Tero Tuovinen

Johdatus Python-kieleen

Tietotekniikan
kandidaatintutkielma
13. joulukuuta 2004

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Tero Tuovinen
Yhteystiedot: tttuovin@cc.jyu.fi
Työn nimi: Johdatus Python-kieleen
Title in English: An Introduction to Python
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 26
Tiivistelmä: Tutkielma esittelee Pythonin perusominaisuuksia, sekä kuvaa lyhyin esimerkein Pythonin syntaksin. Tutkielma tutustuttaa lukijan Pythonin rakenteisiin ja toimintamalleihin, jotka ovat muissa kielissä vieraita. Tutkielman lopussa esitetään lyhyt esimerkki Python-ohjelmasta.
English abstract: The thesis introduces basic functionalities of Python. The thesis guides through syntax, structures and unfamiliar functionalities. It also includes an example sourcecode in the end of the thesis.
Avainsanat: Python, ohjelmointikieli, syntaksi, perusteet, interaktiivinen, olio-ohjelmointi.
Keywords: Python, programming language, syntax, introduction, interactive, object-oriented programming.

Sisältö

Liitteet

1 Johdanto

Python on paljon käytetty, laajalle levinnyt ja lähes kaikkia käyttöjärjestelmiä tukeva ohjelmointikieli. Se on tavukoodia käyttävä **interaktiivinen** olio-ohjelmointikieli. Käännetty tavukoodi ajetaan virtuaalikoneen päällä, josta johtuen Python-ohjelma ei ole yhtä nopeaa kuin esimerkiksi vastaava optimoitu C++ -ohjelma. Interaktiivisuudella Pythonissa tarkoitetaan sitä, että Pythonia voi kirjoittaa suoraan komentotulkkiin. Lähdekoodia ei siis tarvitsee mitenkään tallentaa erilliseen tiedostoon ja ajaa. Tällä hetkellä uusin versio Pythonista on 2.4.

Kandidaatintutkielma opastaa Pythonin maailmaan, tutustuttaa Pythonin syntaksiin, sekä ohjeistaa etsimään lisää tietoa. Luvussa 2 esitellään yleisiä huomioitavia asioita Pythonilla ohjelmoitaessa. Luku 3 esittelee yleisimmät tietorakenteet ja dynaamisen tyypittämisen. Luvussa 4 tutustutaan funktioihin ja olio-ohjelmointiin. Luku 5 esittelee tiedostojen käsittelyä. Luvussa 6 tarkastellaan poikkeuksia ja luvussa 7 generaattoreita. Luku 8 luo katsauksen Pythonin kirjastojen käyttämiseen. Liitteenä on esimerkki Pythonilla toteutetusta ohjelmasta.

2 Lähdekoodin kirjoittaminen

Pythonin ohjelmoinnissa suositaan yleensä valmiita kirjastoja ja rakenteita, joiden avulla koodi saadaan pidettyä lyhyenä, hyvin luettavana ja helposti ymmärrettävänä. Python on suunniteltu näyttämään hyvin pitkälti oikealta englannilta, joten koodin muuttujien ja funktioiden nimet tulee valita huolellisesti kuvaamaan käsiteltävää asiaa. Tämä tyylillinen sääntö on tärkeä, jotta Pythonin ideologia ihmisille tarkoitetusta, luettavasta koodista säilyisi. Luvussa esiintulevia asioita käsitellään laajemmin Python Tutorialissa [5].

2.1 Komentotulkki

Komentotulkki käynnistyy kaikissa käyttöjärjestelmissä kirjoittamalla komentoriville `python`. Komentotulkki helpottaa koodin kirjoittamista, sillä syntaktiset ongelmat on helppo testata ilman koko lähdekoodin muuttamista ja ajamista. Käytännössä kaikki ohjelmat kirjoitetaan erillisiin tiedostoihin.

Käynnistyttyään komentotulkki siirtyy komentotilaan, jota osoittaa kunkin rivin alussa olevat merkit `>>>`. Komentotilassa se on valmis ottamaan vastaan käskyjä. Mikäli käsky on usemman rivin mittainen, kommentotulkki ilmoittaa kolmella pisteellä (`...`) odottavansa ohjelmoinnin jatkumista. Kun ohjelmoija haluaa lopettaa

käskyn kirjoittamisen, tyhjän rivin syöttäminen ajaa käskyt ja komentotulkki näyttää mahdolliset tulosteet ja virheet.

2.2 Rakenne

Python-koodin rakenne **määritellään pelkästään sisentämisen avulla**. Tämä tarkoittaa sitä, ettei Pythonissa käytetä koodilohkojen määrittämiseen mitään erityistä merkkiä, kuten esimerkiksi **C++:ssa** kaarisulkeita `{ }`. Sisennys toteutetaan joko tabulaattorilla tai määrätyllä määrällä välilyöntejä, mutta ei molemmilla.

Hyvä puoli **sisennysorientoituneessa** lähdekoodissa on parantunut luettavuus ja ulkonäöllisesti suhteellisen vakiintunut ulkoasu. Kolikon kääntöpuoli on ongelmallinen **syvyyden lisääminen**, sekä kasvanut inhimillisten virheiden mahdollisuus pitkissä ja syvissä funktioissa.

Seuraava esimerkki osoittaa, mitä syvyydellä tarkoitetaan:

```
if(true):
        if(false):
                if(1):
                        if(true):
                                print "Syvällä"
```

On varsin selvää, että mikäli esimerkkiin liitetään jokin uusi ehto välille, kaikkia syvemmällä olevia ehtoja tulee siirtää tabulaattorin verran syvemmälle. Mikäli rakenne on pitkä ja editori ei tue **lohkoittain sisentämistä**, inhimillisen virheen mahdollisuus kasvaa.

2.3 Kommentointi

Pythonissa kommentointimerkki on #. Pythontulkki ei lue tämän jälkeen mitään rivin loppuun kirjoitettua. Tätä ei tule kuitenkaan käyttää kommentoinnissa, vaan se tulee kirjoittaa lähdekoodin sisälle seuraavasti:

```
""" g  on lista numeroista välillä  1-10.  """
g = [1,2,3,4,5,6,7,8,9,10]
```

Kommentoitaessa koodin sisälle **(docstring)** hyötynä on se, että komentotulkissa käskyllä `help(jokin_funktio)` saadaan näytölle tulostumaan funktion kommentointi. Lisäksi on ohjelmia, jotka keräävät kommentoinnin ja antavat näin hyvän pohjan muulle koodin dokumentoinnille.

Kommentoinnissa tulee huomioida se, ettei Pythonissa funktioiden parametreille tarvitse määrittää mitään tyyppiä. Kuitenkin monesti odotetaan tietyntyyppisiä parametrejä, joten **haluttujen parametrien tyypit** on hyvä kirjoittaa muistiin kommentteihin. Vastaava kommentointikäytäntö on hyvä ottaa käyttöön myös paluuarvojen suhteen.

2.4 Tyypitys ja tulostus

Pythonissa on dynaaminen tyypitys. Tämä merkitsee sitä, että Python määrittelee muuttujan tyypin sen mukaan, mikä arvo siihen sijoitetaan. Seuraavassa esimerkissä muuttujiin sijoitetaan dynaamisesti tyypittyviä arvoja:

```
muuttuja1 = 15
muuttuja2 = "Pikatchu"
muuttuja3 = 0.432
```

Pythonissa dynaaminen tyypin määrittäminen ulottuu myös listojen ja muiden tietorakenteiden sisälle. On täysin sallittua täyttää lista loogisesti erityyppisillä muuttujilla seuraavalla tavalla:

```
lista1  = ["Pikatchu", 1 , 0.1]
lista2  = [muuttuja1, muuttuja2, muuttuja3]
```

Tulostuskomennolla `print` voidaan tulostaa muuttujan sisältö. Komennot

```
print lista1
print lista2
```

tulostavat listojen sisällön näytölle seuraavasti:

```
["Pikatchu", 1 , 0.100000]
[15, "Pikatchu", 0.432000]
```

HUOM! Dynaamisessa tyypityksessä tulee muistaa se tärkeä asia, että **yksinkertaisten tyyppien arvot sijoitetaan muuttujiin arvoina, mutta monimutkaisten tyyppien viitteinä**. Yksinkertaisia tietorakenteita ovat esimerkiksi lukuarvot. Monimutkaisia ovat mm. sanastot ja oliot.

3 Tietorakenteet

Pythonissa on valittavana hyvin monenlaisia mahdollisuuksia tiedon ohjelmansisäiseen taltiointiin. Luvussa esitellään merkkijonojen ja listojen lisäksi kaksi vähän tuntemattomampaa tietorakennetta: **Tuple** ja **Dict**. Luvussa esiteltyihin asioihin lisävalaistusta löytyy Python Tutorialista [5].

3.1 Merkkijonojen yhdistäminen

Pythonissa merkkijonoa voidaan ajatella listana, johon yksittäiset merkit on tallennettu. Luvussa 2.4 esiteltiin lyhyesti merkkijonon sijoittamista muuttujaan. Tässä luvussa paneudutaan merkkijonon käsittelyyn listana. Luvussa 3.1 esitellään merkkijonojen kasvattamista ja luvussa 3.2 näytetään, kuinka merkkijonosta valitaan haluttuja merkkejä.

Merkkijonojen yhdistäminen Pythonissa tapahtuu +-operaation avulla. Muuttujan arvoon `'14'` saadaan lisättyä merkkijono `'2'` komennolla `str(14)+2`, jolloin saadaan merkkijono `'142'`. Jollei merkkijonon ja kokonaisluvun välille ole kirjoitettu summasääntöä, palauttaa komentotulkki muuten virheen. Vastaavasti, mikäli halutaan lisätä numeroon `2` merkkijonoa `'14'` vastaava lukuarvo, kirjoittamalla

```
print 2 + int('14')
```

näytölle tulostuu haluttu arvo `16`.

Pythonissa voidaan merkkijono kirjoittaa joko yksinkertaisin heittomerkein `''` tai lainausmerkein `""` sekä kolminkertaisin heittomerkein `''''''` tai lainausmerkein `"""""""`. Olennainen ero eri merkintätapojen kesken on se, että kolmennetut merkit toimivat myös usean rivin lauseissa.

Täysin sallittua on myös **moninkertaistaa merkkijonot** sanomalla

```
a=3*"Bulbasaur"
print a
```

Tällöin tulostukseksi saadaan

```
BulbasaurBulbasaurBulbasaur
```

On mahdollista lisätä merkkijonoon toinen merkkijono komennolla

```
a=a+"Weezing"
print a
```

Tällöin näyttöön tulostuu seuraava merkkijono:

```
BulbasaurBulbasaurBulbasaurWeezing
```

3.2 Merkkien poimiminen merkkijonosta

Merkkijonoa voi siis käsitellä kirjainlistan tapaan. **Ensimmäisen kirjaimen indeksi on 0.** Kaikki muuttujan `a` merkit tulostetaan komennolla

```
print a[:]
```

Tulosteeksi tulee merkkijono `BulbasaurBulbasaurBulbasaurWeezing`.

Jos **halutaan poimia joku merkkijonon merkeistä**, siihen voidaan viitata seuraavasti.

```
print a[2]
```

Tulosteeksi saadaan kirjain `'l'`. Mikäli halutaan poimia useampia kirjaimia, kaksoispisteen avulla voidaan ilmaista väliä seuraavasti:

```
print a[2:6]
```

Näytölle tulostuu edellisestä käskystä `lbas`. Mikäli välille ei määritellä toista päätepistettä, oletetaan ohjelmoijan tarkoittavan alku- tai loppupistettä toisena päätepisteenä. Esimerkiksi

```
print a[6:]
```

tulostaa näytölle merkkijonon

```
aurBulbasaurBulbasaurWeezing
```

On mahdollista viitata merkkijonoon myös negatiivisilla indekseillä. Tällöin ohjelmoijan odotetaan tarkoittavan indeksejä lähtien viimeisestä merkistä. Esimerkiksi

```
print a[6:-5]
```

tulostaa näytölle kaikki muut merkkijonon kirjaimet, paitsi kuusi ensimmäistä ja viisi viimeistä, eli tulokseksi saadaan

```
aurBulbasaurBulbasaurWe
```

Yllä olevat esimerkit näyttävät, kuinka merkkijonosta valitaan haluttuja alkioita. Edellä mainituilla tavoilla voidaan kuitenkin valita myös mitkä tahansa listan alkiot riippumatta siitä, ovatko ne merkkijonoja vai eivät.

3.3 Lista ja For-toistorakenne

Listat ovat yksi tehokkaimmista Pythonin ominaisuuksista. Niiden luominen on helppoa ja läpikäyminen vaivatonta. Monesti listojen avulla pystytään korvaamaan lähdekoodin `if`-rakenteita. Seuraavassa määritellään tyhjä lista ja siihen lisätään yksi alkio:

```
lista=[]
lista.append("Tämä alkio on merkkijono.")
```

Lista voi sisältää kaikenlaista tietoa. Samassa listassa voi olla mm. numeroita, merkkijonoja, toisia listoja, funktioita ja tyhjiä listoja. Olemassaolevalta listalta voidaan kysyä, onko se **tyhjä** seuraavalla tavalla:

```
if lista!=[]:
        print "Lista ei ole tyhjä"
```

Listaa luotaessa `range`-funktio on monesti hyvin käyttökelpoinen. `range` palauttaa listan kokonaislukuja annettujen sääntöjen perusteella. Seuraavassa luodaan `range`-funktion avulla lista, jossa on alkiot nollasta yhdeksään.

```
g = range(10)  # g on lista numeroista välillä 0-9."""
```

`range`-funktion avulla on mahdollista toteuttaa helposti monenlaisia listoja. Tarkempi esittely niistä löytyy Python Library Referencessä [6].

Tarkastellaan seuraavaksi listan läpikäyntiä `for`-funktion avulla. Muuttuja `jokainen_alkio` toimii `for`-rakenteen sisällä paikkana, johon tallennetaan listan alkion arvo. Tämän jälkeen sitä pystytään käyttämään kuin mitä tahansa Pythonin muuttujaa esimerkiksi seuraavalla tavalla:

```
for jokainen_alkio in lista:
        print jokainen_alkio
        print sin(jokainen_alkio)
        if jokainen_alkio > 5:
                print jokainen_alkio + jokainen_alkio
```

Jos `lista` sisältää numerot `1`, `3` ja `0`, niin `for`-rakenne tulostaa

```
1
0.841470984808
3
0.14112000806
0
0.0
```

3.4 Monikko (Tuple)

Eräs vaihtoehtoinen tapa listoille on käyttää **monikkoa** eli `tuple`a. Se on hieman listaa tehokkaampi tapa tallentaa arvoja. **Monikko määritellään** Pythonissa seuraavasti:

```
tuple1 = 1 , 2 , 'cat'
```

Monikko voidaan myös **yhdistää** toiseen monikkoon seuraavasti:

```
tuple2 = (2,3,4) , tuple1
```

Monikko voidaan **purkaa** eri muuttujiin seuraavasti:

```
arvo1, arvo2, arvo3 = tuple1
```

Yhden alkion sisältävää monikkoa kutsutaan nimellä `singleton`. **Huomaa pilkun merkitys syntaksissa.** Seuraavan esimerkin toisella monikolla ei ole alkiota.

```
single = (1,)
empty = ()
```

Monikko ja lista eroavat toisistaan siten, että monikon arvoja ei voida muuttaa. Tietenkin voidaan luoda uusi monikko vanhalla nimellä, mutta yksittäisiä arvoja ei voida vaihtaa. Seuraavassa luodaan aikaisempien esimerkkien pohjalta lista ja monikko, ja yritetään vaihtaa niiden arvoja:

```
lista1 = [1,2,3]
lista1[1] = 4
tuple1 = 1,2,3
tuple1[2] = 3
```

Kun monikon arvoa yritetään vaihtaa, komentotulkki palauttaa seuraavan virheen:

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

Monikko on kätevä tapa esittää esimerkiksi koordinaatit (x,y,z). Lisäksi monikko sopii hyvin sanaston (`dict`, katso luku 3.5) avaimeksi, koska sen arvo ei muutu.

3.5 Sanasto (Dict)

Yksi kätevä tapa listata asioita Pythonissa on **sanasto** (`dict`). Sanastoa voidaan ajatella listana, joka sisältää pareja arvoista ja avaimista. Sanasto voidaan luoda esimerkiksi seuraavalla tavalla:

```
sanasto={'arvo1':'avain1', 'arvo2':avain2'}
```

Mikäli halutaan, sanastoa voidaan käyttää joko yksi arvo/avain -pari kerrallaan, tai käydä koko sanasto läpi `for`-lauseessa. Molempia tapauksia havainnollistaa seuraava esimerkki:

```
print sanasto[avain1]
print sanasto[avain2]
for k, v in sanasto.iteritems():
        print k + " on avain  ja " + v + " on arvo."
```

jonka tulosteena on

```
arvo1
arvo2
avain1 on avain ja arvo1 on arvo.
avain2 on avain ja arvo2 on arvo.
```

4 Aliohjelmat ja olio-ohjelmointi

Python on olio-ohjelmointikieli. Luvussa esitellään olioihin liittyviä käsitteitä, kuten luokat, oliot ja perintä, sekä aliohjelmat eli funktiot (eli olioiden metodit). Luvussa käsiteltävät aiheet löytyvät Python Tutorialista [5].

4.1 Funktio

Pythonissa funktion nimet kirjoitetaan usein pienellä alkukirjaimella. **Funktio määritellään seuraavasti**:

```
def tulosta(arg1, arg2):
        print arg1 + arg2
```

Funktiokutsu kirjoitetaan Pythonissa samalla tavalla kuin suurimmassa osassa ohjelmointikieliä. Esimerkiksi edellämääritettyä funktiota voidaan kutsua seuraavasti:

```
tulosta("Picatchu","Bulbasaur")
```

Funktio voi olla Pythonissa erillinen aliohjelma, osa jotain luokkaa tai oliota, tai se voi myös sisältyä toiseen funktioon. Funktiota osana luokkaa ja olioita käsitellään tarkemmin luvuissa 4.2 ja 4.3. Seuraava esimerkki havainnollistaa funktion käyttöä osana toista funktiota:

```
def ulkofunktio():
        def sisafunktio():
                print "Attack!!"
        print "Bulbasaur "
        sisafunktio()
```

Kun funktiota `ulkofunktio()` kutsutaan, näytölle tulostuu

```
Bulbasaur
Attack!!
```

Huomioi edellä tulosteiden järjestys.

4.2 Palautusarvo (Return)

Pythonissa **funktion on mahdollista palauttaa useampi arvo kerrallaan**. Tarkkaan ottaen funktio palauttaa aina vain yhden arvon, mutta pakkaamalla useammat arvot monikoksi, vaikuttaa kuin useampi arvo palautettaisiin. Palautusarvoina voidaan käyttää myös mm. toisia funktioita, listoja ja olioita. Palautus tapahtuu käyttämällä `return`-komentoa. Seuraavassa esimerkissä `funktio1` palauttaa merkkijonon, funktion ja listan:

```
def funktio1():
        return "kissa", fun(), [x for x in range(7)]
```

Kutsuttaessa funktiota puretaan palautettava monikko välittömästi muuttujiin. Tällöin näyttää kuin funktio palauttaisi siis useamman arvon. Seuraava esimerkki havainnollistaa käytännön toteutusta muuttujien ollessa `merkkijono1` , `fun1` ja `lista`:

```
merkkijono1, fun1 , lista = funktio1()
print merkkijono
print lista
```

Tulosteena edellisestä saadaan:

```
kissa
[0,1,2,3,4,5,6]
```

4.3 Luokka (Class)

Python tukee moniperintää. Luokka voidaan periä siis useammasta yläluokasta, jolloin se saa niiden kaikkien ominaisuudet. Luokkaa ei ole pakko periä toisesta luokasta. Usein on jopa suositeltavaa käyttää Pythonin muita ominaisuuksia perinnän tilalla. Konstruktoria suoritettaessa luodaan olio, kuten muissakin ohjelmointikielissä.

Luokka määritellään Pythonissa seuraavalla tavalla:

```
class Luokannimi(Peritty_luokka, Peritty_luokka2):
        """Luokan kommentit"""
        def __init__(self, argu1, argu2, argu3=0):
                """Tämä on konstruktori.
                Yläluokan konstruktoria kutsutaan
                argu3:lla. """
                self.arvo1=argu1
                self.arvo2=argu2
                Peritty_luokka.__init__(argu3)

        def OlionMetodi(self):
                """Tyhjä metodi"""
                pass
```

Olion metodin ensimmäisenä argumenttinä on aina viite olioon itseensä `self`, jonka jälkeen tulevat muut argumentit. Tulee huomata, ettei viitettä itseensä lasketa mukaan parametreihin metodia kutsuttaessa. Oletusarvo määritellään yhtäsuuruusmerkin avulla.

4.4 Oliot

Oliot ovat luokan ilmentymiä. Seuraavassa esimerkissä luodaan luvussa 4.3 määritellystä `Luokannimi`-luokasta `olio1`-niminen olio. `arvo1`-muuttujan arvo on `2`, `arvo2`-muuttujan arvo on `3` ja kolmanneksi parametriksi sijoitetaan arvo `6`. Oliota luotaessa kolmas parametri oli vapaaehtoinen, koska sille on määritetty oletusarvoksi `0` (katso luku 4.3).

Olio luodaan komennolla

```
olio1 = Luokannimi(2,3,6)
```

Oliolla on lisäksi metodi nimeltään `OlionMetodi`(katso luku 4.3) , jota kutsutaan ilman parametrejä, ja joka ei tee mitään. **Olion metodeja voidaan kutsua** seuraavalla tavalla:

```
olio1.OlionMetodi()
```

Olioon on mahdollista lisätä ominaisuuksia lennosta. Tämän ominaisuuden ansiosta on helppo toteuttaa toiminnallisuus erillisiin funktioihin. Kun ohjelma edistyy, niin voidaan lisätä funktioita tarvittaessa yksittäiseen olioon.

Seuraavassa määritetään funktio, sekä liitetään se olioon ja käytetään sitä tulostamisessa:

```
def yleinenfunktio():
        return "Palautetaan vaikka tämä merkkijono."

olio1.fun = yleinenfunktio
print olio1.fun()
```

Tulosteeksi edellisestä lausekkeesta tulee

```
Palautetaan vaikka tämä merkkijono.
```

5 Tiedostot

Komentotulkin hyvistä puolista huolimatta, ohjelmakoodi kirjoitetaan käytännössä lähdekooditiedostoihin. Luvussa esitellään lähdekoodin jakamista tiedostoiksi. Lisäksi esitellään, kuinka tietoa tallennetaan ja kuinka sitä haetaan tiedostoista. Enemmän tietoa tiedostojen käyttömahdollisuuksista löytyy Python Library Referencestä [6].

5.1 Lähdekooditiedostoihin jakaminen

Pythonissa lähdekooditiedostoihin jakaminen ja niiden mukaan liittäminen toimii `import`-käskyn avulla. Liitettävän tiedoston tulee olla tavallinen Python-tiedosto.

Yleinen kirjasto ja itse tehty lähdekooditiedosto voidaan **liittää lähdekoodiin** seuraavasti:

```
import string
import omaluomus
```

Kun tiedosto on liitetty osaksi lähdekoodia, tiedoston sisältämät funktiot ja luokat ovat käytössä. Kaikkeen liitettyyn **viitataan kuin tiedoston olioon**. Jos tiedosto `omaluomus.py` sisältää funktion `toiminnallisuus`, niin funktiota kutsutaan seuraavasti:

```
omaluomus.toiminnallisuus()
```

5.2 Tiedostosta lukeminen ja siihen kirjoittaminen

Tiedoston lukeminen tapahtuu toteuttamalla olio, jolta voidaan kysellä tietoja. Seuraavassa esimerkissä tiedoston sisällöstä tehdään olio nimeltään `lue`. Sitten oliolta pyydetään sen sisältämä tieto, joka tallennetaan listaan nimeltä `sisalto`. Tulostus tapahtuu normaalin listarakenteen tapaan.

```
lue = file(tiedostonimi)
sisalto = lue.readlines()
for x in sisalto:
        print x
```

Tiedostoon kirjoittaminen on toimenpiteenä lähes peilikuva tiedostosta lukemisen kanssa. Siinä siis avataan tiedosto, josta tehdään olio nimeltään `kirjoita`.

Sitten olio kirjoittaa merkkijonon `s` muistiin. Lopuksi oliota pyydetään sulkemaan tiedosto. Seuraavassa sama on esitelty Python-koodina.

```
kirjoita = file(filename,'w')
s = "Kaikki tieto, jonka haluat kirjoittaa tiedostoon."
kirjoita.write(s)
kirjoita.close()
```

Python Library Referencestä [6] löytyy tarkemmin tietoa eri asetuksista tiedostoon kirjoittamiseen, kuten mm. jo olemassaolevaan tiedostoon lisäämisestä.

6 Poikkeukset

Pythonissa on määritelty paljon erilaisia poikkeuksia. Perusperiaate on sama kuin muissakin kielissä. Seuraavassa esimerkissä tehdään tahallaan virhe, joka otetaan **poikkeuskäsittelijässä** kiinni.

```
try:
        print "Yritys"
        100/0
        print "Tänne asti ei päästäkään."
except(ZeroDivisionError):
        print "Epäonnistui"
```

Em. epäonnistuneen yrityksen onnistunut poikkeuskäsittely tulostaa siis seuraavaa:

```
Yritys
Epäonnistui
```

Huomaa, että ohjelmaa suoritetaan aina poikkeuksen tapahtumiseen asti, jolloin hypätään suoraan poikkeuskäsittelijään eli kohtaan `except`. Ohjelmassa tulee aina määritellä, mikä poikkeus otetaan kiinni. Jollei poikkeusta määritetä, kaikki poikkeukset, myös **bugien** virheilmoitukset, otetaan kiinni. Lisätietoa erilaisista poikkeuksista löytyy Python Library Referencestä [6].

7 Generaattorit

Generaattoreiden avulla Pythonissa voidaan toteuttaa rakenteita, joiden käytöstä monissa ohjelmointikielissä vasta unelmoidaan. Generaattoreiden oikea käyttö lyhentää lähdekoodia, yksinkertaistaa lähdekoodin rakennetta ja vähentää siinä olevien laskureiden käyttöä. Lisää tietoa generaattoreista löytyy Python Library Referencestä [6].

7.1 Yield

Yield on hyvä esimerkki Pythonin generaattoreista. `yield`in suora suomennos on *tuottaa*. Yksi tapa käyttää generaattoria on iteraattorina. Funktion suorittaminen voidaan katkaista `yield`in avulla aina tarpeen mukaan, kuten tapahtuu seuraavassa esimerkissä:

```
def foo():
        print "First I like to say.."
        yield 1
        print "Second... "
        yield 2
```

Kutsuttaessa funktiota, se palauttaa **generaattoriolion**, jonka avulla voidaan lähteä suorittamaan funktiota paloissa. Generaattorioliota voidaan kutsua seuraavasti:

```
g = foo()
g.next()
```

Tällöin näytölle `g`:n iterointi tulostuu seuraavasti:

```
First I like to say..
1
```

Seuraava funktion osio voidaan vastaavasti iteroida funktion `g.next()` avulla, jolloin näytölle tulostuu

```
Second...
2
```

Aina on mahdollista joko jatkaa funktion suorittamista seuraavaan palaan, tai luoda kokonaan uusi generaattoriolio. Enemmän esimerkkejä `yield`in käytöstä löytyy lähteestä [4].

8 Kirjastot

Pythonissa on paljon valmiita kirjastoja. Usein ohjelmoitaessa onkin syytä kysyä, löytyykö toteutus jostain olemassaolevasta kirjastosta. Luvussa esitellään malliksi yksi hyvä kirjasto. Lisää malleja kirjastoista ja niiden käyttämisestä löytyy Python Library Referencestä [6].

8.1 Säännölliset lausekkeet (Re)

Esimerkkinä hyvästä kirjastosta esitellään tutkielmassa `re` *(Regular expression)*. Kirjasto sisältää paljon merkkijonojen käsittelyyn liittyviä funktioita.

Seuraava esimerkki havainnollistaa `re`-kirjaston käyttöä:

```
import re
reg = re.compile('A+')
print reg.match("A")
```

Esimerkissä kirjasto ensin liitetään lähdekoodiin, jonka jälkeen luodaan **sääntöolio** `reg` ja tulostetaan sen vastaus kyselyyn [6].

Vastaukseksi kyselyyn, `match`-funktio palauttaa joko olion tai `none`n, riippuen siitä, onnistuiko kysely vai ei. Seuraavassa esimerkissä ensin tulostetaan onnistuneen kyselyn vastaus ja sitten esitetään epäonnistunut kysely:

```
<_sre.SRE_Match object at 0x4020e218>
print reg.match("B")# Tässä "B" ei toteuttanut sääntöä.
None
```

8.2 Muita kirjastoja

Pythonin kirjastoista yleisimmin käytetään seuraavia:

`string`	merkkijonot ja niiden käsittely,
`cmd`	tuki riviorientoituneelle komentotulkille,
`math`	yleinen matematiikkakirjasto,
`sys`	systeemin parametrit ja funktiot,
`random`	satunnaislukukirjasto sekä
`time`	aikakirjasto.

9 Yhteenveto

Tutkielmassa käsiteltiin Pythonin perusteita, kuten tyypillisimpiä tietorakenteita, syntaksia, luokkia ja olioita. Lisäksi tarkasteltiin Pythonin niitä piirteitä, jotka eivät ole aivan arkipäivää jokaisessa ohjelmointikielessä.

Tutkielmassa tutustuttiin lyhyesti esimerkkien kautta generaattoreihin, funktioiden ominaisuuksiin ja olion dynaamiseen kehittämiseen. Tutkielman liitteeksi sijoitetun esimerkkilähdekoodin avulla on yritetty selkeyttää Pythonin toiminnallisuutta käytännönläheisesti. Kaikki tutkielmaan liittyvä lähdekoodi on kirjoitettu ajatellen Python-ohjelmointia aloittelevaa, jolla on kuitenkin ohjelmointikokemusta.

Pythonista löytyy paljon tietoa Internetistä. **Python Tutorial** [5] sisältää kaiken olennaisen ohjelmoinnin aloittamiseksi. **Python Library Reference** [6] sisältää kattavan esityksen suurimmasta osasta Pythonin kirjastoista dokumentaatioineen ja esimerkkeineen. **Numerical Python** [2] sisältää lähinnä työkaluja laskennan toteuttamiseksi Pythonilla. **PyOpenGL** [7] kertoo, kuinka Python voidaan liittää **OpenGL**-grafiikkakirjastoon. Jos Pythonin haluaa tutustua painetussa muodossa, vaihtoehtoina ovat esimerkiksi **Programming Python** [1] ja **Dive Into Python** [3].

Viitteet

[1] Lutz Mark, *Programming Python*, 2nd Edition, O'Reilly & Associates, Inc., Sebastopol, 2001.

[2] Dubois Paul F. et. al, *Numerical Python*, saatavilla HTML-muodossa <URL: `http://www.pfdubois.com/numpy`>, viitattu 10.10.2004, Pfdubois.com.

[3] Pilgram Mark, *Dive Into Python*, Saatavilla PDF-muodossa <URL: `http://diveintopython.org/`>, 20.5.2004, DiveIntoPython.org

[4] Pramode C.E, *Python Generator Tricks*, saatavilla HTML-muodossa <URL: `http://linuxgazette.net/100/pramode.htm`>, maaliskuu 2004, LinuxGazette.

[5] van Rossum Guido, *Python Tutorials*, Release 2.3.4, saatavilla HTML-muodossa <URL: `http://docs.python.org/tut/tut.html`>, 20.5.2004, Python.org.

[6] van Rossum Guido, *Python Library Reference*, Release 2.3.4, saatavilla HTML-muodossa <URL: `http://docs.python.org/lib/lib.htm`>, 20.5.2004, Python.org.

[7] *PyOpenGl*, saatavilla HTML-muodossa <URL:`http://pyopengl.sourceforge.net`>, viitattu 10.10.2004, Sourceforge.net.

A Pokemon.py

```
"""
The example file describes how to use Python programming language.
The file is a part of Bachelor's thesis
of the author and most of its examples are included in the file.


Made by: Tero Tuovinen, 23.11.2004
"""

""" The following librarys are included. Math is a basic mathematical
library and utils is authors own library including usefull
functonalities."""
import math
import utils

class Pokemon:
    """ Pokemon is a basic class of the programm.
    The class contains all the values, the lists and the functions
    that should been found in all pokemon objects

    Parameters: none """
    def __init__(self):
        """This is a constructor.
        Parameters: none
        Returns: none"""
        self.attacks=[]
        self.attackIter=upgradeNormalAttacks()
        self.numberOfAttacks=-1
        self.bag = []

    def save(self,filename,s):
        """The function provides basic subroutines for writing to a
        file. In this context, the function is used to save all pokemons
        data.
        Parameters: filename (string)
        Returns: none """
        saveFile=file(filename, 'w')
        saveFile.write(s)
        saveFile.close()

    def help(self):
        """The function is an example for reading a datafile.
```

```
        The function shows the content of the file help.txt.
        The reading is  protected by try - except structure.
        Parameters: none
        Returns: none
        """
        try:
            readHelpFile=file('help.txt','r')
            data = readHelpFile.readlines()
            for eachline in data:
                print eachline
        except:
            print "The file was not found or the file was empty."

class Pikatchu(Pokemon):
    """The class is an example of an inherited class. The base class
    is Pokemon. """
    def __init__(self, name, hp, level):
        """This is a constructor of Pikatchu. At first, we call
        the baseclass constructor.
        After that, all the arguments are placed in the
        corresponding variables.
        Parameters:
        name(string)
        hp(integer)
        level(integer)
        Returns: none
        """
        Pokemon.__init__(self)
        self.name=name
        self.hp=hp
        self.level=level
        self.power=1
    def attack(self, attack):
        """The function shows us how Pikatchu attacks.
        It demonstrates, how tuples can be used.
        Parameters: attack(integer)
        Returns: none"""
        if (attack >= 0) and (attack <=self.numberOfAttacks):
            currentAttack, attackName = self.attacks[attack]
            print "".join(["Picatchu hits ", attackName,
            " -attack.", " Damage is ", currentAttack(self.power),
            " points."])
```

```
    def combo(self):
        """The function shows us all Pikatchu attacks.
        It demonstrates how the range function can be used."""
        print "Combo"
        for x in range(self.numberOfAttacks+1):
            self.attack(x)

    def save(self):
        """The function creates a correct layout for
        saving data. It demonstrates how empty block (pass)
        can be used.
        Parameters: none
        Returns: none
        """
        save_all_data="".join((self.name ,"\nhp:", str(self.hp),
        "\nLevel:" , str(self.level) ,"\nPower:" , str(self.power)))

        try:
            save_all_data = save_all_data + "\n\nAttacks:\n"
            for x in self.attacks:
                save_all_data=save_all_data + str(x[1]) + "\n"
        except:
            pass
        Pokemon.save(self, self.name + ".dat", save_all_data)


def thunderAttack(power):
    """The function is an implementation of thunderattack.
    Parameters: power(int)
    Returns: string"""
    return str(5.5*int(power))



def upgradeNormalAttacks():
    """The function upgrades the attack resources
    of pokemon. Punch and slash are inline functions,
    and the outer function yields allways next needed function.
    It demonstrates how the inline function, yield, tuple and
    a function in the other file can be used.
    Parameters:  none
    Yields: Generator (used as a pointer)
    Returns: none"""
    def punch(power):
```

```
            """The inline uses the file utils.py and the
            function RandomHit.
            Parameters: power (integer)
            Returns: hitdamage (string)"""
            HitDamage = 1.5* int(power)
            return utils.RandomHit(50, HitDamage)

        def slash(power):
            """The inline uses the file utils.py and the function
            RandomHit.                                                  140
            Parameters: power (integer)
            Returns: hitdamage (string)"""
            HitDamage = 2.0 * int(power)
            return utils.RandomHit(30, HitDamage)
        print "Add Punch"
        yield punch , "Punch"
        print "Add Slash"
        yield slash, "Slash"

def addNewAttack(obj):                                                  150
        """ The function provides a secure way of adding new
        attacks into the object. It demonstrates how the yield generator
        can be used as an object AttackIter. If there are no more
        iterations left, except is raised and handled.
        Parameters: none
        Returns: none"""
        try:
            obj.attacks.append(obj.attackIter.next())
            obj.numberOfAttacks=obj.numberOfAttacks+1
        except(StopIteration):                                          160
            pass

def addSpecialAttack(obj, specialAttack):
        """The function adds a new special attack.
        It demonstrates how a function can be used as a parameter.
        Parameters: SpecialAttack(function)"""
        obj.attacks.append(specialAttack)
        obj.numberOfAttacks=obj.numberOfAttacks+1

def addItems(obj,value):                                                170
        """The function provides a dictonary of all the
        possible items in the program.
        It demonstrates how dict structure can be used.
        Parameters: value(string)
```

```
        Returns: none"""
        Items = {'bi':'bicycle', 'po':'pokedex'}
        obj.bag.append(Items[value])

"The creation of My Pokemon."
MyPoke=Pikatchu("My Picatchu",35,1)

"Teaching My Pokemon to attack."
addNewAttack(MyPoke)

"More Attacks are added."
addNewAttack(MyPoke)

"No more attacks are available."
addNewAttack(MyPoke)

"My Pokemon attacks using attack 1"
MyPoke.attack(1)
"and 0"
MyPoke.attack(0)

"Now I want to teach a speacial attack to My Pokemon"
addSpecialAttack(MyPoke,(thunderAttack, "Thunderattack"))

"Attack by using all the possible attacks."
MyPoke.combo()

"My first Pokedex :)"
addItems(MyPoke, 'po')

"Save all My Pokemon data to a file."
MyPoke.save()
"If I want see the help file..."
MyPoke.help()
```

B Utils.py

```
"""
The file demonstrates howto import your own file.
```

```
Made By: Tero Tuovinen    15.10.2004
"""
"""The library random is needed in the file."""

import random

def RandomHit(percents, HitDamage):
        """ The function decides wheather your hit
        is a succes or a failure.

        Parameters:
        percents (integer) = how good you are [0 - 100]
        HitDamage (integer)= how hard you hit if you succeed.

        Returns:
        How many points is your hitvalue (string)
        """
        if random.random() > percents/100:
                return str(HitDamage)
        return str(0)
```

C My Pokemon.dat

```
My Picatchu
hp:35
Level:1
Power:1

Attacks:
Punch
Slash
Thunderattack
```

D Help.txt

```
This is a helpfile for the pokemon program.
Made By: Tero Tuovinen, 27.10.2004
```