LUT Python ohjelmointiopas 2023 - osa 12 Sisällysluettelo

Luku 12: Tiedon esitysmuodoista

Tietokoneiden kanssa työskennellessä kuulee usein sanottavan, että tietokoneet käsittelevät asioita bitteinä. Tämä väittämä itsessään on tietenkin totta, mutta mitä nämä bitit oikein ovat ja miten niitä käytetään? Asian selvittämiseksi tutustumme tässä luvussa bitteihin ja tiedon erilaisiin esitysmuotoihin tietokoneohjelmissa.

Lähtökohtaisesti ohjelma käsittelee tietoa itselleen optimaalisessa muodossa, jotta ohjelmista saadaan mahdollisimman nopeita ja helppoja ymmärtää. Tämän lisäksi ohjelmat joutuvat käsittelemään tietoa matalammalla tasolla eli kirjoittamaan ja lukemaan tietoa tietokoneen muistiin, jolloin tieto pitää esittää elektroniikan kannalta sopivassa muodossa. Toisaalta esitettäessä tietoa tietokoneen käyttäjälle se muutetaan usein esim. taulukoiksi ja diagrammeiksi, jotta käyttäjän olisi helpompi ymmärtää sitä. Lisämausteensa tietojen esittämiselle tuo erilaiset merkkitaulukot, joita on käytetty erilaisissa tietokoneissa ja eri maissa erilaisten kirjaimien takia eli meillä on mm. skandinaavisia ja kyrillisiä merkkejä. Tutustuimme aiemmin tiedostojen käsittelyyn ja rajoituimme silloin tekstitiedostoihin, mutta useimmat ohjelmat käsittelevät binaaritiedostoja ja tekstitiedostojakin voi todellisuudessa olla hyvin monenlaisia. Näihin asioihin tutustumisen lisäksi palaamme aiemmin nähtyihin desimaalilukujen pyöristysvirheisiin eli miksi Python tulostaa ”virheellisiä” lukuja kuten 15.600000000000001 ja ei aina pyöristä lukuja niin kuin peruskoulun matematiikan opettaja opetti.

Bittejä ja erilaisia kantalukuja

Käsittelemme päivittäin lukuja ja ne ovat tyypillisesti kymmenjärjestelmän lukuja. Lukujärjestelmän idea on yksinkertainen eli meillä on esim. 10 erilaista numeroa ja kun ne on käsitelty, siirrytään seuraavaan isompaan ”kategoriaan” ja jatketaan vastaavalla tavalla. Luku 10 on meille ihmisille tuttu, sillä lähtökohtaisesti meillä on 10 sormea ja 10 varvasta, joiden avulla voi harjoitella 10-lukujärjestelmän toimintaa. Numeerisesti nämä 10 erilaista numeroa ovat 0, 1, 2, 3, 4, 5, 6, 7, 8 ja 9, joilla pystytään siis esittämään 10 erilaista tilannetta alkaen 0:sta ja päättyen 9:ään, jonka jälkeen otetaan seuraava kymmen mukaan ja aloitetaan alusta eli 10, 11, jne.

Tietokoneet perustuvat tyypillisesti 2-kantaisiin lukuihin tai niiden monikertoihin kuten 8 (2^3) tai 16 (2^4). Tämän taustalla on elektroniikka eli 2-kantaisella luvulla voi olla 2 arvoa, jotka ovat 0 ja 1. Nämä on ihmisen helppo ymmärtää elektroniikasta tiloina, jotka vastaavat tilannetta *ei-jännitettä* ja *on-jännite*. Ohjelmointipuolella näitä vastaavat tilat ovat *False* ja *True*, tai 0 ja 1, joten nämä asiat ovat luonnollisia tiloja ohjelmoijalle, kunhan niiden kanssa toimii hetken. Todellisuudessa pelkästään luvuilla 0 ja 1 ei pääse pitkälle, mutta kun niitä on monta, muuttuu tilanne toiseksi ja siksi näiden bittien lisäksi tietokoneissa puhutaan usein sananleveydestä. Sananleveys tarkoittaa prosessorin rakennetta eli montako bittiä prosessori pystyy käsittelemään yhdellä kertaa ja usein tietokoneissa on vastaavan levyiset ulkoiset väylät eli sananleveys määrää sen, montako bittiä kulkee rinnakkain yhtä aikaa tietokoneen sisällä. 0 ja 1 määrittävät siis yhden bitin tilan ja jos sanassa on esim. 8 bittiä, sananleveys on 8 bittiä ja ne mahdollistavat 2^8 eli 256 erilaista kombinaatiota ja siten numeroiden 0-255 esittämisen. Tietokoneiden kehittymisen myötä niihin tuli 8 bitin jälkeen 16, 32 ja 64 bittiä eli sananleveys kasvoi ja tällä hetkellä puhutaan usein salausalgoritmeista, joita on esim. 256, 512 ja 1024 bittisiä. Nämä bitit ja sanaleveydet eivät ole tällä erää ohjelmoijien kannalta normaalisti keskeisiä asioita, sillä tietokoneiden kehittymisen myötä prosessorien tehot ovat saavuttaneet riittävän tason eikä näitä asioita yleensä tarvitse miettiä. Historiasta kiinnostuneet voivat katsoa Wikipediasta esim. Intel 8080 -prosessorin historiaa. Tämä 1974 julkaistu prosessori oli 8 bittinen ja sen voidaan katsoa aloittaneen PC-koneiden aktiivisen kehityksen, sillä sitä seurasi 8086 prosessori, josta tuli Intelin x86 arkkitehtuurin perusta (https://en.wikipedia.org/wiki/Intel_8080).

Käytännössä tietokoneen sananleveys määrittää, kuinka monta bittiä ryhmitellään yhdeksi numeroksi. Näin ollen tietokoneen muistista tuleva jono bittejä eli numeroita 1 ja 0 ryhmitellään sananleveyden perusteella sanoiksi, jotka muutetaan sopivan kantaluvun numeroiksi. Kantaluku määrittää kuinka monta erilaista arvoa numero voi saada ja tässä yhteydessä kiinnostavat kantaluvut ja niiden mahdolliset arvot ovat seuraavat:

- 2 (=2^1), mahdolliset arvot 0 ja 1
- 8 (=2^3), mahdolliset arvot 0, 1, 2, 3, 4, 5, 6 ja 7
- 16 (=2^4), mahdolliset arvot 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e ja f
- 10, mahdolliset arvot 0, 1, 2, 3, 4, 5, 6, 7, 8 ja 9; ei perustu 2-potensseihin

Kantaluvuista ja niiden muunnoksista löytyy tarkempaa tietoa esim. Wikipediasta (https://fi.wikipedia.org/wiki/Bin%C3%A4%C3%A4rij%C3%A4rjestelm%C3%A4)

Binaarilukumuunnokset

Tietokoneen sisällä kaikki tieto käsitellään ja tallennetaan bitteinä, eli kaksikantalukuina, jotka saavat arvoja 0 ja 1. Ryhmittelemällä nämä bitit isompiin kokonaisuuksiin voimme muuttaa binaariluvut kokonaisluvuiksi jatkokäsittelyä varten. Kuten edellä oli puhetta, bitit voidaan ryhmitellä sananleveyden eli tietokoneen ja sen prosessorin perusteella. Toinen vaihtoehto on ryhmitellä bitit tietotekniikassa käytettävän datan mittayksikön *tavun* perusteella (https://fi.wikipedia.org/wiki/Tavu_(tietotekniikka)). Käytännössä tavu sisältää 8 bittiä ja on siten tietokoneesta sekä prosessorista riippumaton yksikkö eli sopiva yleiseen käyttöön. Näin ollen 8 bittiä eli yksi tavu muistia on 2-kantainen esitysmuoto numeroarvolle väliltä 0-255.

Bitit ja tavut yms. kuulostaa hienolta, mutta katsotaan esimerkin 12.1 avulla, kuinka bittijonoista eli binaariluvuista saadaan meille tuttuja 10-kantaisia kokonaislukuja. Ohjelma pyytää käyttäjää syöttämään bittijonon (eli vapaamuotoisen jonon nollia ja ykkösiä) ja tämän jälkeen käy läpi jonon laskien siitä samalla kymmenlukuarvon. Koska bittiarvoissa merkitsevin (lukuarvoltaan suurin) bitti tulee vasemmanpuoleisimmaksi, käydään bittijono läpi oikealta vasemmalle.

Esimerkki 12.1. Bittilukujen laskeminen, binaariluvusta kokonaisluvuksi

```
def bittiluku():
    Bittijono = input("Anna binaariluku: ")

    Tulos = 0
    Pituus = len(Bittijono)
    Bittijono = Bittijono[::-1] # Bittijonoa luetaan lopusta alkuun päin

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

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

    print("Bittijonosi on 10-kantaisena", Tulos)
    return None

bittiluku()
```

Tuloste

```
Anna binääriluku: 1101
Bittijonosi on 4 bittiä pitkä.
Bittijonosi on 10-kantaisena 13
```

Bittijonossa lukuarvot edustavat kahden potensseja. Jos bittijono on esimerkiksi 5 bittiä pitkä, on siinä silloin bitteinä ilmaistuna lukuarvot 2^4, 2^3, 2^2, 2^1 ja 2^0. Nämä toisen potenssit vastaavat numeroarvoja 16, 8, 4, 2 ja 1. Käytännössä biteillä ilmaistaan, lasketaanko kyseinen bitti mukaan vai ei: jos bitti saa arvon 1, lasketaan sitä vastaava toisen potenssi lukuarvoon, jos taas arvon 0, lukua ei lasketa mukaan. Jos tarkastelemme esimerkiksi binaarilukua 1101, laskettaisiin se seuraavasti:

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

Jos vastaavasti laskisimme 10-kantaisen esityksen bittiarvolle 101101, saisimme seuraavan tuloksen:

Bittiluku	**1**	**0**	**1**	**1**	**0**	**1**
2:n potenssi	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Tulos	**1***32 + **0***16 + **1***8 + **1***4 + **0***2 + **1***1 = 32 + 8 + 4 + 1 = **45**					

Luonnollisesti voimme tehdä myös ohjelman, joka muuntaa kokonaisluvun binaarilukumuotoon ja sellainen on esimerkissä 12.2. Ohjelma pyytää käyttäjää syöttämään kokonaisluvun ja selvittää ensin kahden potenssin, joka on suurempi kuin annettu kokonaisluku. Tämän jälkeen ohjelma testaa, voiko luvusta vähentää kahden sen hetkisen potenssin. Jos vähennys on mahdollista eli erotus > 0, merkitään bittijonoon 1 ja lasketaan erotus. Jos taas ei, merkitään bittijonoon 0 ja siirrytään pienempään potenssiin.

Esimerkki 12.2. Bittilukujen laskeminen, kokonaisluvusta binaariluvuksi

```
def laskeBinaari(Luku):
    Potenssi = 0
    Jono = ""
    while ((2 ** Potenssi) <= Luku): # Esitykseen tarvittava bittien määrä
        Potenssi = Potenssi + 1

    while (Potenssi > -1):
        if ((Luku - 2**Potenssi) < 0): #Jos arvo liian suuri, merkitään 0
            Jono = Jono + "0"
        else:
            Jono = Jono + "1"   # Bittiarvo voidaan vähentää, merkataan 1
            Luku = Luku - 2 ** Potenssi
        Potenssi = Potenssi - 1 # Lähestytään arvoa 0 joka kierroksella
    return Jono

def paaohjelma():
    Lukuarvo = int(input("Anna kokonaisluku: "))
    Tulos = laskeBinaari(Lukuarvo)
    print("Antamasi 10-kantainen luku on 2-kantaisena", Tulos)

paaohjelma()
```

Tuloste

```
Anna kokonaisluku: 74
Antamasi 10-kantainen luku on 2-kantaisena 01001010
```

Merkkitaulukot

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

Puhuimme aiemmin, että kiintolevylle bittijonot tallennetaan kahdeksan bitin joukkoina eli tavuina. 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. Käytännön esimerkki tästä logiikasta on normaali vuosikymmeniä vanhaa ASCII-taulukko, jolla voimme havainnollistaa menetelmää. Ja lisätietoja löytyy Internetistä, esim. http://fi.wikipedia.org/wiki/ASCII.

Taulukko 12.1 ASCII-taulukko

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

Yllä olevaan taulukkoon 12.1 on kerätty alkuperäisen ASCII-taulukon numerot ja niitä vastaavat merkit. Ensimmäiset 32 merkkiä (0-31) on varattu ohjauskoodeille ja muille ei-näkyville merkeille. Jos luemme levyltä bittisarjan 001000001, tarkoittaa se kokonaislukuna arvoa 65. Jos taas tulkitsemme tämän numeroarvon ASCII-taulukon avulla, huomaamme lukeneemme tiedostosta ison A-kirjaimen. Jos tiedostossa vastaavasti lukisi ”01100001 01110101 01010100 01101111”, voitaisiin se lukea arvoina ”92 117 84 111”, eli ”auTo”.

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

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

Windows XP ja Pythonin versiot ennen versiota 3 tuottivat välillä ongelmia UTF-8-merkistökoodauksen kanssa ja tästä syytä oli usein varmempi käyttää merkkitaulukkoa 1252 ja ohjelma alkoi alla olevalla määrityksellä. Nykyisissä versioissa on toimiva UTF-8 koodaus eikä alla olevan tyylisiä määrityksiä enää tarvita. Tähän voi törmätä lähinnä vanhoja ohjelmia ylläpitäessä eli itse tätä ei nykyään tarvitse käyttää.

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

Tässä vaiheessa merkki ja merkistö -teemaa herää varmaan epäilys, että tuloste riippuu tulostettavan arvon lisäksi siitä, miten arvo tulostetaan. Esimerkissä 12.3 näkyy, miten ASCII-taulukon yksittäisen merkin paikan taulukossa voi selvittää `ord`-funktiolla ja toisaalta ASCII-taulukon merkin voi tulostaa `chr`-funktiolla antamalla halutun merkin paikan taulukossa. Tai siis tämä toimii käytettävän merkistön osalta, sillä ASCII-taulukossa ei ole esimerkin euro-merkkiä eikä alkiota 54353. Huomaa myös esimerkissä näkyvä lukujen 48-57 tulostus useilla eri asetuksilla eli kokonaislukuna, oktaalilukuna, heksadesimaalilukuna ja ASCII-taulukosta näitä lukuja vastaavat merkit eli luvut 0-9. Eli sama tieto tulostuu nyt 4 eri tavalla asetuksista riippuen, joten asetusten kanssa kannattaa olla tarkkana.

Esimerkki 12.3. Tulosteet, merkkitaulukot ja asetukset

```
print("Merkin järjestysnumero merkkitaulukossa:")
print(ord("a"))
print(ord("€"))
print()

print("Järjestysnumeroa vastaava merkki merkkitaulukossa:")
print(chr(97))
print(chr(54353))
print()

print("int-octa-hexa-ascii taulukon arvo")
for i in range(48, 58):
    print("{0:d} - {1:o} - {2:x} - {3:s}".format(i, i, i, chr(i)))
```

Tuloste

```
Merkin järjestysnumero merkkitaulukossa:
97
8364

Järjestysnumeroa vastaava merkki merkkitaulukossa:
a
퓝

int-octa-hexa-ascii taulukon arvo
48 - 60 - 30 - 0
49 - 61 - 31 - 1
50 - 62 - 32 - 2
51 - 63 - 33 - 3
52 - 64 - 34 - 4
53 - 65 - 35 - 5
54 - 66 - 36 - 6
55 - 67 - 37 - 7
56 - 70 - 38 - 8
57 - 71 - 39 - 9
```

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

Merkkijonojen lajittelusta

Merkkijonojen käsittelyn yhteydessä totesimme, että niitä voi vertailla >, < ja == merkeillä samalla tavalla kuin lukuja. Tämän lisäksi listan yhteydessä totesimme, että `sort()`-jäsenfunktio lajittelee merkkijonot vastaavalla tavalla kuin numerot. Tähän mennessä olemme aina todenneet, että toimii, mutta miten ja miksi toimii, on jäänyt käsittelemättä. Mutta nyt kun olemme käyneet läpi ASCII-taulukon periaatteet, asia rupeaa selkiämään, sillä merkkijonojen lajittelu perustuu lajiteltavien merkkien paikkaan ASCII-taulukossa, tai yleisemmin käytetyssä merkistössä. Näin ollen esim. a lajitellaan A:n jälkeen, koska a:n paikka ASCII-taulukossa on 97 ja A:n paikka on 65. Tiivistetysti merkkien vertailu muuttuu numeroiden vertailuksi ja asia selkiytyy merkittävästi, etenkin jos muistaa luvussa 10 käsitellyt asiat kohdassa *Huomioita sarjallisten muuttujien vertailusta*. Yleisenä varoituksena kannattaa taas nostaa erikseen skandinaaviset merkit, jotka ovat ASCII-taulukon laajennetussa osassa ja siten ASCII-taulukon mukaan järjestetyt ääkkösiä sisältävät merkkijonot eivät todennäköisesti ole oikeassa järjestyksessä.

Esimerkki 12.4 demonstroi listassa olevien merkkijonojen lajittelua.

Esimerkki 12.4. Merkkijonojen lajittelu ja ASCII-taulukko

```
Nimet = ["aab", "aba", "baa", "Aaa", "aAa", "aaA"]
Nimet = ["aab", "aba", "baa", "Aaa", "aAa", "aaA", "AAa"]
Nimet = ["aab", "aba", "baa", "Aaa", "aAa", "aaA", "Öaa"]

for Nimi in Nimet:
   print(Nimi)
print()

Nimet.sort()
for Nimi in Nimet:
   print(Nimi)
```

Tuloste

```
aab
aba
baa
Aaa
aAa
aaA
Öaa

Aaa
aAa
aaA
aab
aba
baa
Öaa
```

Binaaritiedostot eli sananen pickle-moduulista

Tiedostonkäsittelyn yhteydessä keskityimme tekstitiedostoihin ja totesimme, että myös binaaritiedostoja on olemassa. Käytännössä kaikki kaupalliset ohjelmat tallentavat tiedot binaarimuodossa, tai tietokantaan, ja tekstimuodot ovat pääasiassa yhteensopivuuden takia tarjolla lisäominaisuutena. Tyypillisesti binaaritiedostot ovat pienempiä ja niissä on enemmän tietoa nopeammin käsiteltävässä muodossa, joten binaaritiedostojen käyttö on usein luonnollinen valinta. Tässä oppaassa tavoitteena on tähän asti ollut ymmärtää tiedostojen käytön periaatteet ja nyt kun tutustumme binaaritiedostoihin huomaamme, että tyypillisesti tiedoston tallennus ja luku redusoituu yhteen tallenna/lue -käskyyn, joka hoitaa kaiken siitä eteenpäin. Näin ollen binaaritiedostojen käyttö on tyypillisesti varsin helppoa, mutta toisaalta niiden kohdalla tulee ymmärtää, että tiedostomuodon muuttaminen saattaa tehdä aiempien tiedostojen lukemisen ja käytön mahdottomaksi eli tiedostomuodon muutokset pitää tehdä hallittuna prosessina. Tekstitiedostoja voi katsella oikeilla koodieditoreilla, mutta siitä ei ole tyypillisesti juurikaan hyötyä, vaikka tallennettujen tietojen muoto olisi tiedossa. Kuten binaariluku-osiossa edellä todettiin, tiedot voi jakaa tavun kokoisiin lohkoihin ja muuttaa esim. kokonaisluvuiksi, mutta esim. C-kielessä ohjelmoija voi valita kokonaisluvun käytössä olevan tavumäärän ja se ei näy tiedoston bittikuvioita katsoessa.

Pythonissa on `pickle`-moduuli, jolla voi tallentaa tietoa binaarimuodossa. Käytännössä `pickle`:llä voi tallentaa ja lukea esimerkiksi kokonaisia luokkia, listoja ja sanakirjoja yksinkertaisesti funktioilla `dump` ja `load`. Esimerkissä 12.5 näkyy binaaritiedoston tallennus. Koska `pickle`-moduuli tallentaa tiedon binaarimuotoisena, näyttää käytetyn tiedoston sisältö hyvinkin sotkuiselta ja riippuen erilaisista tekijöistä tiedoston sisältö saattaa näyttää erilaiselle. Suurimmat erot koodissa on tiedoston avaaminen kirjoitettavaksi binaarimoodissa eli ”`wb`”, ja tallennus yhdellä `pickle.dump` – käskyllä.

Esimerkki 12.5. pickle-moduuli, binaaritallennus

```
import pickle
import sys

try:
    Tiedosto = open("L12E5.bin","wb")
    Lista = ["Turtana","Viikinkilaiva",{"Joe-poika":"Papukaija"},
                 327000884764897.5]

    pickle.dump(Lista, Tiedosto)
    Tiedosto.close()
except OSError:
    print("Tiedoston käsittelyssä virhe, lopetetaan.")
    sys.exit(0)
```

Tiedosto L12E5.bin

```
€
•C       ]”(ŒTurtana”Œ
Viikinkilaiva”}”Œ Joe-poika”Œ Papukaija”sGBò–}9>e.
```

Binaaritiedostoa ei kannata muokata itse, koska tyypillisesti muutokset tekevät tiedostosta käyttökelvottoman. Huomaa, että emme määritelleet merkistökoodausta tiedostoa avattaessa. Tähän on looginen selitys eli emme kirjoittaneet tiedostoon merkkejä vaan tavuja, joten merkistökoodausta ei tarvita. Tiedoston sisältö ei ole ihmisen luettavissa, eikä sen ole tarkoituskaan olla. Tietokone pystyy kuitenkin käsittelemään sitä tehokkaasti.

Esimerkissä 12.6 näkyy vastaavasti binaaritiedoston luku `pickle`:n `load`-funktiolla. Tärkein ero on taas tiedostoa avattaessa eli luemme binaaritiedoston moodilla ”`rb`” ja luku tapahtuu yhdellä `load`-käskyllä. Tietoja tulostettaessa joudumme nyt katsomaan esimerkistä 12.5 tietorakenteen muodon eli että listassa oli 4 tietoalkioita, joita voidaan käyttää normaalisti.

Esimerkki 12.6. Binaaritiedoston luku Pythonin pickle-moduulilla

```
import pickle
import sys

try:
    Tiedosto = open("L12E5.bin","rb")
    Luettu = pickle.load(Tiedosto)
    Tiedosto.close()
except OSError:
    print("Tiedoston käsittelyssä virhe, lopetetaan.")
    sys.exit(0)

print(Luettu)
print(Luettu[1])
print(Luettu[2])
```

Tuloste

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

Kuten näimme, `pickle`-moduulilla voi tallentaa ja ladata kehittyneempiä tietorakenteita muuttamatta niitä ensin merkkijonoiksi. `pickle`-moduuli osaa käsitellä kaikkia tässä oppaassa esiteltyjä tietomuotoja. Kannattaa muistaa, että `pickle`:ä käyttäessä tiedosto pitää avata binaaritilassa avauskytkimillä ”`rb`” (read-binary) ja ”`wb`” (write-binary).

Pari sanaa pyöristämisestä

Olemme käyttäneet tässä oppaassa Pythonia jo monenlaiseen asiaan ja olet varmaan joskus ihmetellyt Pythonin tulostamia numeroita sekä niiden tarkkuutta. Kaksi keskeistä ongelmaa näihin liittyen ovat ensinnäkin numeroiden esitystarkkuus ja tallennusmuodosta johtuvat epätarkkuudet sekä toisaalta Pythonin pyöristyssäännöt. Esimerkissä 12.7 näkyy esimerkit näistä molemmista ongelmista.

Esimerkki 12.7. Pythonin liukulukujen tarkkuus ja pyöristykset

```
print("Liukulukujen tallennusmuodosta johtuvia epätarkkuuksia, esim. IEEE-754:")
print(3 * 5.2)
print()

print("Pythonin pyörityssäännöt poikkeavat suomalaisista säännöistä:")
print("Tulostetaan '2.5' pyöristettynä kokonaisluvuksi:", round(2.5))
print("Tulostetaan '3.5' pyöristettynä kokonaisluvuksi:", round(3.5))
```

Tuloste

```
Liukulukujen tallennusmuodosta johtuvia epätarkkuuksia, esim. IEEE-754:
15.600000000000001

Pythonin pyörityssäännöt poikkeavat suomalaisista säännöistä:
Tulostetaan '2.5' pyöristettynä kokonaisluvuksi: 2
Tulostetaan '3.5' pyöristettynä kokonaisluvuksi: 4
```

Esimerkissä 12.7 tulostetaan ensin tulon ’3*5.2’ tulos, joka on 15.600000000000001 eli noin 15.6, muttei tasan 15.6. Lähtökohtana on, että tietokoneen muistiin ei mahdu numeroita äärettömällä tarkkuudella, vaan desimaaliluvut on määritelty tietyillä standardeilla (yleensä IEEE-754, The IEEE Standard for Floating-Point Arithmetic) ja niitä käsitellään sovittujen sääntöjen mukaan. Tästä löytyy tarkempaa tietoa esimerkiksi Wikipediasta hakusanalla ”liukuluku” ja englanninkielinen artikkeli on suomenkielistä laajempi. Käytännössä etenkin laskettujen desimaalilukujen osalta kannattaa olettaa, että niiden viimeisissä desimaaleissa on eroja ja siten yhtäsuuruusvertailuja ei kannata tehdä. Tulosteet kannattaa yleensä myös pyöristää tulokset järkevälle tarkkuudelle, tyypillisesti 1 tai muutama desimaali.

Toinen esimerkissä 12.7 näkyvä outo asia on desimaalilukujen pyöristäminen, sillä meille suomalaisille on itsestään selvää, että vitonen pyöristyy ylöspäin. Tämä on selkeä toimintapa, mutta tuo mukanaan ongelmia. Ajatellaanpa seuraavaa: Kahvilassa on myynnissä pulla hintaan 2,5€ ja kakkupala hintaan 3,5€. Erittäin vanha kirjanpitojärjestelmä osaa laskea vain kokonaisluvuilla, joten se pyöristää aina numerot kokonaisluvuksi. Näin ollen, kun eräänä päivänä saadaan myytyä 10 pullaa ja 10 kakkupalaa, on kirjapidossa tulona 70€ mutta kassassa on vain 60€ rahaa. Tämä ei kuulosta hyvältä.

Pythonissa lukujen pyöristämiseen liittyvä kumuloitumisongelma on ratkaistu pyöristämällä vitonen aina parillista numeroa kohti esimerkin 12.7 mukaisesti, jolloin pyöristysvirhe ei pääse kumuloitumaan. Tällainen pyöristäminen saattaa kuulostaa hassulta suomalaiseen korvaan, mutta kannattaa huomata, että maailmassa on peruskoululaitoksia, joissa tällaista pyöristystä opetetaan ensimmäisestä luokasta lähtien, joten heille meidän suomalaisten käyttämä pyöristys tuntuu varmasti hyvin epäloogiselle. Tietokoneiden käsitellessä numeroita parilliseen numeroon suuntaava pyöristys on kätevä vaihtoehto, koska tällöin pyöristyksestä johtuva virhe minimoituu. Pyöristystavoistakin löytyy lisää tietoa esimerkiksi Wikipediasta hakusanalla ”pyöristäminen”.

Yhteenveto

Osaamistavoitteet

Tämän luvun keskeiset asiat on nimetty alla olevassa listassa. Tarkista sen avulla, että tunnistat nämä asiat ja sinulla on käsitys siitä, mitä ne tarkoittavat. Mikäli joku käsite tuntuu oudolta, käy siihen liittyvät asiat uudestaan läpi. Nämä käsitteet ja asiat on hyvä muistaa ja tunnistaa periaatteellisella tasolla eli kun törmäät näihin asioihin ohjelmointitehtävissä, voit palata ja perehtyä näihin asioihin tarvittavalla tarkkuudella.

- Ymmärrät tiedon tallentamisen bitteinä eli ykkösinä ja nollina sekä bittien muuntamisen tavujen kautta numeroiksi (E12.1, 12.2)
- Ymmärrät merkkitaulukoiden käytön sekä ASCII-taulukon periaatteet (Taulukko 12.1, E12.3)
- Ymmärrät merkkijonojen lajittelun periaatteet (E12.4)
- Ymmärrät binaaritiedostojen käsittelyn `pickle`-moduulin avulla (E12.5, 12.6)
- Ymmärrät desimaali- eli liukulukujen ja pyöristämisen haasteita ohjelmoinnissa (E12.7)