johonkin kansioista, jotka on mainittu `sys.path` listalla.

Esikäännetyt .pyc-tiedostot

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

from...import -sisällytyskäsky

Joissain tapauksissa saatamme törmätä tilanteeseen, jossa haluamme saada funktiot ja muuttujat suoraan käytettäviksemme ilman pistenotaatiota. Eli siis siten, että pääsisimme esimerkiksi komentoriviparametreihin käsiksi ilman `sys`-etuliitettä käyttämällä pelkkää muuttujannimeä `argv`.

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 `sys`-moduulin osat suoraan käyttöömme, voisimme toteuttaa sen käskyllä ”`from sys import *`”, ”`*`” import-osan jälkeen tarkoittaa että haluamme tuoda kaikki funktiot ja muuttujat käyttöömme. Toisaalta jos haluaisimme tyytyä pelkkiin komentoriviparametreihin, niin voisimme tehdä sen käskyllä `from sys import argv`

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

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

Palauttaisivat tulosteen

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

Kun taas sisällytystapa

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

Palauttaisivat tulosteen

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

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

Omien moduulien tekeminen ja käyttäminen

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

Oman moduulin luominen

Esimerkki 9.2 Oma tiedosto moduulina

```
# -*- coding: cp1252 -*-
#Tiedosto munmoduli.py

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

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

versio = 1.0
sana = "ketsuppijäätelö"
```

Yllä oleva lähdekooditiedosto toimii esimerkin moduulina. Se on tallennettu alla olevan ajettavan koodin kanssa samaan kansioon ja se sisältää kaksi funktiota, `terve` ja `summaa`. Lisäksi moduulille on määritelty vakiomuuttujat versio ja sana, joista ensimmäinen kertoo munmoduli-moduulin versionumeron ja toinen sisältää vain satunnaisen merkkijonon. Huomionarvoisena seikkana kannattaa muistaa, että moduuli toimisi myös jostain sys.path-listalle määritellystä kansiosta. Seuraavaksi teemme ohjelman jolla kokeilemme moduulin toimintaa:

```
# -*- coding: cp1252 -*-
#Tiedosto munmoduli_ajo.py

import munmoduli

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

print "munmodulin versionumero on",munmoduli.versio
print "Päivän erikoinen taas on:",munmoduli.sana
```

Tuloste

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

Kuinka se toimii

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

Yhteenveto

Nyt olemme katsoneet jonkin verran sisällyttämistä, joten voimme siirtyä eteenpäin tutustumaan varsinaisiin kirjastomoduuleihin. Jatkossa tulemme tutustumaan lähes viikoittain uusiin kirjastomoduuleihin, joten on tärkeää, että tästä luvusta opit ainakin valmiiden moduulien sisällyttämisen lähdekoodiisi.

Kirjastomoduuleja

PSF

math-moduuli

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

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

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

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

random-moduuli

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

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   #Satunnainen joukko
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]

>>> random.random()    # Satunnainen liukuluku
0.17970987693706186
>>> random.randrange(6)    # satunnainen kokonaisluku; välinä range(6)
4
```

urllib2-moduuli

urllib2 on moduulina varsin erikoinen. Se ei ole pelkästään koneensisäinen moduuli, vaan itse asiassa mahdollistaa internet-protokollien kautta tiedon hakemisen Internetistä. Tämä tarkoittaa siis sitä, että moduuli osaa hakea verkkoresurssien, kuten www-sivujen tai Internetin uutissyötteiden tietoja omien funktioidensa avulla. Yksinkertaisimmillaan tämä voidaan toteuttaa seuraavalla komennolla:

```
import urllib2
for rivi in urllib2.urlopen('http://www.it.lut.fi'):
    print rivi
```

Tämä tulostaa tietojenkäsittelyn laitoksen verkkosivujen etusivun html-lähdekoodin interaktiiviseen ikkunaan; tuloste olisi yli 10-sivuinen joten tavallisuudesta poiketen jätämme sen tässä näyttämättä. Jos toistolauseen lukemat rivit olisi tallennettu listaan olisimme myös samalla saaneet talteen sivujen lähdekoodin.

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

datetime-moduuli

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

```
>>># luodaan päiväys
>>> import datetime
>>> nyt = datetime.date.today()
>>> nyt
datetime.date(2006, 8, 15)
>>> nyt.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'08-15-06. 15 Aug 2006 is a Tuesday on the 15 day of August.'
>>># päiviä voi vähentää toisistaan kuten lukuja
>>> syntymapaiva = datetime.date(1934, 3, 13)
>>> ika = nyt - syntymapaiva
>>> ika.days
26453
>>>
```

time-moduuli

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

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

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

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

Unix-järjestelmissä ajanotto toimii hieman eri tavalla, mutta antaa samankaltaisen vastauksen. Unixissa aikaa ei oteta mistään järjestelmärajapinnan apufunktiosta, vaan se otetaan suoraan prosessin käyttämältä prosessoriajalta. Määritelmä ”prosessoriaika” on jo itsessäänkin hieman epäselvä ajanottomenetelmä, mutta tätä funktiota voi käyttää laskenta-aikojen optimoimiseen kummalla tahansa alustalla. Funktiota käyttäessä täytyy vain muistaa, että saadut ajat ovat suhteellisia, eikä mitattu tulos ole vertailukelpoinen kuin samassa ajoympäristössä samalla työasemalla.

Time-moduuli sisältää myös paljon muita toimintoja, jotka eivät kuitenkaan ole tätä kurssia ajatellen kovinkaan relevantteja. Niistä voit lukea tarkemmin Python Software Foundationin referenssikirjastosta.

Esittelemme lisää kirjastoja myöhemmillä viikoilla sitä mukaa kun niitä tarvitsemme. Jos taas haluat välittömästi tutustua uusiin moduuleihin ja katsella millaisia moduuleja on olemassa, löytyy niistä kuvaukset Python Software Foundationin referenssikirjastosta, jonka lyhyt käyttöohje löytyy liitteestä 2.

Luku 10: Poikkeusten käsittelyä

Alkuperäislähde Byte of Python, luku 13

Johdanto

BoP

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

Virheistä yleisesti

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

```
>>> Print 'Moi maailma!'
    File "<stdin>", line 1
      Print 'Moi maailma!'
                          ^
SyntaxError: invalid syntax

>>> print ‘Moi maailma!’
Moi maailma!
```

Huomaa, että tulkin antama virhekoodi on tarkkaan ottaen `SyntaxError`, ja että tulkki itse asiassa myös osoittaa pienen nuolen avulla sen, missä kohtaa virhe tapahtui. Tulkki siis tietää mitä tapahtui ja missä, mutta ei osannut tehdä virheelle mitään muuta kuin antaa huomautuksen ja lopettaa. Entä jos jatkossa rakentaisimmekin tulkille toipumissuunnitelman, joka kertoisi sille mitä tehdään jos virhe tapahtuu?

try…except-rakenne

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

```
>>> luku = input("Anna luku: ")
Anna luku: hyppyrotta

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in -toplevel-
    luku = input("Anna luku: ")
  File "<string>", line 0, in -toplevel-
NameError: name 'hyppyrotta' is not defined
>>>
```

Python palauttaa virheen nimeltä NameError, joka käytännössä tarkoittaa sitä, että tulkki luuli tässä tapauksessa merkkijonoa “hyppyrotta” muuttujan nimeksi jota ei ollut olemassa. 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 NameError, voimme rakentaa `try-except` –rakenteen. Käytännössä tämä tapahtuu siten, että laitamme try-osioon sen koodin, jonka haluaisimme tavallisesti ajaa, ja except-osioon sen koodin, joka ajetaan mikäli virhe tapahtuu.

Esimerkki 10.1 Virheen käsitteleminen

```
# -*- coding: cp1252 -*-
#Tiedosto kaato.py

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

Tuloste

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

Kuinka se toimii

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

```
>>>
Anna luku:

Traceback (most recent call last):
  File "C:/Python24/bah.py", line 5, in -toplevel-
    luku = input("Anna luku: ")
KeyboardInterrupt
>>>
```

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

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

Saisimme tulostukseksi

```
>>>
Anna luku:
Keskeytit ajon.
>>>
```

Toisaalta taas muuttamalla koodi muotoon

```
# -*- coding: cp1252 -*-
#Tiedosto kaato2.py

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

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

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

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

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

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

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

Esimerkki 10.2 Else-rakenne

```
# -*- coding: cp1252 -*-
#Tiedosto kaato4.py

try:
    luku = input("Anna luku: ")
    print "Annoit luvun", luku
except Except:
    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

BoP

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

Finally-osio käytännössä

Esimerkki 10.3 Finally-osio

```
# -*- coding: cp1252 -*-
#Tiedosto finally.py

import time

try:
        tiedosto = file('uutinen.txt')
        while True:
                rivi = tiedosto.readline()
                if len(rivi) == 0:
                        break
                time.sleep(2)
                print rivi,
finally:
        tiedosto.close()
        print 'Puhdistetaan jäljet, suljetaan tiedosto'
```

Tuloste

```
>>>
Balin palapelitehdas
pisti pillit pussiin pian
pelipalojen palattua piloille pelattuina:
Puhdistetaan jäljet, suljetaan tiedosto

Traceback (most recent call last):
  File "C:/Python24/bah.py", line 12, in -toplevel-
    time.sleep(2)
KeyboardInterrupt
>>>
```

Kuinka se toimii

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

Huomionarvoista on kuitenkin se, että juuri ennen virheen esiintymistä suoritamme finally-osion ja vapautamme käyttämämme tiedoston. Finally-osio onkin tehty juuri tätä varten - jos tapahtuu virhe, voi ohjelma vielä viimeisinä tehtävinään vapauttaa käyttämänsä tiedostot ja verkkoresurssit, jotta käyttöjärjestelmä voi antaa ne eteenpäin toisille ohjelmille.

Luku 11: Algoritmista koodiksi

Pseudokoodin tai algoritmin muuttaminen toimivaksi koodiksi

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

Pseudokielet ovat ainutlaatuisia siksi, että niiltä puuttuu kokonaan kielioppi. Ne eivät varsinaisesti ole humaanisia kirjakieliä siinä missä esimerkiksi englanti, saksa tai suomi saatikka sitten teknisiä ohjelmointikieliä kuten Python tai C. 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 - normaalisti säiliöitä ja lippuja – yksinkertaistamaan esityksen kokonaisuutta.

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

Pseudokoodiesitys lisäyslajittelusta

```
def Lisäyslajittelu(lista t)

   n = listan t alkioiden lukumäärä

   // Käydään taulukon jokainen  (paitsi ensimmäinen) alkio läpi

   for i = 2 to n //siirrytään toisesta alkiosta kohti loppua, alkuosa
                  pysyy lajiteltuna
       j = i

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

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

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

         end while
   end for
```

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

```
def lisays
    for
        while
```

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

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

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

```
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, sekä tapa tallentaa tiedot, joten lisäämme sen vielä loppuun ensin ehdot:

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

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

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

Huomaamme, että olemme luoneet Pythonilla pseudokoodista lisäyslajittelufunktion.

Huomautus

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

Luku 12: Bittiarvot ja merkkitaulukot, merkkioperaatiot

Alkuperäinen lähde Python Foundation Reference Library, luku 2.3.6.1

Bittiarvot

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

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

Esimerkki 12.1: Bittilukujen laskeminen, binaariluvusta kokonaisluvuksi

```
# -*- coding: cp1252 -*-
#Binääriluvusta kokonaisluvuksi

def bittiluku():
    bittijono = raw_input("Anna binaariluku: ")

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

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

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

    print "Bittijonosi on kokonaislukuna",tulos

bittiluku()
```

Tuloste

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

Kuinka se toimii

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

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 1, 2, 4, 8 ja 16. 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, laskettaisi se seuraavasti:

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

Jos vastaavasti laskisimme bittiarvon esitykselle 101101, saisimme seuraavanlaisen tuloksen:

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

Toisaalta, voimme myös laskea bittiesityksen kokonaisluvuille seuraavanlaisella ohjelmalla:

Esimerkki 12.2: Bittilukujen laskeminen, kokonaisluvusta binaariluvuksi

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

def laske_binaari(luku):
    potenssi = 0
    while True:  #Laskee kuinka monta bittiä esitykseen tarvitaan
        if 2**potenssi <= luku:
            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 -=1    #Lähestytään arvoa 0 joka kierroksella
        if potenssi == -1:   #Ollaan tultu luvun loppuun
            break
    return jono

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

Tuloste

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

Kuinka se toimii

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

Merkkitaulukot

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

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

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

Yläpuolella olevaan taulukkoon on kerätty ASCII-taulukosta kirjaimia ja numeroarvoja, sekä joitain erikoismerkkejä vastaavat numeroarvot. Jos luemme levyltä bittisarjan 1000001, tarkoittaa se kokonaislukuina arvoa 65. Jos taas tulkitsemme tämän numeroarvon ASCII-taulukon avulla, voimme havaita että luimme tiedostosta ison A-kirjaimen. Jos tiedostossa vastaavasti lukisi ”01011100 01110101 01010100 01101111”, voitaisi se lukea arvoina ”97 117 84 111”, eli ”auTo”.

Meitä ajatellen ASCII-taulukossa on kuitenkin yksi suuri ongelma; se ei tunne skandinaavisia merkkejä. Alkuperäinen ASCII-taulukko suunniteltiin nimenomaisesti englanninkieliselle aakkostolle, joten alkuperäinen ratkaisu ei sisältänyt Ä, Ö eikä Å -kirjaimia. Tämän vuoksi olemme myöhemmin joutuneet laajentamaan olemassa olevia

merkkitaulukkoja sekä luomaan joukon uusia. Lisäksi käyttöjärjestelmiin on toteutettu mahdollisuus valita mitä merkkitaulukkoa ohjelmien lukemisessa ja tulkitsemisessa käytetään.

Tähän liittyy myös Python-lähdekoodeissa oleva ensimmäinen rivi

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

Tällä rivillä määrittelemme, että käytämme lähdekooditiedoston tulkitsemiseen merkkitaulukkoa 1252, joka Windows XP-käyttöjärjestelmässä tarkoittaa eurooppalaista tai skandinaavista näppäinkarttaa, joka sisältää ruotsin, suomen, norjan, tanskan, ranskan ja saksan kielissä tarvittavat lisämerkit. Vastaavasti näppäinkartta 1257 sisältää baltian maissa käytetyt erikoismerkit, ja 1251 kyrilliset aakkoset. Alkukirjaimet `cp` tulevat käyttöjärjestelmätermistä *code page*. 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'
>>>
```

Huomionarvoista on, että jokaisella Windows XP:n merkkitaulukolla 128 ensimmäistä merkkiä ovat samat kuin ASCII-taulukossa ja erikoismerkit on tallennettu järjestysnumeroille 128-255. Tämä mahdollistaakin sen, että englanninkieliset merkit ja järjestelmämerkit – ensimmäiset 32 merkkiä – pysyvät aina samoina vaikka merkkitaulukkoa vaihdettaisiin kesken käytön.

Modernissa tietotekniikassa ollaan siirtymässä kohti universaaleja merkkitaulukoita, joista esimerkkinä mainittakoon ASCII-yhteensopiva UTF-8. Tässä merkkitaulukossa jokaista merkkiä kohti käytetään 4*8 bittiä, eli merkkitaulukko kattaa 2^{32} (4 294 967 296) merkkiä. Tällä merkkimäärällä voidaan latinalaisten, kyrillisten ja arabialaisten aakkosten lisäksi kuvata niin Kiinan kuin Japanin tai Koreankin sanamerkit, jolloin tarve erillisille merkkitaulukoille poistuu pysyvästi. UTF-8 on kuitenkin vielä suhteellisen uusi merkkitaulukkomuoto, eikä sitä vielä tueta kaikissa käyttöjärjestelmissä tai ohjelmissa. Python-ohjelmointikielessä UTF-8-tuki kuitenkin jo löytyy, ja se voitaisi ottaa käyttöön antamalla määrittelyriviksi

```
# -*- coding: UTF8 -*-
```

Ongelmaksi kuitenkin muodostuu itse Windows, joka ei tunnista UTF-8-merkistöä oikein, joten tulkin tulostukset erikoismerkeillä voivat olla pahasti pielessä. Tämän vuoksi onkin suositeltavaa käyttää merkkitaulukkoa 1252.

Merkkijonojen metodit

PSF

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

`capitalize`()
: Palauttaa merkkijonosta kopion, jossa ensimmäinen merkki on muutettu isoksi kirjaimeksi.

`endswith`(*testi*[, *start*[, *end*]])
: Palauttaa arvon `True` jos merkkijono päättyy muotoon *testi*, muutoin palauttaa arvon `False`. Lisäkytkimet *start* ja *end*, joilla voidaan määrätä testattava alue.

`startswith`(*prefix*[, *start*[, *end*]])
: Sama kun endswith() mutta testaa merkkijonon alkua.

`expandtabs`([*tabsize*])
: Palauttaa merkkijonosta kopion, jossa kaikki sisennysmerkit on korvattu tabsize –määrällä välilyöntejä. Tabsize on oletusarvoisesti 8. (Huom! IDLEllä 4)

`find`(*sub*[, *start*[, *end*]])
: Palauttaa ensimmäisen merkin sijainnin, jos annettu merkkijono sub löytyy testijonosta. Lisäkytkimet start ja end mahdollistavat hakualueen rajaamisen. Palauttaa arvon -1 jos arvoa ei löydy.

`index`(*sub*[, *start*[, *end*]])
: Kuin `find()`, mutta palauttaa virheen `ValueError` jos merkkijonoa sub ei löydy.

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

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

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

islower()
: Palautttaa arvon True jos kaikki merkkijonon merkit ovat pieniä kirjaimia ja merkkijonossa on ainakin yksi kirjain. Muutoin False.

isupper()
: Sama kuin islower() mutta isoille kirjaimille.

isspace()
: Palautttaa arvon True jos kaikki merkkijonon merkit ovat välilyöntejä ja merkkijonossa on ainakin yksi merkki. Muutoin False

lower()
: Palauttaa merkkijonon kopion, jossa kaikki kirjaimet on muutettu pieniksi kirjaimiksi.

lstrip([*chars*])
: Palauttaa merkkijonon, josta on vasemmasta reunasta poistettu kaikki määritellyt merkit. Jos poistettavia merkkejä ei määritellä, poistettaan pelkästään välilyönnit ja sisennykset. Tasaus on aina vasemmassa kulmassa, ensimmäinen ei-poistettava merkki määrää tasauksen paikan:

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

: Oikean reunan tasaamiseen käytetään vastaavaa funktiota nimeltä 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 parametric count, joka määrää kuinka monta korvausta maksimissaan tehdään.

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

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

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

Jos erotinmerkkiä ei anneta, käytetään oletuserottimena välilyöntiä.

upper()

Muuttaa kaikki merkkijonon kirjaimet isoiksi kirjaimiksi.

Luku 13: Graafisten käyttöliittymien alkeet

Teksti julkaistu alun perin LTY:n oppaassa ”Python-Tkinter ja Graafinen käyttöliittymä”, luku 1. Opas kokonaisuudessaan saatavilla osoitteesta http://wiki.python.org/moin/Languages/Finnish

Perusteet

Toisin kuin vielä esimerkiksi 15 vuotta sitten, Nykyisin useimmat ohjelmat julkaistaan käyttöjärjestelmille, joissa on mahdollisuus käyttää graafista käyttöliittymää. Tämän vuoksi käytännössä kaikki laajemmat ohjelmat, ohjelmistot sekä työkalut toimivat nimenomaan graafisella käyttöliittymällä, jossa valinnat ja vaihtoehdot esitetään valikoina tai valintoina, joiden avulla käyttäjä voi hiiren avulla valita mitä haluaa tehdä. Monesti aloitteleva ohjelmoija kuitenkin luulee, että tällaisen käyttöliittymän toteuttamista varten tarvitaan jokin monimutkainen tai kallis kehitystyökalu tai että se olisi erityisen vaikeaa. Ehkä joidenkin ohjelmointikielien yhteydessä tämä pitää paikkansa, mutta Pythonissa yksinkertaisten graafisten käyttöliittymien tekeminen onnistuu helposti Tkinter-kirjastomoduulin avulla.

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

Seuraavilla kahdella esimerkillä tutustumme siihen, kuinka yksinkertaisen graafisen ikkunan tekeminen onnistuu. Tämä tietenkin tarkoittaa, että ensimmäiseksi tarvitsemme pohjan, johon käyttöliittymän rakennamme.

Graafinen käyttöliittymä

Esimerkki 13.1: Perusikkuna

Esimerkkikoodi

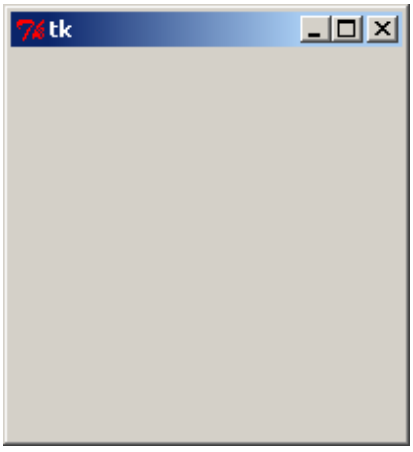
```
# -*- coding: cp1252 -*-

from Tkinter import *

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

Esimerkkikoodin tuottama tulos

Ajamme esimerkin koodin, tuottaa tulkki seuraavanlaisen käyttöliittymäikkunan:



Kuva 1: Esimerkin 1.1 tuottama käyttöliittymäikkuna.

Kuinka koodi toimii

Ensimmäinen esimerkkikoodi on varsin lyhyt eikä vielä sisällä paljoakaan toimintoja. Ensimmäisellä rivillä meillä on aikaisemmistakin oppaista tuttu näppäimistökartan määrittely, jolla saamme käyttöömme skandinaaviset merkit sekä muut laajennetun ANSI-standardin mukaiset merkit. Toisella rivillä otamme käyttöön Tkinter-kirjaston. Tällä kertaa kannattaa huomata, että sisällytys toteutetaan hieman tavallisuudesta poikkeavalla from...import * -syntaksilla johtuen siitä että sen käyttäminen on tämän kirjaston yhteydessä näppärämpää.

Kolmannella rivillä luomme käyttöliittymän perustan tekemällä muuttujasta `pohja` Tk()-luokan juuri- eli pohjatason (*root*). Tähän pohjatasoon lisäämme jatkossa kaikki haluamamme komponentit ja toiminnot, tällä kertaa kuitenkin riittää, että jätämme sen tyhjäksi. Neljännellä rivillä käynnistämme Tkinter-käyttöliittymän kutsumalla pohjatasoa

sen käynnistysfunktiolla `mainloop`. Ohjelma lähtee käyntiin ja tuottaa tyhjän ikkunan. Tämä tapahtuu sen vuoksi, että ohjelman pohjatasolle ei ole sijoitettu mitään muuta komponenttia. Jos olisimme laittaneet sille vaikka painonapin, olisi ruutuun ilmestynyt pelkkä painonappi.

Komponenttien lisääminen

Käyttöliittymästä ei ole kuitenkaan 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. `Frame`-komponentista puhumme lisää myöhemmin, tarkastelkaamme ensin kuinka pohjatasolle lisätään vaikkapa tekstikenttä.

Esimerkki 13.2: Tekstikenttä perusikkunassa

Esimerkkikoodi

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

from Tkinter import *

pohja = Tk()

tekstikentta = Label(pohja, text="Moi maailma!")
tekstikentta.pack()

pohja.mainloop()
```

Esimerkkikoodin tuottama tulos

Ajamme esimerkin koodin, tuottaa tulkki seuraavanlaisen käyttöliittymäikkunan:

Kuva 2: Esimerkin 1-2 ikkuna.

Kuinka koodi toimii

Tässä lähdekoodissa tuotamme ensimmäisen käyttöliittymän, joka tekee jotain. Alkuun suoritamme samanlaiset alustustoimenpiteet kuten aiemmin. Määrittelemme näppäimistökartan, otamme käyttöön Tkinter-moduulin sekä luomme pohjatason muuttujaan `pohja`. Tästä eteenpäin määrittelemme ensimmäisen komponenttimme.

Tällä kertaa luomme komponentin nimeltä `Label`, jota käytetään tavallisimmin staattisen tekstin esittämiseen. Tämä sopiikin meille varsin hyvin, koska haluamme ainoastaan saada ohjelman tuottamaan tekstiä aiemmin tekemäämme ikkunaan. Tässä tapauksessa luomme komponentin muuttujaan `tekstikentta` ja annamme sille määrittelyssä seuraavat parametrit:

1) Ensimmäinen parametri `pohja` tarkoittaa sitä, että luomme komponentin pohjatason päälle. Tämä tarkoittaa sitä, että käynnistäessämme ohjelman, ilmestyy luomamme tekstikenttä tekstikentta pohjatason määrittelemän alueen sisään. Tähän annetaan tavallisesti parametrina se ikkunan osa tai alue, johon napin halutaan ilmestyvän, mutta koska meillä ei ole käytössä olevaa ikkunointiasettelua niin komponentti voidaan laittaa suoraan pohjalle.
2) Toinen parametri määrittelee yksinkertaisesti sen, mitä tekstikentässä on tarkoitus lukea. Luonnollisesti parametriksi voi antaa myös muuttujannimen, mutta ilman lisätoimenpiteitä teksti luetaan muuttujasta ainoastaan alustuksen yhteydessä. Tekstikentän päivittämisestä puhutaan myöhemmin lisää.

Seuraavaksi paketoimme kyseisen komponentin. Tällä käskyllä ilmoitamme tulkille, että emme anna kyseiselle komponentille enempää alustusmääritteitä ja että komponentin voi ”pakata” käyttöliittymään. Pakkauksen yhteydessä voimme vielä antaa joitain sijoittelua koskevia käskyjä, mutta tällä kertaa niitä ei tarvita. Muista, että **ainoastaan pakattu komponentti näkyy käyttöliittymän ikkunassa**. Jos komponenttia ei pakata, ei tulkki koskaan piirrä sitä ruudulle ja silloin luultavasti käyttöliittymäsi toimii väärin. Lopuksi laitamme ohjelman käyntiin kutsumalla pohjatason `mainloop`-funktiota.

Huomioita Tkinter-moduulista

Koska Tkinter-moduuli on hyvin laaja ja toimintatavoiltaan jonkin verran tavallisesta Python-ohjelmakoodista poikkeava, ei tässä oppaassa aihetta käsitellä tämän enempää. Lisää graafisten käyttöliittymien toteuttamisesta Tkinter-moduulilla voit lukea tämän oppaan jatko-osasta ”Python – Tkinter ja graafinen käyttöliittymä”. Opas löytyy mm. verkosta osoitteesta

http://wiki.python.org/moin/Languages/Finnish

Loppusanat

Huomioita

Olemme tämän oppaan turvin tutustuneet 13 luvussa Python-ohjelmoinnin alkeisiin sekä joihinkin ohjelmistokehityksen perusajatuksiin. Tähän pisteeseen päästyämme voimme jo alkaa puhumaan ohjelmistotuotannosta sekä varsinaisesta ohjelmoinnista tavoitteenamme tehdä toimivia, niin sanotusti ”oikeita” ohjelmia sen sijaan, että rakentelemme esimerkkejä olemassa olevien rakenteiden päälle. Tästä eteenpäin loogisia jatkoaiheita ovatkin laitteistoläheisemmät ohjelmointikielet sekä toisaalta graafisten käyttöliittymien toteuttaminen.

Python on siitä hyvä ohjelmointikieli, että voit halutessasi jatkaa vielä kauan kielen kanssa harjoittelua. Se, että ymmärsit perusasiat, on jo merkki siitä, että saat halutessasi tehtyä paljon muitakin asioita kuin mitä tässä oppaassa käsiteltiin. Vaikka et tietäisi tarvitsevasi ohjelmointitaitoja jatkossa, sinun ei kannata täysin unohtaa nyt oppimiasi asioita: Nykytietotekniikalla ohjelmointi on aihe, joka tulee vastaan hyvinkin yllättävissä paikoissa, kuten Excel-taulukoiden tai interaktiivisten PowerPoint-esitysten yhteydessä. Lisäksi, mikäli olet tietotekniikan opiskelija, on sinun hyvä opetella Pythonia, koska se on vahvassa kasvussa oleva nuori kieli, jonka käyttäjäkunnasta löytyy jo tässä vaiheessa mm. Nokia ja NASA.

Lisäluettavaa

Tästä oppaasta eteenpäin jatkavalle on olemassa useita vaihtoehtoja. Jos haluat tutustua tarkemmin esimerkiksi kuvien muokkaamiseen ja piirtämiseen Python-ohjelmointikielellä, on oppaalle julkaistu jatko-osa ”Python - Graafinen ohjelmointi Imaging Librarylla”. Jos taas haluat opetella tekemään Windows-ohjelmien kaltaisia graafisia käyttöliittymiä, kannattaa sinun tutustua oppaaseen ”Python - Tkinter ja graafinen käyttöliittymä”. Molemmat oppaat ovat täysin suomenkielisiä ja ennen kaikkea ilmaisia.

Kaikki yllämainitut oppaat, sekä muutama liitteeksi tai lisäluettavaksi tarkoitettu miniopas on saatavilla Python Software Foundationin verkkokirjastosta osoitteesta

http://wiki.python.org/moin/Languages/Finnish

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/.

Liite 1: Valmistelut

IDLE ja Python 2.5.1 asennusohje

- Ensimmäinen vaihe Python-ympäristön asennuksessa on asennuspaketin hakeminen osoitteesta http://www.python.org/download/releases/2.5.1/

- Valitse sivulta omaan käyttöjärjestelmääsi sopiva asennusversio. Mikäli työasemassasi on 64-bittinen prosessori ja sitä tukeva käyttöjärjestelmän versio, voit myös ladata niitä varten optimoidun version. Esimerkiksi jos olet Windows-käyttäjä, valitse tiedosto, jonka nimi on "python-2.5.1.msi". Klikkaa tiedostonnimeä aloittaaksesi lataaminen.

- Nyt ruudulle ilmestyy selaimen tiedostoikkuna, josta valitse "Save to disk" tai "Save as...", tai selaimesi vastaava toiminto. Tallenna tiedosto haluamaasi paikkaan, kuten esimerkiksi työpöydälle.

- Kun tiedosto on latautunut, tarkasta että saamasi tiedosto oli kokonainen. Windowsin Python 2.5.1 -paketin kooksi Windows ilmoittaa 10714 KB. Ylläolevan linkin alareunasta löytyy myös MD5-tarkistussumma, mutta jos et tiedä mitä ne ovat, riittää kunhan varmistat että pakettisi koko on samaa luokkaa annetun ilmoituksen kanssa.

Työvaiheet, Windows-asennus

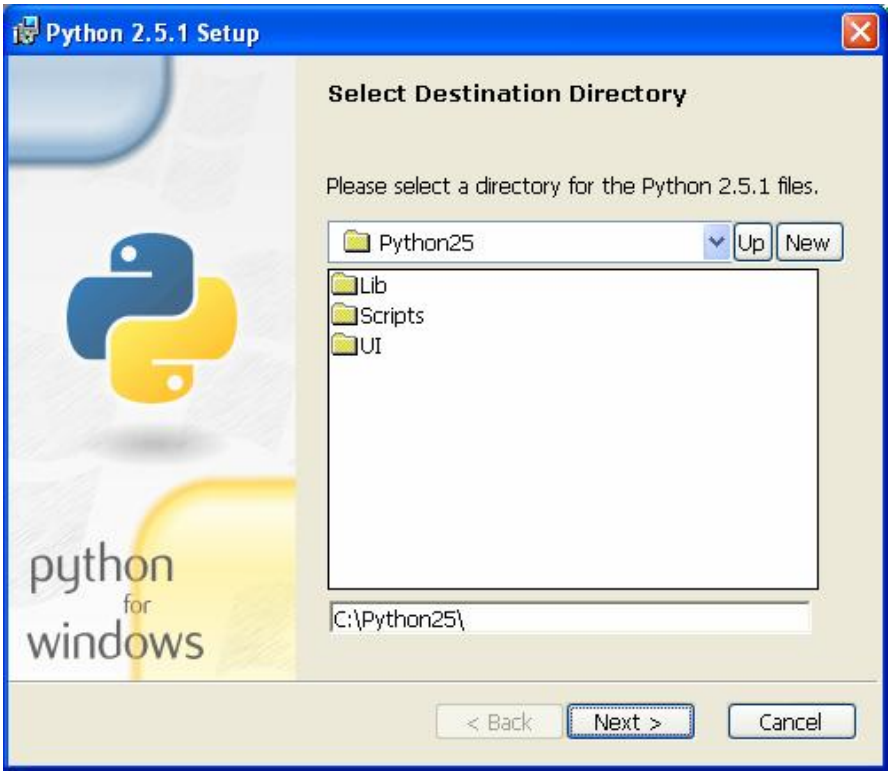
Tässä vaiheessa sinulla tulisi olla koneella valmiiksi latautunut versio Python -kehitysympäristöstä. Seuraavaksi asennamme itse ohjelman, joten varmista, että sinulla on riittävät oikeuden asentaa käyttämääsi koneeseen ohjelmia.
Jos teet töitä kotikoneelta käsin, niin tilanne luultavasti onkin näin, mutta muussa tapauksessa vaihda koneellasi tunnuksille, joilla voit tehdä asennuksia ja muuttaa asetuksia.

- Klikkaa tiedostoa python-2.5.1.msi.
- Asennus lataa hetken, ja tämän jälkeen itse asennusohjelman pitäisi käynnistyä. Mikäli saat virheilmoituksen "System Administrator has set policies to prevent this installation", ei sinulla ole riittäviä oikeuksia ohjelman asentamista varten, jolloin joudut ottamaan yhteyttä järjestelmänvalvojaasi.
  Jos pakettisi oli ehjä ja oikealle käyttöjärjestelmälle valittu, niin seuraavanlainen ikkuna pitäisi aueta ruudulle:

- Mikäli ikkunasi ei ole tämän näköinen, siirry kohtaan "vianselvitys", joka on tämän ohjeen lopussa.

- Tarkasta, että asennettava Python on varmasti versionumero 2.5.1 eikä esimerkiksi 2.4 tai 2.3. Tällä kurssilla käytetään ainoastaan versiota 2.5.1, eikä muita versioita tueta. Versionumero on esim. ikkunan otsikossa.

- Mikäli haluat, että kaikki koneen tunnukset voivat käyttää Pythonia, valitse "Install for all users", muussa tapauksessa valitse "Install just for me". Mikäli et ole varma miten haluat toimia, pidä oletusvalinta "Install for all users" ja klikkaa painiketta "Next >" jolloin tämän näköisen ikkunan tulisi aueta:

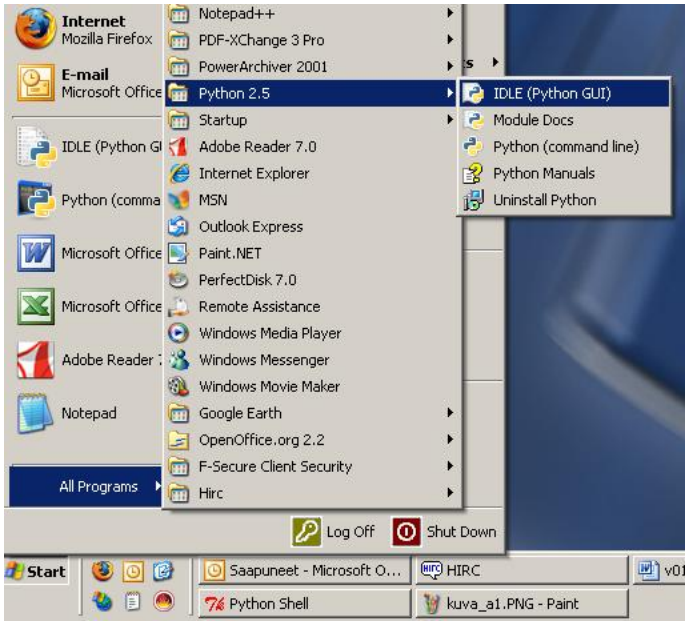


- Tässä ikkunassa valitset paikan, mihin Python asennetaan. Oletuskansio on "c:\Python25\". Laita paperille ylös kansio, johon aiot Pythonin asentaa, tätä tietoa saatetaan tarvita myöhemmin. Kun olet valinnut mieleisesi paikan, klikkaa painiketta "Next >".

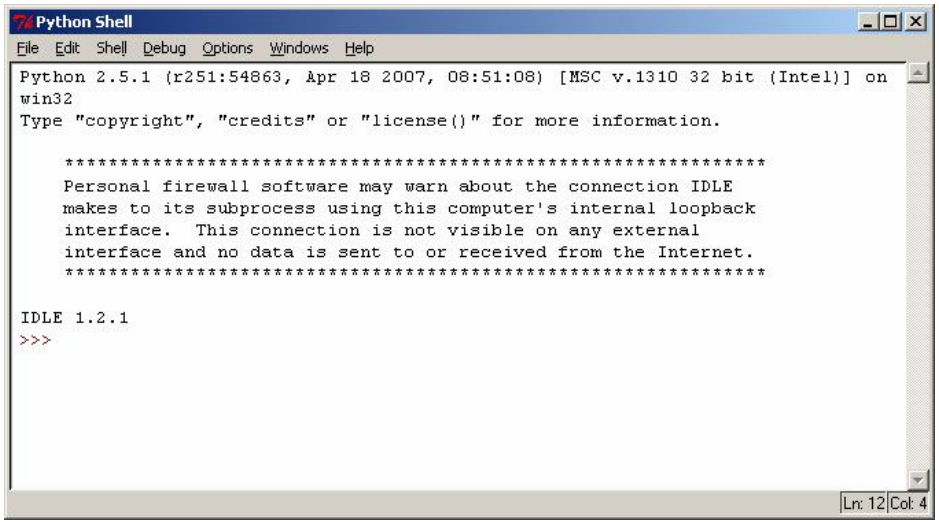
- Tässä ikkunassa voit valita, mitä osia haluat Pythonista asentaa. **Jos et tiedä mitä teet, älä muuta näitä asetuksia!** Kurssia varten on hyvä valita ja asentaa kaikki komponentit, ja näin asia oletusvalinnoilla onkin. Valitse lopuksi 'Next >'

- Tämän ikkunan tulisi ilmestyä näytölle itsestään, ja siitä voit seurata ohjelman valmistumista. Älä koske mihinkään vaan odota että seuraava ikkuna tulee näkyviin:

- Mikäli yllä oleva ikkuna on näkyvissä, on asennus onnistunut. Klikkaa "Finish". Seuraavaksi vielä etsitään ja käynnistetään ohjelma sekä varmistetaan, että kaikki toimi oikein.
- Avaa "Käynnistä"-valikko ja etsi sieltä kansio Python 2.5, josta valitse IDLE (Python GUI).



- Ohjelma käynnistyy ja mikäli seuraava ikkuna ilmestyy, on asennus onnistunut ja Python toimii koneellasi.



- Tarkasta vielä kerran tässä vaiheessa, että ohjelmasi versionumerot täsmäävät esimerkin kanssa. Eli siis itse Python on oltava versio 2.5 (Ensimmäinen rivi) ja IDLE 1.2 tai uudempi (alin rivi).
- Jos sait ilmoitukset "Subprocesses did not connect", estää koneesi palomuuri tai virustutka tai vastaava käytönvalvontaohjelma aliprosessien käynnistämisen ja tällöin joudut muuttamaan niiden asetuksia siten, että prosessi 'pythonw.exe' saa täydet paikalliset (local) toimintavaltuudet. Ohjelman prosessit ovat koneensisäisiä ja liittyvät käyttöympäristöön. Mitään tietoa ei lähetetä koskaan ulkoverkkoon, ellei käyttäjän käsin ajama koodi näin tee. Jos et tiedä miten tämä tehdään, konsultoi järjestelmänvalvojaasi.

Mikäli kuitenkin kaikki toimii, on Python nyt asennettu koneellesi ja voit aloittaa tehtävien tekemisen.

Virhetilanteet

1) Käynnistäessäni asennuksen saan seuraavanlaisen ikkunan:

- Windowsin mielestä koneellesi on jo asennettu Python-ympäristö. Jos tiedät, että kyseessä on virhe tai epäonnistunut aiempi asennus, poista se Ohjauspaneelin "Lisää/Poista Ohjelma" -valikon kautta ja uudelleenkäynnistä asennus.
- Muussa tapauksessa voit koittaa korjaus-asentaa Pythonin. Korjaaminen läpikäy asennettavat perustiedostot ja palauttaa ne alkuperäiseen asennuksenjälkeiseen tilaansa. Se ei kuitenkaan tuhoa sinun aiemmin tekemiä koodejasi.

2) Saan asennuksen aikana seuraavan virheikkunan:

Tämä ikkuna on merkki siitä, että asennus keskeytyi saamatta toimintoaan valmiiksi. Tähän voi olla monta syytä, mutta luultavimmin se on jokin seuraavista:

- Painoit epähuomiossa Cancel asennuksen aikana.
- Sinulla ei ole riittäviä oikeuksia suorittaa asennusta, mutta oikeutesi riittävät asennusohjelman käynnistämiseen.
- Yritit asentaa Pythonin levylle, joka joko oli täysi, täyttyi asennuksen aikana tai johon sinulla ei ole kirjoituslupaa.
- Asennuspakettisi sisältää virheellisiä tiedostoja.

Kokeile uudelleenasennusta. Mikäli virhe toistuu, hae paketti uudelleen verkosta. Mikäli virhe toistuu myös uudella paketilla, kokeile asennusta perusasetuksilla. Mikäli uuden paketin perusasetusasennus epäonnistuu, kannattaa ottaa yhteys järjestelmänvalvojaan tai tekniseen tukihenkilöön.

Python-tulkin laajennusmoduulien asennus

Mikä on laajennusmoduuli?

- Laajennusmoduuli on Python-tulkin käyttämä moduuli, jota ei toimiteta normaalin Python-kehitysympäristön mukana.
- Laajennusmoduuleilla lisätään tulkkiin uusia toiminnallisuuksia, kuten kuvankäsittelyfunktioita tai tieteellisen laskennan algoritmeja.
- Laajennusmoduulit haetaan tavallisesti erikseen verkosta ja asennetaan käsin.

Tarkoituksenamme on asentaa kolme laajennusmoduulia Python-kehitysympäristöön jatkokäyttöä varten, ja ne ovat

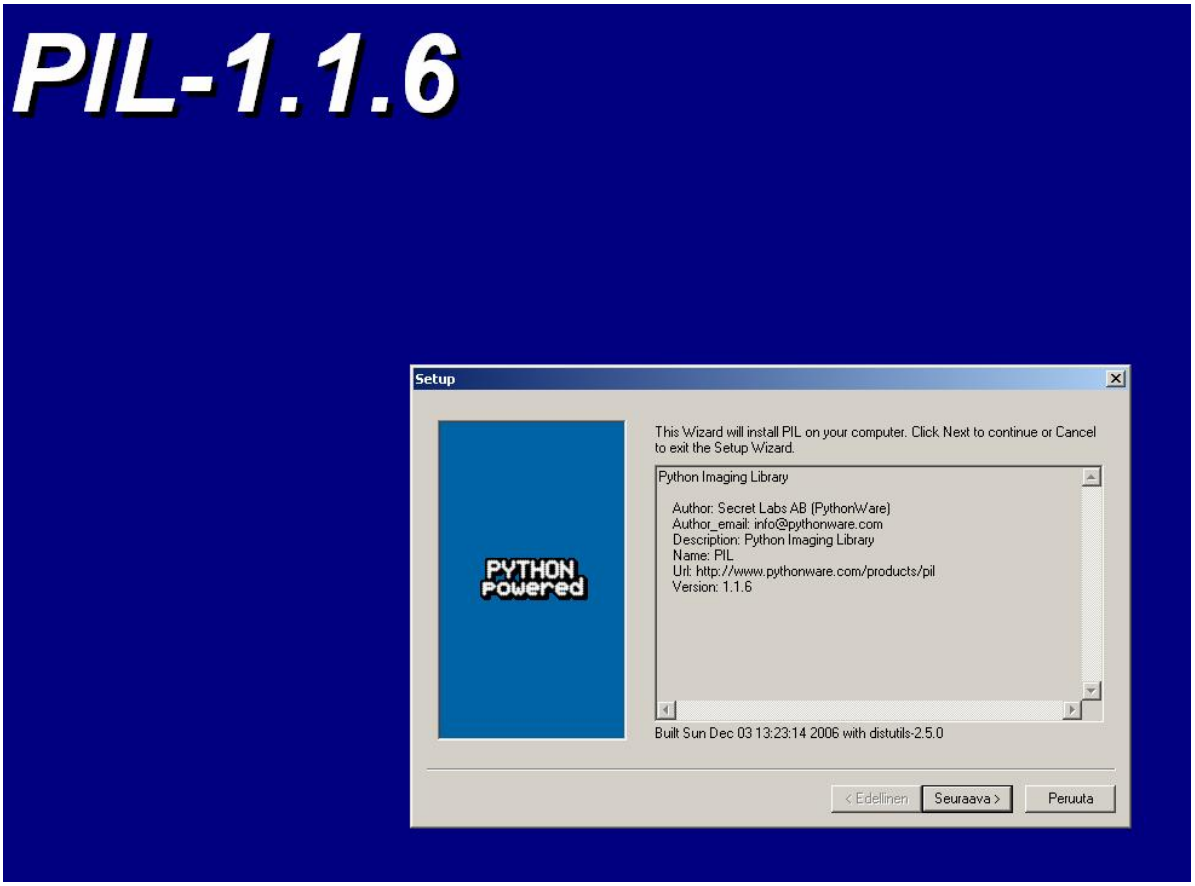
- NumPy – Tieteelliseen laskentaan käytettäviä funktioita ja matriisiesityksen sisältävä laahjennusmoduuli. Latausosoite http://numpy.scipy.org
- Python Imaging Library – Kuvankäsittelyfunktioita sisältävä laajennusmoduuli. Latausosoite http://www.pythonware.com/products/pil/
- py2exe – Työkalu itsenäisen Python-ohjelman toteuttamiseen. Latausosoite http://www.py2exe.org/

Huomaa, että kaikista kolmesta laajennusmoduulista on olemassa eri ohjelmointiympäristöihin tarkoitettu versio. Jos asensit Python-tulkista version 2.5.1, niin huolehdi siitä että latasit laajennusmoduulin versiolle 2.5 tarkoitetun paketin.

Kaikki kolme pakettia asennetaan identtisillä asennusvaiheilla. Seuraavalta sivulta alkava ohje neuvoo, kuinka asennamme tällaisen paketin Python-tulkkiin. Käytämme esimerkissä Python Imaging Library-moduulia:

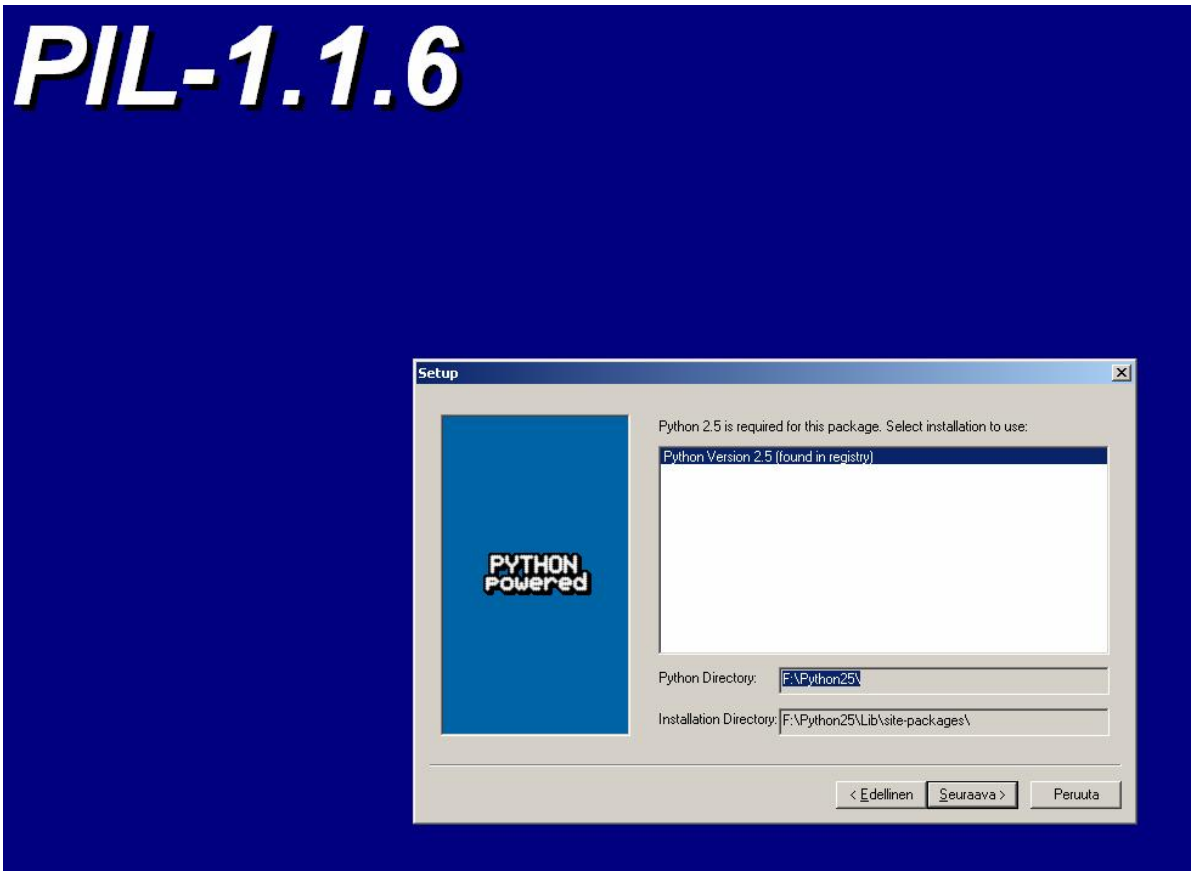
Laajennusmoduulin asennus työvaiheittain

- Kun paketti on latautunut työasemallesi, kaksoisklikkaa sitä. Paketti purkaa hetken aikaa tiedostoja. Kun tämä on valmis, ruudun pitäisi muuttua tällaiseksi:



- Tässä vaiheessa et voi vielä vaikuttaa asennuksen etenemiseen, joten riittää kun painat ”Seuraava >” jatkaaksesi asennusta.

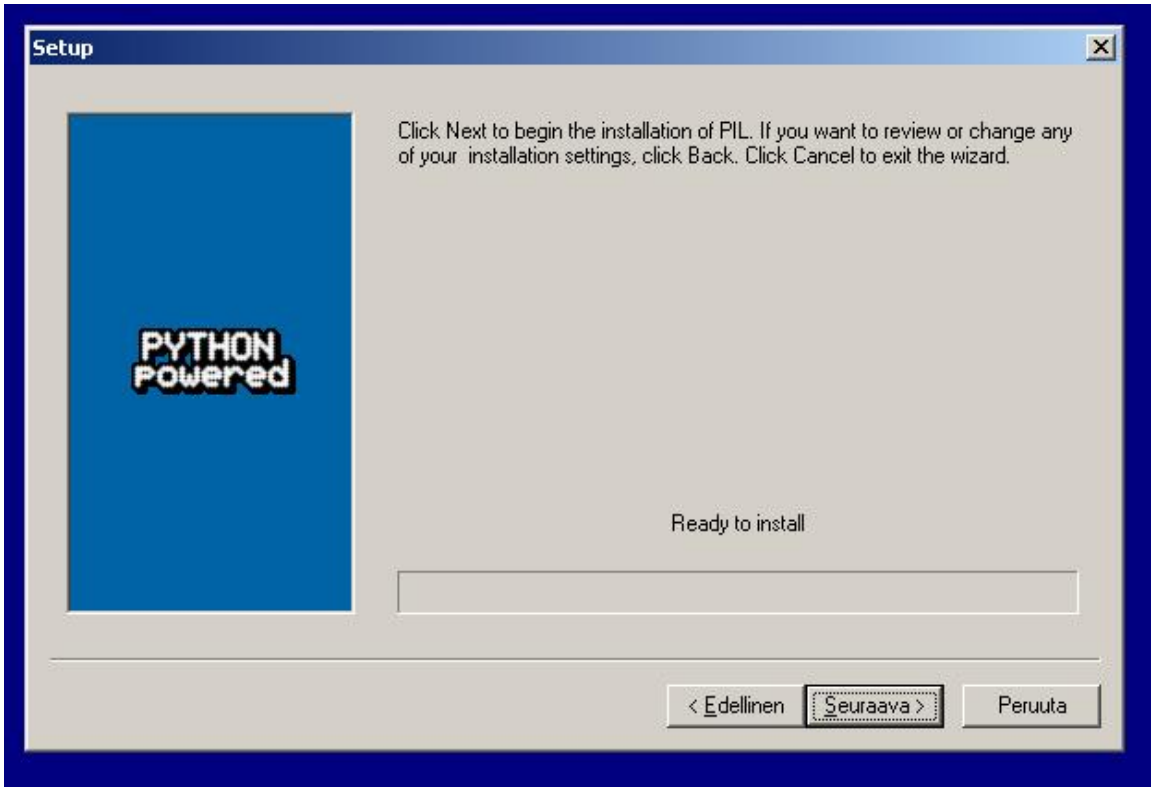
- Tämän jälkeen ruudun pitäisi muuttua tällaiseksi:



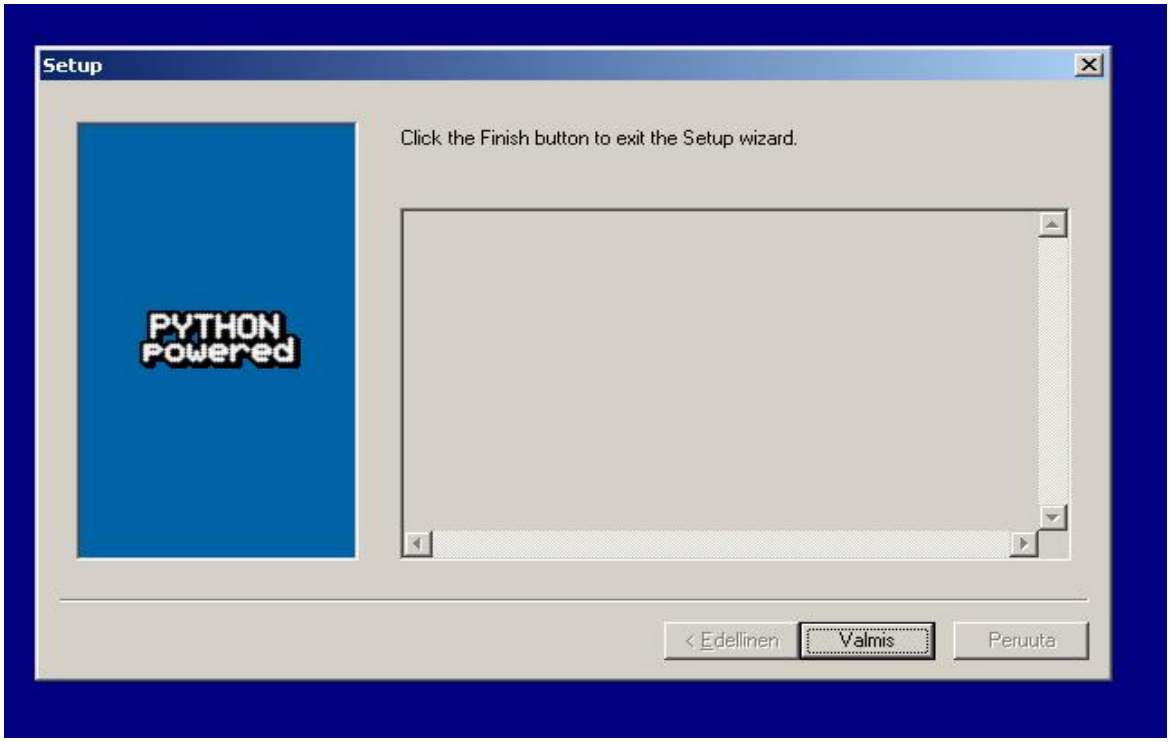
- Mikäli teit kaiken oikein Python-tulkin asennuksessa, on asennusohjelma löytänyt Python 2.5:n koneen rekisteritiedoista. Jos asennuskansio ja muut ovat automaattisesti valittuja, paina painiketta ”Seuraava > ”.
- Mikäli ohjelma ei löytänyt Python-tulkkiasi rekisteritiedoista, on luultavaa että jokin meni kehitysympäristön asennuksessa pieleen. Voit toki myös antaa käsin kansion, johon tulkin asensit, mutta on todennäköistä, että sinulla tulee olemaan jatkossa ongelmia ohjelman kanssa. Suosittelemmekin vahvasti, että asennat Python-kehitysympäristön uudelleen ja yrität saada automaattisen tunnistamisen toimimaan tällä tavoin. Jos uudelleenasennus ei auta, tarkasta seuraavat asiat:
  1. Muistithan tarkastaa, että yrität asentaa oikeaa laajennusmoduulin versiota?
  2. Joskus palomuuri-, virusturva- tai käytönvalvontaohjelmat estävät rekisteritietoihin kirjoittamisen ohjelman asennuksen yhteydessä. Tarkasta ettei omasi tee näin. Jos pystyt tekemään niin, sammuta ohjelmat asennuksen ajaksi. **Tätä vaihetta ennen on kuitenkin ehdottomasti katkaistava koneen verkkoyhteys virusturvan varmistamiseksi.** Konsultoi tarvittaessa mikrotukihenkilöäsi.
  3. Mikäli käytät yleistä työasemaa, voi olla että käyttäjätunnuksesi luokitus ei salli uusien ohjelmien asentamista, vaikka asennusohjelman käynnistäminen

onnistuisikin. Mikäli epäilet, että tilanne on näin, ota yhteyttä mikrotukihenkilöösi.

- Seuraavassa vaiheessa asennusikkunan pitäisi näyttää tältä:

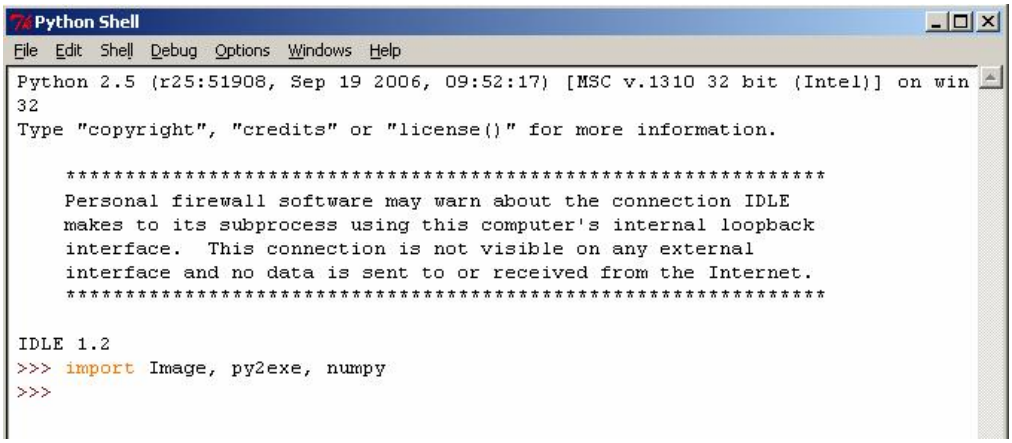


- Tässäkään vaiheessa et voi juuri vaikuttaa asennukseen, joten klikkaa valintaa ”Seuraava>”.
- Nyt ruudun tilapalkin tulisi siirtyä ruudun reunasta toiseen samalla, kun ohjelma kopioi tiedostoja ja tekee asetukset laajennusmoduulin käyttöä varten. Lopulta ikkunan tulisi näyttää tältä:



- Valitsemasi laajennusmoduuli on nyt asennettu työasemallesi. Lopeta asennusohjelma klikkaamalla valintaa “Valmis”.

Seuraavaksi valitse seuraava asennettavista moduuleista ja toista nämä työvaiheet kaikille kolmelle moduulille. Lopulta sinulla pitäisi olla kaikki kolme laajennusmoduulia asennettuna. Tämän jälkeen käynnistä Python-tulkki (IDLE) ja kirjoita tulkin aktiiviseen ikkunaan käsky ”`import Image, py2exe, numpy`” (kts. alla). Mikäli tulkki ei antanut virheilmoitusta, olet asentanut kaikki kolme laajennusmoduulia oikein.



Windows XP tiedostopäätteiden esiin saaminen

Mikä on tiedostopääte?

- Tiedostonpääte on tunniste, josta käyttöjärjestelmä tietää mikä tiedosto on tyypiltään. Esimerkiksi Microsoft Officen Word-asiakirjojen pääte on .doc. Tämä siis tarkoittaa, että tekemäsi asiakirja "Mun dokkari" on tallennettu tiedostoon nimeltä "Mun dokkari.doc". Samoin Powerpoint käyttää päätettä .ppt, sekä OpenOffice päätettä .odt; Pythonin tiedostopääte on nimeltään .py.

- Windows-käyttöjärjestelmillä on ikävä tapa oletusarvoisesti piilottaa tiedostonpäätteet, jotka se tuntee, mikä aiheuttaa sen, että näet ainoastaan osan "Mun dokkari", kun selaat kiintolevysi sisältöä. Lisäksi tämä aiheuttaa erityisen ikävän ongelman, koska et voi tietää varmasti mikä tiedostonpäätteesi on, jos windows piilottelee niitä satunnaisesti. Usein päädytäänkin tilanteeseen, jossa tiedostonnimeksi tulee epähuomiossa "Mun dokkari.doc.doc".

- Ohjelmoidessa Pythonilla tämä on erityisen ikävää, koska IDLE ei suorita apukorostuksia kuin vain ja ainoastaan .py-tyyppisille tiedostoille. Tällöin siis tiedostoa "Mun_koodi.py.txt" ei käsitellä lähdekoodina, mitä se tosiasiallisesti olisi.

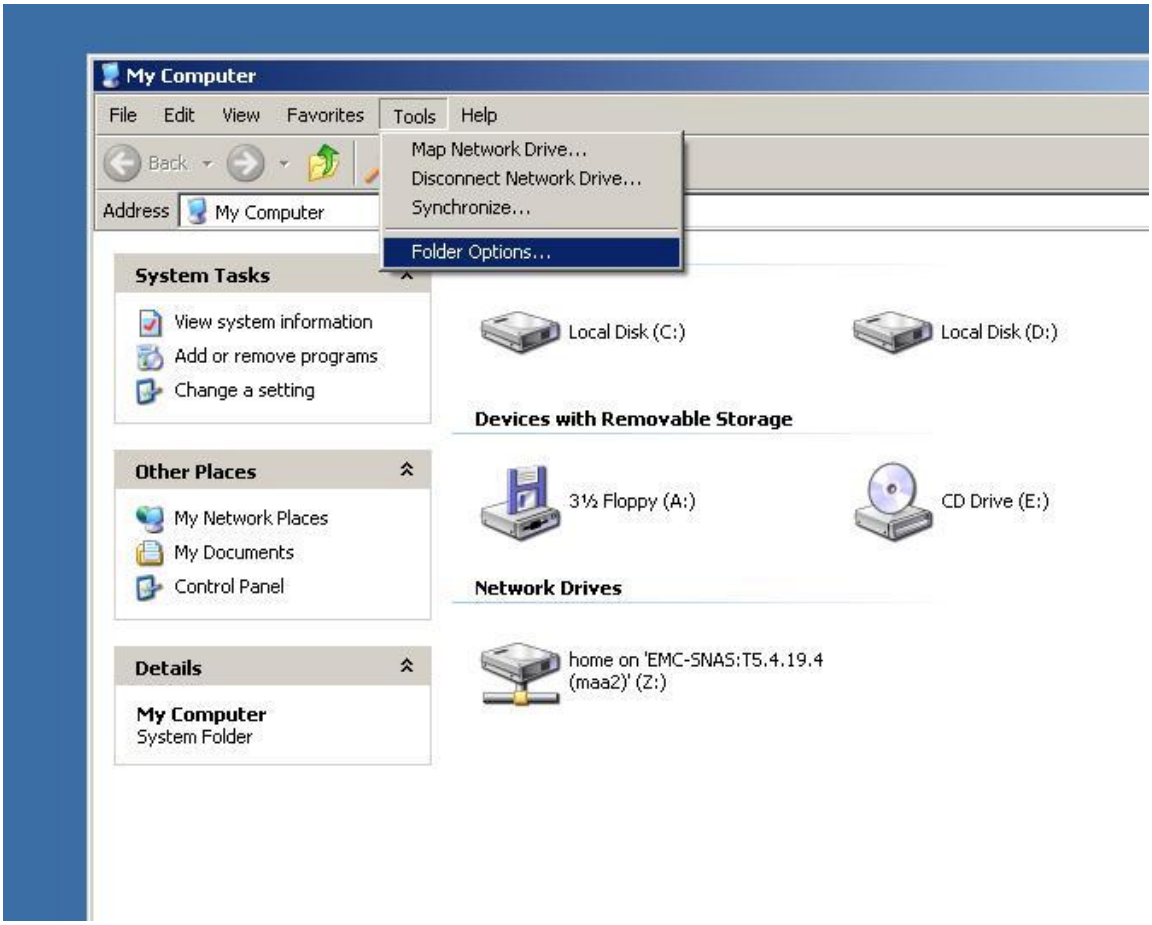
- Siksi tämä tutoriaali läpikäykin vaiheet, jotka joudut tekemään saadaksesi Windows XP-järjestelmässä esiin tiedostonpäätteet, vaikka tiedosto olisikin tunnettu.

TYÖVAIHEET, WINDOWS XP Home ja Professional

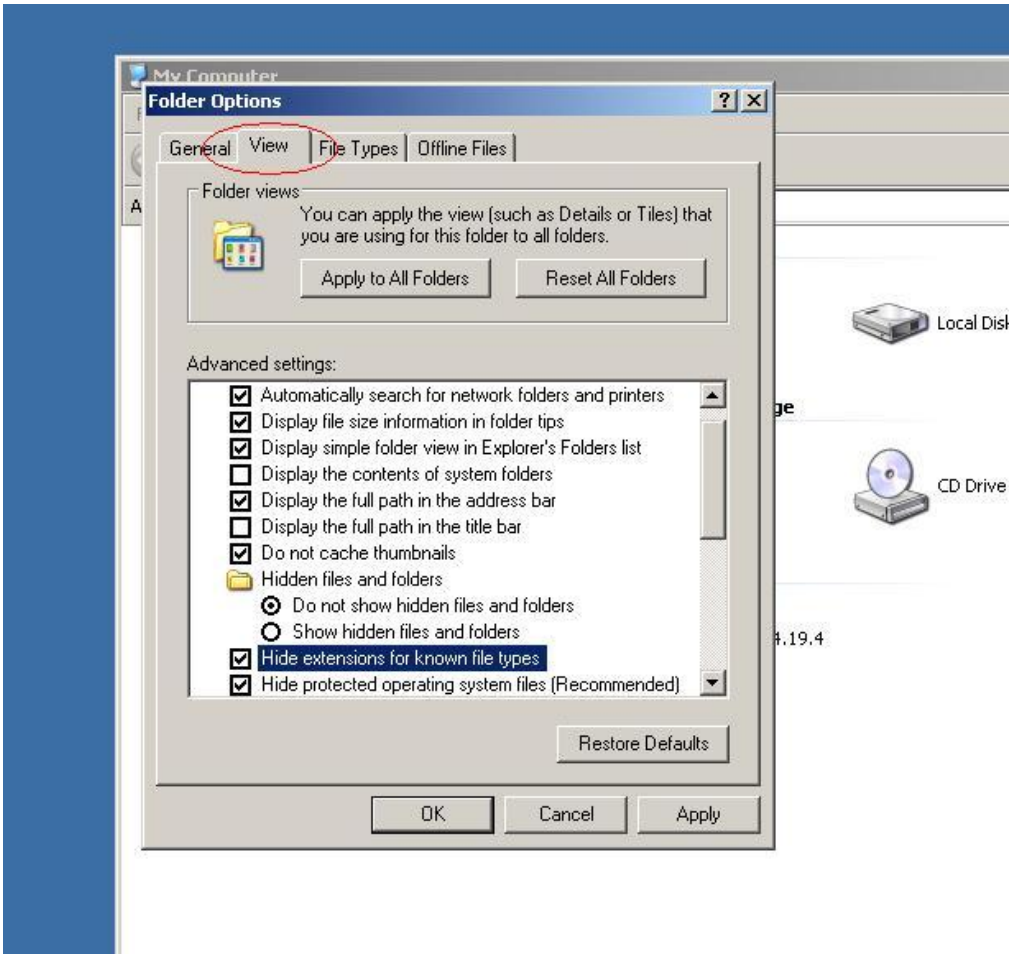
Tässä vaiheessa sinulla tulisi olla koneella asennettu versio Python -kehitysympäristöstä. Seuraavaksi suoritamme tarvittavat muutokset, jotta pääset tarkastelemaan lähdekooditiedostoja.

- Avaa Käynnistä-valikko, ja valitse sieltä 'Oma tietokone'. Kuvassa oikea valinta on korostettu sinisellä. Ohjekuvat ovat englanninkielisestä versiosta, joten seuraa niitä mikäli et halua tai voi käyttää suomenkielistä Windowsia.

- Avaa alasvetovalikosta "Työkalut" alin valinta, "Kansion asetukset". Jälleen kerran oikea valinta on kuvassa korostettu sinisellä palkilla.

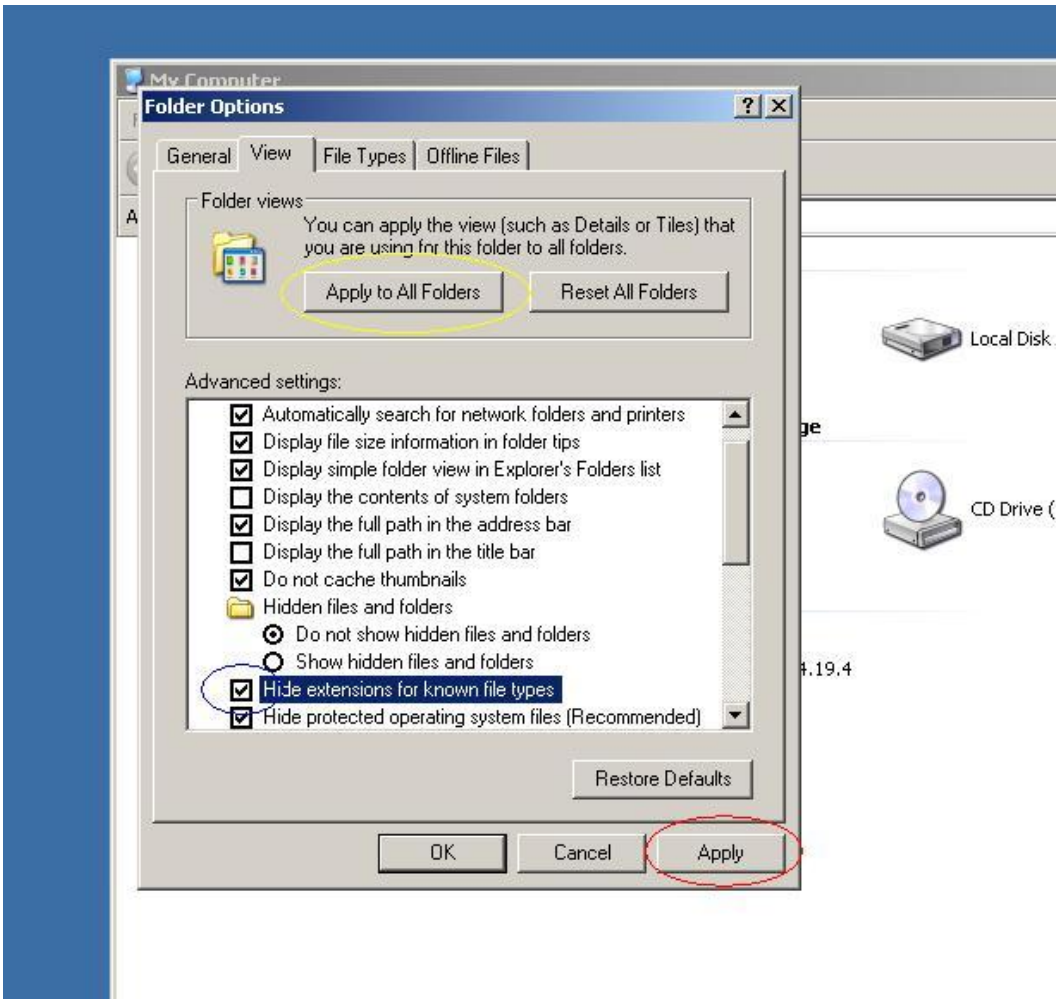


- Sinulle tulisi aueta ikkuna, joka sisältää erinäisiä kansioiden asetuksia. Klikkaa **välilehtivalikosta "Näytä"-lehteä** (merkitty seuraavan sivun kuvaan punaisella ympyrällä), jolloin seuraavanlainen ikkuna aukeaa:

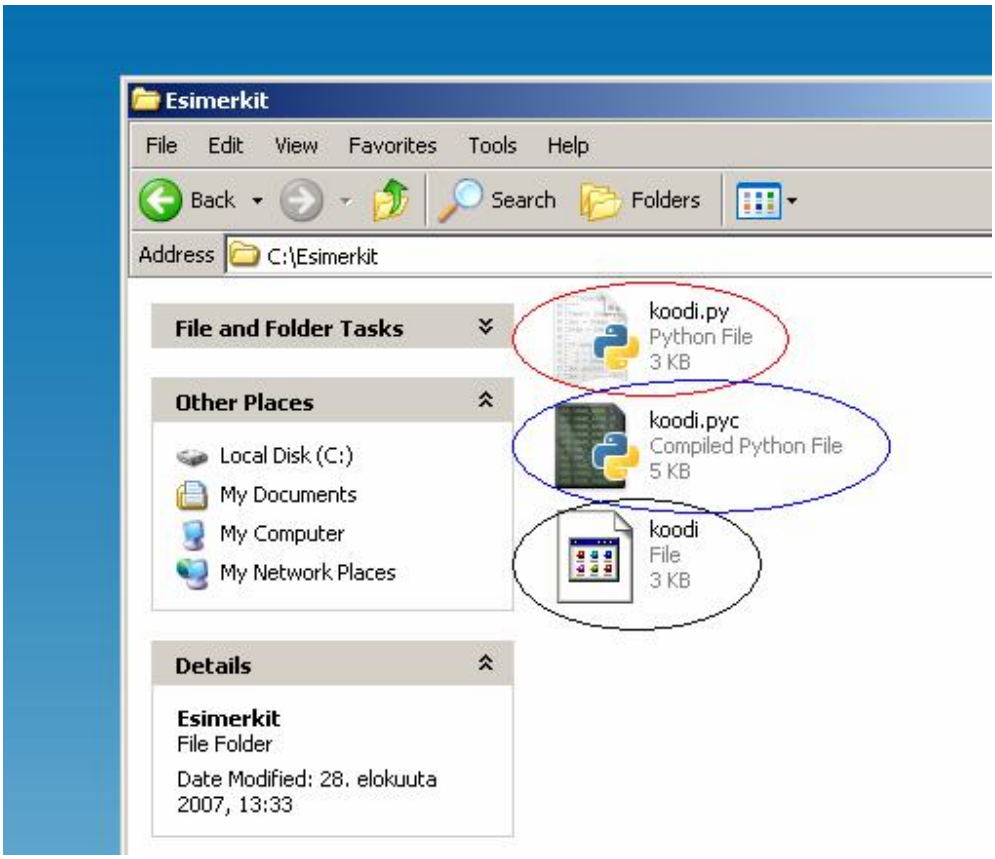


- **Poista rasti** laatikosta, jonka selitteessä lukee "Piilota tiedostopääte tunnetuilta tiedostotyypeiltä" (korostettu sinisellä palkilla).

- Kun olet varma, että poistit rastin (keskimmäinen ympyrä), paina ikkunan alareunassa olevaa nappia "Käytä" (Alin ympyrä) ja tämän jälkeen nappia "Käytä asetuksia kaikissa kansioissa" (Ylin ympyrä). Nyt voit sulkea ikkunan painamalla nappia "OK".



Tiedostopääte on nyt otettu näkyviin. Voit tarkastaa asian selaamalla kansioitasi, jolloin näet että kaikkia tiedostonnimi seuraa nyt tiedostopääte.



- Yllä olevassa esimerkkikuvassa näet tiedostonpäätteet toiminnassa. Jos asensit IDLE:n oikein, näet lähdekooditiedostojesi päätteenä ".py"-tunnisteen. Lisäksi tiedostosi ikonina on ylimmän esimerkin tavoin valkoinen paperi ja Python-logo.

- Kuvassa keskimmäisenä on myös logollinen ikoni mustalla paperilla, ja sen tunniste on ".pyc". **.PYC EI OLE LÄHDEKOODITIEDOSTO**, vaan esikäännetty koodi jolla tulkki nopeuttaa toimintaansa. Älä koskaan avaa tai editoi näitä, vaan ainoastaan valkoisella kuvakkeella merkittyjä ".py"-tiedostoja.

- Kuvassa on myös alimpana mustalla ympyrällä merkittynä tiedosto, jolla ei ole päätettä. Tämäkin tiedosto sisältäisi Python-koodia, mutta puuttuvan päätteen takia sen enempää Windows kuin IDLE:kään ei sitä tunnista oikein. Muista siis aina tarkastaa, että tallentamasi koodi sisältää oikeanlaisen päätteen.

Windows XP ympäristömuuttujien asettaminen

Mikä on ympäristömuuttuja?

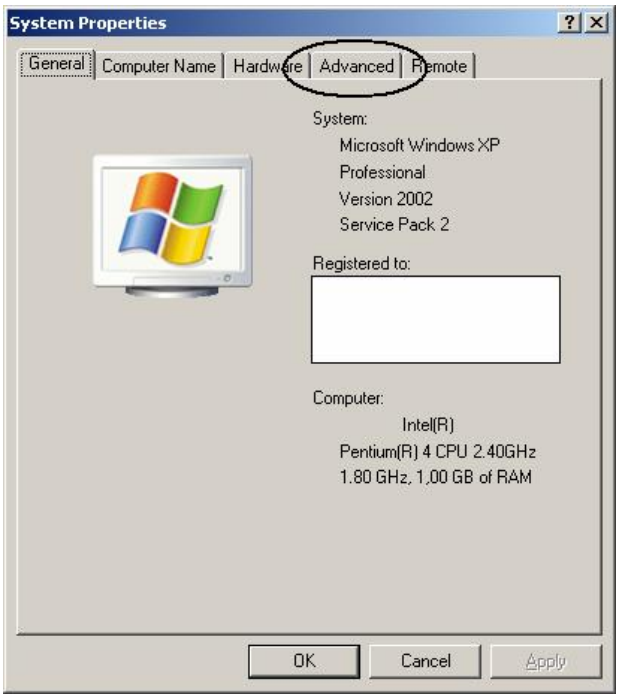
- Ympäristömuuttuja on muuttuja, jonka avulla Windows-käyttöjärjestelmälle kerrotaan, mistä kansiosta jokin asia löytyy.
- Käytännössä teemme siten, että kerromme Windows-käyttöjärjestelmälle sijainnin, mistä Python-tulkki löytyy jotta voimme myöhemmin ajaa komentorivikehotteen avulla ohjelmia.

Et välttämättä vielä tässä vaiheessa ymmärrä kaikkia työvaiheita ja niiden merkitystä ohjelmointiympäristön toiminnalle, mutta tekemällä nämä asennusvaiheet varmistat että ohjelmointiympäristö toimii oikein. Ensimmäisen kerran tarvitsemme komentoriviparametreja viikolla 8, joten siihen asti pärjäät myös ilman tätä asennusvaihetta, mikäli et sitä tässä vaiheessa halua suorittaa.

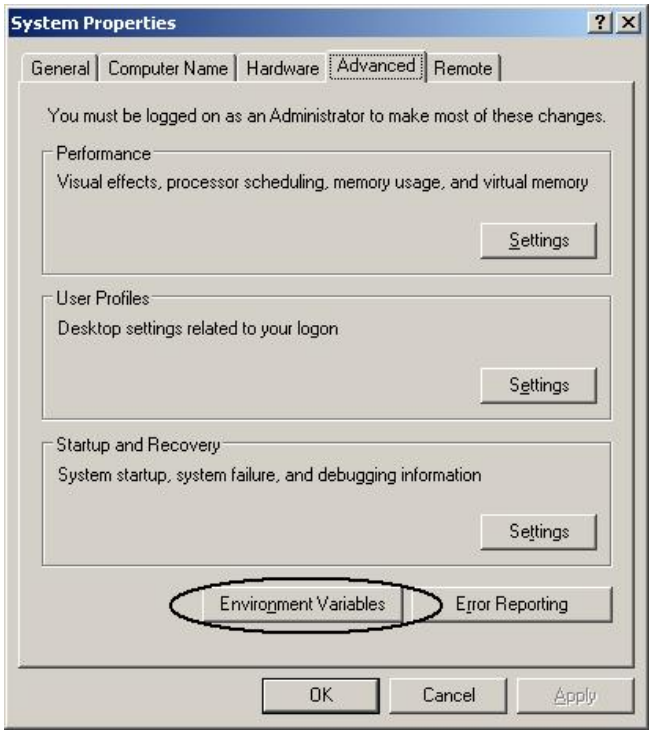
Muuttujan asettaminen vaiheittain

Asennus aloitetaan valitsemalla Käynnistä-valikosta ”Oma tietokone” ja klikkaamalla sitä hiiren toisella napilla. Aukeavasta alasvetovalikosta valitaan ”Ominaisuudet”

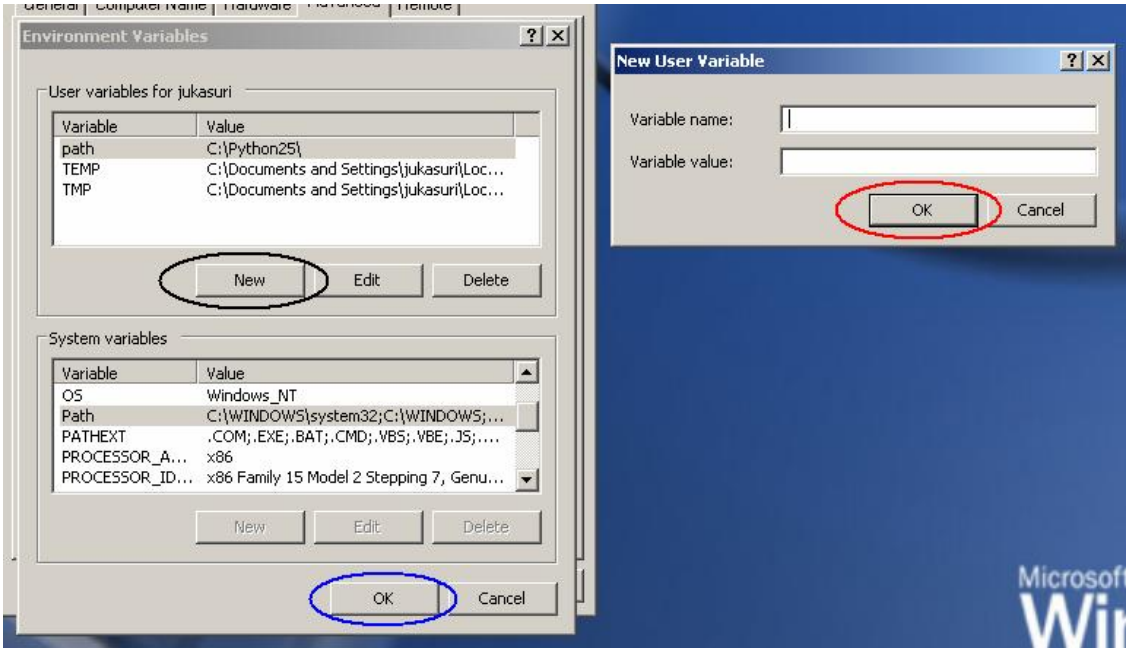
Aukeavasta valikosta valitaan välilehti ”Lisäasetukset”, joka kuvassa on merkattu mustalla soikiolla.



Tässä valikossa valitaan ruudun alareunassa oleva painike ”Ympäristömuuttujat”, joka kuvassa on ympyröity mustalla soikiolla.

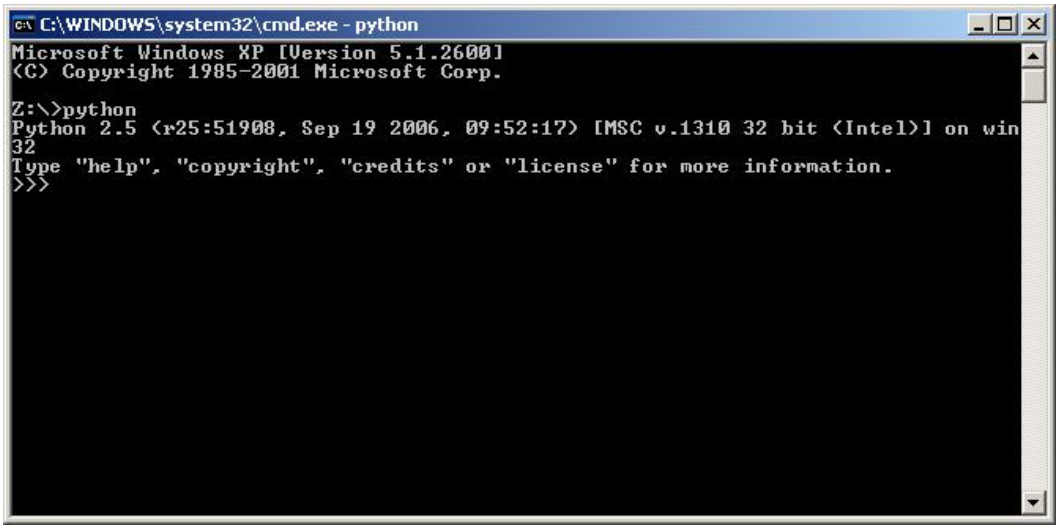


Nyt sinulla pitäisi olla seuraavanlainen ikkuna:



Painamalla mustalla soikiolla merkittyä painiketta, sinulle pitäisi aueta oikealla olevan ikkunan kaltainen syöttökenttä. Tähän kenttään aseta muuttujan nimeksi ”path” ja muuttujan arvoksi osoite, johon Python-tulkin asensit (oletus C:\Python25\). Hyväksy lisäys painamalla punaisella merkittyä OK-painiketta, ja sulje ikkuna sinisestä OK-painikkeesta.

Tämän jälkeen voit sulkea loputkin auki olevat ikkunat ja käynnistää tietokoneen uudelleen. Kun olet uudelleenkäynnistänyt, paina näppäinyhdistelmää Start-R tai valitse Käynnistä-valikosta valinta ”Suorita…”. kirjoita aukeavaan ikkunaan komento ”command”, jolloin tämän kaltaisen ikkunan tulisi aueta:



Kirjoita komentokehotteeseen ”Python”. Mikäli Python-tulkki käynnistyi oikein, olet asentanut ympäristömuuttujat onnistuneesti. Voit sulkea ikkunan ensin sammuttamalla tulkin näppäinkomennolla Ctrl-Z ja tämän jälkeen kirjoittamalla komentokehotteeseen käskyn ”exit”.

Huomioita koodieditorin valinnasta

Mikäli et aio käyttää Python-lähdekoodin kirjoittamiseen IDLE-editoria, joudut valitsemaan sitä varten jonkin muun sopivan editointiohjelman. Pythonia käytettäessä editorin valinta on kriittistä, sillä osa Pythonin yksinkertaisesta syntaksista nojautuu kunnollisen editorin tuomaan tukeen. Oikein valittu editori tekee Python-ohjelmoinnista helppoa, sekä auttaa sinua löytämään virheitä ennen kuin edes yrität ajaa kirjoittamasi ohjelman. Tämän ansiosta pääset nopeammin ja helpommin eteenpäin, etkä joudu tuhlaamaan aikaa etsiessäsi turhia tai tarpeettomasti tehtyjä virheitä.

Tärkeä perusvaatimus editorille on se, että se tukee **käskyjen korostusta.** Tämä tarkoittaa sitä, että kirjoittamasi Python-koodin osat on värjätty ja merkitty siten, että erotat koodin loogisen rakenteen paremmin samalla kun kirjoitat sitä. Tämä toiminto myös pienentää kirjoitusvirheiden sekä ohjausmerkkien unohtamisen todennäköisyyttä merkittävästi.

Jos olen Windows-käyttäjä, suosittelemme että käytät IDLEä. IDLE osaa käskyjen korostuksen sekä tukee Python-ympäristöä monin tavoin; esimerkiksi koodin ajaminen on IDLEn avulla mahdollista ilman turhia välivaiheita. Erityisen tärkeää on joka tapauksessa muistaa seuraava; **älä käytä Notepadia** – se on huono valinta kahdesta syystä; se ei tue korostuksia, eikä siitä ei löydy minkäänlaista tukea sisennyksien hallintaan. Pythonissa tämä on tärkeä seikka, josta tulemme myöhemmin puhumaan enemmän. Hyvä editoriohjelma, esimerkiksi IDLE tai VIM, osaa automaattisesti myös tämän asian.

Jos olet Linux- tai FreeBSD-käyttäjä, on sinulla paljon suurempi valinnanvapaus editoriohjelman suhteen. Jos olet kokenut ohjelmoija, olet luultavasti jo aiemmin käyttänyt VIMiä tai Emacsia. Nämä molemmat ovat tehokkaita editoreja, eikä Pythoninkaan tapauksessa ole syytä etsiä vaihtoehtoja. Aloittelevien ohjelmoijien kannattaa tutustua esimerkiksi Kate:een, tai käyttää myös Windows-puolelta tuttua IDLEä.

Jos haluat tutkia myös muita vaihtoehtoja, ylläpitää Python Software Foundation (www.python.org) kattavaa listaa Python-editoreista. Lisäksi, mikäli haluat mieluummin kokonaisen Python-kehitysympäristön, on niistäkin saatavilla lista samasta osoitteesta. Viimeistään siinä vaiheessa, kun aiot kirjoittaa laajempia ohjelmia Pythonilla, tulet hyötymään kokonaisesta kehitysympäristöstä.

Liite 2: Lyhyt ohje referenssikirjastoon

LTY

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

Allaolevaan taulukkoon on listattu ne luvut, joista luultavimmin löytyy lisätietoa tämän oppaan läpikäymistä asioista. Mikäli haluat tutustua luvun sisältöön, mene Internetissä osoitteeseen http://docs.python.org/lib/lib.html) ja etsi vastaava luku. Verkossa olevat dokumentit ovat englanninkielisiä.

Luku	**Sisältö**
2.1	Sisältää tietoa Python-ympäristön sisäänrakennetuista funktioista ja käskyistä kuten input(), print(), import sekä len()
2.3.1 ja 2.3.2	Tietoa Boolean-arvoista sekä loogisista väittämistä (and, or, not).
2.3.3	Tietoa vertailuoperaattoreista (<,>, !=)
2.3.4	Tietoa numeerisista tyypeistä (int, float jne.) sekä operaatioista joissa niitä voi käyttää (+,-,*, int() jne.)
2.3.6	Tietoa merkkijonoista ja sarjamuuttujista, lisäksi käsitellään operaatioita joita niillä voidaan tehdä sekä leikkauksista.
2.3.6.1	Kaikki merkkijonojen kanssa käytettävissä olevat metodit.
2.3.9	Sisältää tietoa erilaisista tiedostokahvojen käsittelyyn käytettävistä funktiosta, esim. fseek(), readline() jne.
3.1	sys-kirjastomoduulin esittely
4.1.1	Merkkijonojen merkkisarjoja
5.7	math-kirjastomoduulin esittely
5.9	random-kirjastomoduulin esittely
6.1	os-kirjastomoduulin esittely
6.10	datetime-kirjastomoduulin esittely
6.11	time-kirjastomoduulin esittely
11.5	urllib2-kirjastomoduulin esittely
16.1	Tkinter-kirjastomoduulin esittely
16.5	Lyhyt ohje IDLE-ohjelmointiympäristöön

Liite 3: Yleinen Python-sanasto

LTY

Tämä liite sisältää lyhyen sanakirjan sanoista, joita käytetään yleisesti ohjelmoinnin, Python-ympäristön tai ohjelmointilähtöisen tietotekniikan parissa.

Sana tai Termi	Selite
ajoympäristö	Ajoympäristö on tietokoneen sisäisten osien, käyttöjärjestelmän ja tulkin yhdessä muodostama kokonaisuus, missä 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 läpikäy arvoja systemaattisesti. Esimerkiksi toistorakenteen kierroslukulaskuri on malliesimerkki askeltajasta.
automaattinen muistinhallinta	Kts. dynaaminen muistinhallinta.
debuggeri	Debuggeri on ohjelma, jolla voidaan kontrolloida ohjelman ajonaikaista toimintaa, valvoa tulkin etenemistä sekä tarkastella ohjelmansisäisien muuttujien arvoja. Debuggeria käytetään virheiden löytämiseen ja poistamiseen.
dynaaminen muistinhallinta	Dynaamisella muistinhallinnalla tarkoitetaan järjestelmää, joka suorittaa automaattisesti tietokoneen keskusmuistin varaamisen ja vapauttamisen ohjelman tarpeiden mukaisesti. Vastakohta manuaalinen muistinhallinta
editori	Kts. koodieditori
ehtolauseke	Ehtolauseke on looginen väittämä, joka liitetään if- ja elif-rakenteisiin. Jos ehtolauseke on Tosi, suoritetaan se osio, johon lause oli liitettynä. Muussa tapauksessa suoritetaan rakenteen Else-osio tai sen puuttuessa rakenne ohitetaan kokonaan.

ehtorakenne	Ehtorakenne on rakenne, jossa suoritettavan koodiosion valinta perustuu siihen, täyttyykö ehtolauseke vai ei. Pythonissa ehtolausekkeena on if-elif-else-rakenne.
funktio	Funktio on koodista muodostettu looginen kokonaisuus, joka voi itsenäisesti suorittaa sille annettuja tehtäviä.
funktiokirjasto	Yleisohjelmointitermi joka tarkoittaa käytännössä samaa kuin kirjastomoduuli.
järjestelijä (rooli)	Järjestelijä on muuttuja, johon tallennetaan arvoja järjestykseen asettamista varten. Pythonissa normaalisti lista.
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.
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 merkkaa sen missä kohtaa tiedostoa ollaan lukemassa sen sisältöä tai mihin kohtaan seuraavaksi kirjoitetaan. Siirtyy automaattisesti luku- ja kirjoitusfunktioiden mukana.
kirjastomoduuli	Kirjastomoduulilla tarkoitetaan sellaista moduulia, joka on toimitettu asennuspaketin mukana. Esimerkiksi *sys* tai *random.*
kokoaja (rooli)	Kokoaja on muuttuja, johon tallennetaan tulos kaikista siihen mennessä läpikäydyistä arvoista.
komentorivi (kehote)	Komentorivikehote on ikkuna, johon käyttäjä syöttää tekstimuotoisia käskyjä joilla ohjataan tietokonetta.
kommentti	Kommentti on lähdekoodiin lisätty merkintä, jota tulkki tai kääntäjä ei huomioi ja joka on ensisijaisesti tarkoitettu koodaajan omiksi muistiinpanoiksi. Kommenttilauseita saatetaan kuitenkin joissain tilanteissa käyttää mm. käyttöympäristöä koskevien meta-tietojen antamiseen.
koodieditori	Tekstinkäsittelyohjelma, joka on suunniteltu ohjelmointia varten. Perusominaisuuksiin yleisesti kuuluu mm. käskysanojen ja rakenteiden korostaminen sekä automaattinen sisennyksien hallinta.

kulkija (rooli)	Kulkija on muuttuja, jota käytetään jonkin tietorakenteen läpikäymiseen. Harvinainen Python-ympäristössä.
kytkin (muuttuja)	Kts. yksisuuntainen lippu (rooli)
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 tietomuotoa, johon voidaan käsin määritellä jäsenmuuttujat sekä jäsenfunktiot.
lähdekoodi	Tiedosto tai joukko tiedostoja, johon on tallennettu varsinaiset koodirivit yhdestä tehtävästä tai ohjelmasta.
manuaalinen muistinhallinta	Manuaalinen muistinhallinta on muistinhallintatapa, jossa käyttäjä 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 näppäinkartasta, jolla koodi on kirjoitettu, tieto siitä, millä kielellä lähdekoodin kommentit ja tulostukset ovat jne.
metodi	Kts. jäsenfunktio
moduuli	Python-kielessä moduulilla tarkoitetaan lähdekoodiin ulkopuolelta sisällytettyä tiedostoa, joka sisältää funktioita sekä muuttujan arvoja.
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 '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.
osio (koodi-)	Pythonissa koodiosiolla tarkoitetaan rakenteensisäistä palaa koodia, joka on merkitty yhdeksi kokonaisuudeksi sisennystason avulla. Esimerkiksi if-rakenne sisältää vähintään yhden osion, if-osion.
palautusarvo	Palautusarvo on se arvo, jonka funktio lähettää sitä kutsuneelle funktiolle 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 pääryhmän (70%) muodostaa kiintoarvo, askeltaja ja tuoreimman säilyttäjä.
sarjallinen muuttuja	Sarjallinen muuttuja on muuttuja, joka sisältää alkioita tai tietueita ja jota for-lause voi muokata suoraan tietueesta seuraavaan siirtymällä. Pythonissa on kolme erilaista sarjallista muuttujaa: lista, tuple sekä sanakirja. Sarjallisen muuttujan englanninkielinen termi on ”*Sequence*”.
semantiikka	Semantiikka tarkoittaa kielen loogisuutta. Tämä eroaa syntaksista sillä, että semantiikka koskee kielen tarkoittamaa asiaa, ei sen rakenteellista oikeellisuutta. Esimerkiksi lause "Laiska ajatus myy sinisiä filosofeja autolle." on syntaksisesti aivan oikein mutta semanttisesti täysin mieletön.
seuraaja (rooli)	Seuraaja on muuttuja, johon tallennetaan jokin aiemmin käytetty muuttujan arvo mahdollista jatkokäyttöä varten.
shell-ikkuna	Linux-puolen vastaava termi Windows-puolen komentorivikehotteelle. Kts. Komentorivikehote.
sisäinen funktio	Funktio, jonka käyttämistä varten ei erikseen tarvitse tehdä funktiomäärittelyä tai antaa sisällytyskäskyä. Esimerkiksi len() tai print().
sopivimman säilyttäjä (rooli)	Sopivimman säilyttäjä on muuttuja, johon on tallennettu paras tai soveltuvin toistaiseksi löydetty arvo.
syntaksi	Syntaksi tarkoittaa kielioppia. Erityisesti tietojenkäsittelytieteissä syntaksilla tarkoitetaan varattuja sanoja sekä käskylauserakenteita. Esimerkiksi lause ”Python kieli ohjelmointi on.” on sisällöllisesti oikein, mutta syntaktisesti väärin. Syntaktisesti oikein se onkin ”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. Pythonin tiedostopääte on .py.
tietue	Tietue on sarjallinen osamuuttuja, joka koostuu useasta alkiosta. Esimerkiksi sanakirjan osamuuttujat ovat tietueita, jotka sisältävät kaksi alkioa, avaimen ja arvon.
tilapäissäilö (rooli)	Tilapäissäilö on muuttuja, johon tallennetaan lyhytaikaista säilytystä varten jokin tietty arvo. Verrattavissa laskimen muisti-toimintoon.
toistoehto (lauseke)	ehto, jonka totuusarvo tarkastetaan aina toistorakenteen uuden kierroksen alkaessa. Jos ehto on Tosi, suoritetaan kierros, jos taas Epätosi, lopetetaan toistorakenne.
toistorakenne	Toistorakenne on ohjelman rakenne, jota toistetaan kunnes jokin haluttu toistoehto saavutetaan. Pythonissa toistorakenteita ovat while- ja for-rakenteet

tulkki	Tulkki on ohjelma, jolla voidaan suorittaa lähdekooditiedostoja. Tulkki lukee lähdekooditiedostoja sitä myöten, 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 käyttöönotettu ulkoisesta tiedostosta.
ulkoinen tiedosto	Ulkoinen tiedosto tarkoittaa mitä tahansa tiedostoa, joka ei ole varsinainen lähdekooditiedosto.
vakio (-arvo)	Kts. kiintoarvo (rooli)
yksisuuntainen lippu (rooli)	Yksisuuntainen lippu on muuttuja, joka säilyttää arvoa, jolla valvotaan esim. ehtolausekkeen täyttymistä.

Liite 4: Tulkin virheilmoitusten tulkinta

Python-tulkin standardivirheilmoitukset ja niiden selitteet

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 metodin tai jäsenfunktion nimen väärin. Toinen vaihtoehto on että yritit käyttää pistenotaatiota paikassa, jossa sitä ei voi käyttää.
EOFError	input() tai raw_input() ei saanut luettavaa arvoa.	Lopetit ohjelman suorituksen CTRL-D-yhdistelmällä syöte-kehotteessa.
IOError	kirjoitus- tai tulostusoperaatio epäonnistui.	Koetit lukea tiedostoa, jota ei ole olemassa tai koitit kirjoittaa lukumoodilla tai levy, jolle kirjoitit, täyttyi.
ImportError	import-käskyn suoritus ei onnistunut.	Moduuli, jonka koetit sisällyttää, oli kirjoitettu väärin tai tallennettu paikkaan josta tulkki ei sitä löytänyt.
IndexError	Annettu sijainti ylitti jonon tai listan pituuden.	Yritit lukea merkkiä tai alkiota, joka on merkkijonon tai listan ulkopuolella.
KeyError	Pyydettyä avainta ei löydy.	Koetit lukea sanakirjasta tietuetta, jonka avainta ei löytynyt.
KeyBoardInterrupt	Tapahtui näppäimistökeskeytys.	Keskeytit tulkin ajon näppäinyhdistelmällä CTRL-C (tai vastaava).
NameError	Annettu nimi on virheellinen.	Koitit joko alustaa muuttujaa tai funktiota epäkelvolla nimellä tai kirjoitit nimen väärin.
RuntimeError	Tapahtui yleinen virhe.	Python-tulkki ei osannut määritellä millainen virhe tapahtui, mutta jotain meni vikaan.
SyntaxError	Koodin syktaksissa on	Olet luultavasti sisentänyt jonkin

	virhe.	rivin väärin tai koodistasi puuttuu pilkkuja tai sitaattimerkkejä.
SystemError	Tulkin sisäisissä tiedostoissa tapahtui virhe.	Python-tulkin asennus on mennyt jostain syystä rikki. Aja korjausasennus asennuspaketista.
SystemExit	Ohjelma lopetti toimintansa.	Lopetit ohjelman funktiolla sys.exit().
TypeError	Tietotyypeissä on yhteensopivuusongelma	Koetit muuttaa merkkijonon numeroarvoksi tai tehdä laskutoimituksen merkkijonolla tai kirjoittaa tiedostoon ei-merkkijonoarvon.
UnboundLocalError	Viittasit muuttujaan, jolla ei ole arvoa.	Koetit käyttää tunnettua muuttujaa, jolle ei ole määritelty arvoa paikassa, jossa muuttujalla on oltava yksiselitteinen arvo.
Unicode … Error (useampia)	unicode-merkkirivin käsittelyssä tapahtui virhe.	Koetit käyttää unicode-riviä, jota ei saatu käännettyä tai joka sisälsi virheitä.
ValueError	Arvon määrittelyssä tapahtui virhe.	Koitit antaa operaattorille tai funktiolle arvon, joka on oikeantyyppinen mutta sopimaton.
ZeroDivisionError	Ohjelmassa tapahtui nollalla jako.	Yritit jakaa luvun nollalla tai muuttujalla, jonka arvo oli 0. Tapahtuu myös jakojäännös- ja tasajako-operaattorien kanssa.

Lisäksi on olemassa vielä joitakin virhetiloja, jotka kuitenkin ovat niin erikoisluonteisia tai epätodennäköisiä, että niiden läpikäyminen on triviaalia. Täydellinen lista virheilmoituksista löytyy kuitenkin myös 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 5: Python-ohjelmointikielen tyylisäännöt

Alkuperäinen lähde Python Enhancement Proposal 8, http://www.python.org/dev/peps/pep-0008/

Ohjelmoinnin tyyleistä

Tässä liitteessä määrittelemme joitain yksinkertaisia tyylisääntöjä, joita noudattamalla pystymme parantamaan tuottamamme lähdekoodin tasoa. Ohjelmointikielen tyyleillä nimittäin tarkoitetaan tapaa, jolla lähdekoodi muotoillaan. Normaalisti tällaisen oppaan määrittelyllä pyritään lähinnä helpottamaan lähdekoodin lukemista, mutta tyyleissä on myös muutamia muita hyötyseikkoja. Jos esimerkiksi kirjoitat aina lähdekoodisi tietyllä tyylillä tai käytät muuttujien nimeämiseen samaa logiikkaa, opit pian näkemään helpommin koodissasi olevia virheitä ja ongelmia. Toisaalta, jos kirjoitat koodia muiden ihmisten kanssa, helpottaa yhteinen esitysasu toisen ihmisen kirjoittaman koodin hahmottamista. Viimeisenä kannattaa vielä tietenkin muistaa, että yhtenäinen tyyli ja kommentointi auttavat myös sinua itseäsi jos joskus joudut palaamaan aiemmin tekemäsi – ja nyttemmin yksityiskohdat unohtamasi – lähdekoodin pariin.

Perusajatus Python-ohjelmoinnille

Guido van Rossumin alkuperäinen ajatus ohjelmoinnista on, että ihmiset käyttävät enemmän aikaa koodin tulkitsemiseen kuin sen kirjoittamiseen. Tämän vuoksi Python-ohjelmoinnissa ja – ohjelmointityylissä pyritään aina suosimaan kielen ymmärrettävyyttä sekä pitämään lähdekoodin kieliasu yhtenäisenä. Kannattaa kuitenkin pitää mielessä, että joskus on olemassa tilanteita, joihin tietyt tyylisäännöt tai tyylioppaat eivät vain yksinkertaisesti päde. Tällöin tulee pyrkiä pitämään linja yhtenäisenä ja huolehtia Python-koodin tärkeimmästä ominaisuudesta – sen ymmärrettävyydestä ja selkeydestä.

Seuraavaksi esittelemme aihealueittain Python-ohjelmointikielelle jotain perussääntöjä, joita tulisi opetella noudattamaan jo ohjelmoinnin alkuvaiheista lähtien.

Tyylisäännöt

Sisennyksestä

Käytä sisennyksissä koodin tasolta toiselle aina neljää (4) välilyöntiä per taso. Ainoan poikkeuksen tähän tekee erittäin vanhojen lähdekoodien, jotka käyttävät aiemmin voimassa ollutta 8 välilyönnin standardia, ylläpito ja korjaus.

Älä koskaan käytä yhtäaikaisesti sisennysmerkkejä 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.

Tyhjistä riveistä

Erota määritellyt funktiot ja dynaamiset rakenteet tosistaan vähintään kahdella tyhjällä rivillä. Erottele määritellyt funktiot muusta koodista tyhjällä rivillä ennen funktiota ja funktion jälkeen. 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:

```
Näin: import os
      import sys

Ei näin: import sys, os
```

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 header-kommentit tulee 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)

Ryhmät tulee erotella toisistaan tyhjällä rivillä.

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 voidaan sallia.

Välilyönneistä

Välilyöntien käyttäminen koodin luettavuuden parantamiseksi on hyvä käytäntö, mutta siihen liittyy muutamia sääntöjä, joilla haluttua vaikutusta voidaan tehostaa.

Älä käytä välilyöntiä seuraavissa kohdissa:

- Välittömästi kaarisulkeiden, hakasulkeiden tai aaltosulkeiden perään.

```
Näin: leipa(voita[1], {juustoa: 2})
Ei näin:  leipa( voita[ 1 ], { juustoa: 2 } )
```

-Ennen pilkkua, kaksoispistettä tai puolipistettä:

```
Näin: if x == 4: print x, y; x, y = y, x
Ei näin:   if x == 4 : print x , y ; x , y = y , x
```

Ennen sulkeita, jotka aloittavat funktiokutsun parametrilistan:

```
Näin: kinkku(1)
Ei näin:  kinkku (1)
```

Ennen sulkeita, jotka määrittelevät leikkauksen tai alkioviittauksen:

```
Näin: dict['avain'] = list[index]
Ei näin:   dict ['avain'] = list [index]
```

Enemmän kuin yksi välilyönti viittausten tai sijoitusten ympärillä:

```
Näin:

    x = 1
    y = 2
    pitka_nimi = 3

Ei näin:

    x          = 1
    y          = 2
    pitka_nimi = 3
```

Käytä välilyöntiä seuraavissa kohdissa:

Erottele seuraavat merkit ja vertausoperaatiot aina molemmin puolin välilyönneillä:

```
sijoitus (=), arvoa muuttava sijoitus (+=, -= etc.),
vertailut (==, <, >, !=, <>, <=, >=, in, not in, is, is not),
Boolean-arvot (and, or, not).
```

Erottele numeroarvot ja muuttajat lasku- ja sijoitusoperaattoreista:

```
  Näin:

    i = i + 1
    vastaanotettu += 1
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)

Ei näin:

    i=i+1
    vastaanotettu +=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:

```
Näin:

    def complex(real, imag=0.0):
        return magic(r=real, i=imag)

Ei näin:

    def complex(real, imag = 0.0):
        return magic(r = real, i = imag)
```

Kommentoinnista

Yleisesti

Kommentit, jotka eivät pidä yhtä koodin toiminnan kanssa 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.

Kirjoita kommentointi aina englanniksi, mikäli on pienikin todennäköisyys, että koodisi voi päätyä julkisesti saataville ja mikäli sinua ei ole erikseen ohjeistettu tekemään toisin.

Koodiosion kommentointi

Jos kirjoitat koko koodiosiota koskevaa kommenttitekstiä, sisennetään se samalle tasolle millä koodiosio on. Monirivisessä kommentoinnissa käytä menetelmää, jossa ’#’-merkin jälkeen tulee ainakin yksi välilyönti. Kappaleenvaihto monirivisessä kommentissa merkitään tyhjällä kommenttirivillä, jolla on ainoastaan ’#’-merkki.

Koodinsisäinen kommentointi

Koodinsisäinen kommentointi tarkoittaa kommentteja, jotka kirjoitetaan koodirivin perään. Koodinsisäinen kommentti erotetaan koodirivistä vähintään kahdella välilyönnillä. Lisäksi koodinsisäiset kommentit häiritsevät koodin lukemista, joten niitä ei tulisi käyttää kuvaamaan itsestään selviä asioita. Esimerkiksi kommentti

```
x = x + 1                 # x kasvaa yhdella
```

on turha ja haittaa koodin luettavuutta. Sen sijaan kommentti

```
x = x + 1                 # kasvatetaan listan ylarajaa yhdella
```

on tietyissä tapauksissa hyödyllistä tietoa.

Nimeämiskäytännöistä

Vaikka alkuperäinen nimeämiskäytäntö Pythonin moduulikirjastoissa on välillä hieman sekava, on silti tarkoituksena jatkossa pyrkiä nimeämään funktioiden ja muuttujien nimet siten, että ne noudattavat tiettyä kaavaa.

Nimeämislogiikasta

Käytä muuttujien, funktioiden ja rakenteiden nimeämiseen jotain seuraavista tavoista:

```
    - b (yksittäinen pieni kirjain)

    - B (yksittäinen iso kirjain)

    - muuttujannimi (pelkkiä pieniä kirjaimia)

    - muuttujan_nimi_viivalla_erotettuna

    - MUUTTUJANNIMI (pelkkiä isoja kirjaimia)

    - MUUTTUJAN_NIMI_VIIVALLA_EROTETTUNA

    - IsotKirjaimet (muuttujan nimen sanat alkavat isolla kirjaimella)

      Huomioi tämä: Jos käytät tätä nimeämislogiikkaa, niin käytä isoja
      kirjaimia myös lyhenteissä: HTTPServerError on parempi kuin
      HttpServerError.

    - pieniAlkuKirjain (sama kuin yllä, mutta alkaa pienellä
kirjaimella)

    -  Isot_Alkukirjaimet_Viivalla_Erotettuna

    -  _aloittava_alaviiva (plus muut kombinaatiot)
```

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, muuttujien ja funktioiden nimistä

Moduuleilla tulisi olla lyhyt, kokonaan pienillä kirjaimilla kirjoitettu nimi. Myös alaviivaa voidaan käyttää, mikäli sillä voidaan parantaa nimen luettavuutta. Koska tulkki hakee moduuleja niiden tiedostonnimistä, olisi moduulin nimen syytä olla erittäin lyhyt ja yksinkertainen. Mikäli oletetaan, että koodia voidaan käyttää vanhoissa Mac- tai DOS-koneissa, on nimen hyvä olla maksimissaan 7 merkkiä pitkä. Samoin alaviivan käyttöä kannattaa näissä tapauksissa välttää.

Funktioiden nimien tulisi olla kirjoitettu kokonaan pienillä kirjaimilla. Lisäksi niiden kanssa voi käyttää alaviivaa, mikäli se parantaa nimen ymmärrettävyyttä. (”laske”,”_tarkasta”)

Mikäli haluamasi muuttujannimi on järjestelmän varattu sana (esimerkiksi ”except” tai ”print”), on parempi menetelmä jättää muuttujannimestä kirjain pois (”xcept”, ”prnt”) kuin lisätä alaviiva (”except_”,”_print”). Tietenkin paras tapa on olla käyttämättä varattujen sanojen kaltaisia muuttujannimiä.

Normaalien muuttujien nimien tulisi olla kuvaavia, sekä mahdollisuuksien mukaan lyhyitä ja ytimekkäitä (luku1, luku2, syote, laskuri, askel …). Pyri välttämään lyhenteiden käyttöä muuttujan nimissä. Älä käytä mitään-tarkoittamattomia merkkijonoja (tlst, kmnt_x, asefw4, blaa1, blaa2 …) muuttujien niminä.

Huomioita

Tässä on lyhyesti kuvattuna Pythonin tärkeimmät tyylisäännöt. 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/Languages/Finnish.

Lisäksi voit tutustua myös alkuperäiseen Python-tyylioppaaseen, joka löytyy osoitteesta http://www.python.org/dev/peps/pep-0008/. Tämä opas on englanninkielinen.

Python 2.5 versiohistoria ja lisenssi

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes

Release	Derived from	Year	Owner	GPL compatible?
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.5	2.4	2006	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2006 Python Software Foundation; All Rights Reserved'' are retained in Python 2.5 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY

DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI"), and the Individual or Organization (``Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR

THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the ``ACCEPT'' button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,

WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.