Ohjelmoinnin perusteet Pythonilla

Päivitetty 18.1.2012

Yleistä

- Materiaali sisältää lähinnä Ohjelmoinnin perusteet Y -kurssin harjoitustehtävissä tarvittavia keskeisiä asioita
- Vastaavat tiedot löytyvät huomattavasti laajemmin kurssin opetusmonisteesta

Ohjelmoimaan oppii vain ohjelmoimalla ja oivaltamalla!

Näytölle tulostaminen

- Komennolla **`print`** tulostetaan näytölle

```
print "Ohjelmointi on hauskaa!"
```

```
luku1 = 3
luku2 = 5
print "Valitut luvut ovat", luku1, "ja", luku2
```

```
luku1 = 3
luku2 = 5
print "Lukujen summa on", luku1 + luku2
```

Näppäimistön lukeminen

- Komennolla **`raw_input`** pyydetään käyttäjältä tietoja

```
nimi = raw_input("Kerro nimesi\n")
print "Hei", nimi
```

- Vastaus palautetaan aina merkkijonona, vaikka käyttäjä antaisikin luvun

(`\n` **`raw_input`**-käskyn lopussa tarkoittaa rivinvaihtoa, joka tarvitaan Goblin-järjestelmää varten)

Laskutoimitukset

- Muuttujaan voi tallettaa laskutoimituksen tuloksen

`+`	yhteenlasku
`-`	vähennyslasku
`*`	kertolasku
`/`	jakolasku
`**`	potenssiin korotus
`%`	jakojäännös

```
tulos = 2 + 3
```

```
tulos = 2 * (3 + 5)
```

```
tulos = (luku1 + luku2) * 3
```

```
tulos = tulos + 2
```

```
tulos += 2
```

Tyyppimuunnokset

- Tietokone voi käsitellä tietoa vain, jos se on oikeassa muodossa
- **`raw_input`** palauttaa käyttäjän vastauksen aina merkkijonona

```
luku = raw_input("Anna luku, kerron sen kahdella\n")
tulos = float(luku) * 2
print "Tulos on", tulos
```

Pythonin tyyppejä: int, float, bool, str

Tyyppimuunnokset

- Muunnokseen on monta eri tapaa

```
rivi = raw_input("Anna 1. luku\n")
luku1 = float(rivi)
rivi = raw_input("Anna 2. luku\n")
luku2 = float(rivi)
tulo = luku1 * luku2
```

```
luku1 = raw_input("Anna 1. luku\n")
luku1 = float(luku1)
luku2 = raw_input("Anna 2. luku\n")
luku2 = float(luku2)
tulo = luku1 * luku2
```

```
luku1 = float(raw_input("Anna 1. luku\n"))
luku2 = float(raw_input("Anna 2. luku\n"))
tulo = luku1 * luku2
```

```
luku1 = raw_input("Anna 1. luku\n")
luku2 = raw_input("Anna 2. luku\n")
tulo = float(luku1) * float(luku2)
```

Vakiot

- Vakio on ohjelmassa kiinteästi määritelty arvo, jolle on annettu sitä kuvaava nimi eikä sen arvoa muuteta
- Vakiot tekevät koodista helpommin ymmärrettävää
- Sen sijaan, että kirjoittaisi lukuarvon useaan kohtaan koodia, voi käyttää pelkkää vakion nimeä
- Vakioiden nimet kirjoitetaan isoilla kirjaimilla

```
KILOHINTA = 5.40
PAKKAUSKULUT = 3.20


paino = float(raw_input("Anna paino kilogrammoina:\n"))
print "Tuote maksaa", paino * KILOHINTA + PAKKAUSKULUT, "euroa."
```

Ehtolauseet

- `if`-ehtolauseilla voidaan ohjata ohjelman toimintaa erilaisten ehtojen avulla
- Ehtolauserakenteita on erilaisia eri käyttötarkoituksia varten

```
if luku >= 0:
  print "Luku on nolla tai suurempi"
```

```
if luku >= 0:
  print "Luku on nolla tai suurempi"
else:
  print "Luku on nollaa pienempi"
```

Ehtolauseet

```
if luku > 0:
  print "Luku on nollaa suurempi"
elif luku < 0:
  print "Luku on nollaa pienempi"
else:
  print "Luku on nolla"
```

- Vain ensimmäinen ehdot täyttävä osio suoritetaan, loput jätetään huomiotta
- Ehtolauseessa voi olla (tai olla olematta) rajaton määrä **`elif`**-osia ja yksi **`else`**-osa

Ehtolauseet

- Ehdoissa voi hyödyntää vertailuoperaattoreita

`==`	yhtäsuuri
`!=`	erisuuri
`>`	suurempi kuin
`<`	pienempi kuin
`>=`	suurempi tai yhtä suuri kuin
`<=`	pienempi tai yhtä suuri kuin

- Ehdon voi kääntää sanalla **`not`**

```
if not luku > 9:
```

- Ehtoja voi yhdistellä sanoilla **`and`** ja **`or`**

```
if luku1 > 4 and luku2 < 3:
```

```
if (luku1 > 4) or (luku2 < 3 and luku2 >= 1):
```

```
if luku > 2 and luku <= 8:
```

```
if 2 < luku <= 8:
```

Tarvittaessa täytyy käyttää sulkeita lausekkeiden ryhmittelemiseen.

`while`-silmukka

- **`while`**-silmukan avulla voidaan toistaa koodia niin kauan kuin jatkamisehto on voimassa

```
kierros = 0
while kierros < 5:
  print kierros
  kierros += 1
print "Silmukka suoritettiin", kierros, "kertaa"
```

- Jatkamisehto tarkistetaan jokaisen kierroksen alussa, myös ensimmäisen
- Silmukkaa ei siis välttämättä suoriteta kertaakaan

`while`-silmukka

- Jatkamisehto määritellään samalla tavalla kuin `if`-lauseessa
- Jos jatkamisehto ei ole voimassa, ohjelman suorittaminen jatkuu seuraavasta sisentämättömästä rivistä silmukan jälkeen

```
kierros = 0
while kierros != 8:
  print kierros
  kierros += 1
print "Silmukka suoritettiin", kierros, "kertaa"
```

while-silmukka

- Suorituskertojen määrää ei tarvitse tietää välttämättä ennakkoon

Laske miljoonaa pienemmät kahden potenssit:

```
luku = 1
while luku < 1000000:
  print luku
  luku *= 2
```

- Silmukan suorittaminen päättyy, kun luku tulee liian suureksi eikä jatkamisehto ole enää voimassa

while-silmukka

- **while**-silmukan avulla voidaan pyytää käyttäjältä esimerkiksi lukuja, joiden määrää ei tiedetä ennakkoon

  Laske positiivisten lukujen määrä, lopeta nollalla:

```
laskuri = 0
luku = int(raw_input("Anna ensimmainen luku\n"))
while luku != 0:
  if luku > 0:
    laskuri += 1
  luku = int(raw_input("Anna seuraava luku\n"))
print "Annoit", laskuri, "positiivista lukua"
```

- Ensimmäinen luku pyydetään jo ennen silmukkaa
- Seuraava luku pyydetään kierroksen lopussa ja jatkamisehto tarkistetaan seuraavan kierroksen alussa

`while`-silmukka

Yleisimmän käyttötavan muistilista:

```
luku = 0


while luku < 10:

  ...

  luku += 1
```

Jatkamisehdossa olevan muuttujan pitää olla olemassa jo ennen ensimmäistä kierrosta

Varmista, että jatkamisehto muuttuu jossakin vaiheessa epätodeksi. Yleensä kuitenkin aikaisintaan vasta ensimmäisen kierroksen jälkeen.

Muista päivittää jatkamisehtoon liittyvän muuttujan arvoa

for-silmukka

- **for**-silmukan avulla voidaan toistaa koodia, jos toistojen määrä tiedetään heti silmukan alussa

  Toista silmukka viisi kertaa:

```
for luku in range(5):
  print "Hei maailma!"
```

- **for**-silmukkaan kuuluu aina muuttuja, jonka arvo päivittyy automaattisesti (edellisessä esimerkissä tätä muuttujaa ei vain tarvittu mihinkään)
- **for**-silmukan avulla voidaan käydä läpi myös luvut ennalta määritellyltä väliltä
- Toistojen määrä tai lukuväli voi olla kiinteästi koodissa tai arvot voidaan lukea muuttujista

`for`-silmukka

- Askellus määritellään **`range`**-funktiolla
- Lukuja läpikäyvät **`for`**-silmukat voi korvata aina myös **`while`**-silmukalla

Toista silmukka viisi kertaa / käy läpi luvut 0, 1, 2, 3, 4:

```
for luku in range(5):
    print luku
```

```
range(toistoja)
```

```
luku = 0
while luku < 5:
    print luku
    luku += 1
```

Käy läpi luvut 13-28 yksitellen (13, 14, 15, ..., 26, 27, 28):

```
for luku in range(13, 29):
    print luku
```

```
range(alaraja, yläraja+1)
```

```
luku = 13
while luku < 29:
    print luku
    luku += 1
```

for-silmukka

Käy läpi joka toinen luku välillä 12-20 (12, 14, 16, 18, 20):

```
for luku in range(12, 21, 2):
    print luku
```

```
range(alaraja, yläraja+1, askel)
```

```
luku = 12
while luku < 21:
    print luku
    luku += 2
```

Käy läpi luvut 33-50 takaperin (50, 49, 48, ..., 35, 34, 33):

```
for luku in range(50, 32, -1):
    print luku
```

```
range(yläraja, alaraja-1, -askel)
```

```
luku = 50
while luku > 32:
    print luku
    luku -= 1
```

- Huomaa, ettei **for**-silmukka saavuta koskaan viimeistä parametrina annettua lukua!

Tulostuksen muotoilu

- Tähän asti on tulostettu muuttujien arvoja erottelemalla tulostettavat osat pilkuilla

```
luku1 = 2
luku2 = 3
print "Ensimmainen luku on", luku1, "ja toinen luku on", luku2
```

- Muotoilukoodien avulla muuttujien paikat voidaan kirjoittaa suoraan tekstin sekaan

```
luku1 = 2
luku2 = 3
print "Ensimmainen luku on %d ja toinen luku on %d" % (luku1, luku2)
```

Käytä näistä vain toista, älä sekoita niitä samaan tulostuskäskyyn.

Tulostuksen muotoilu

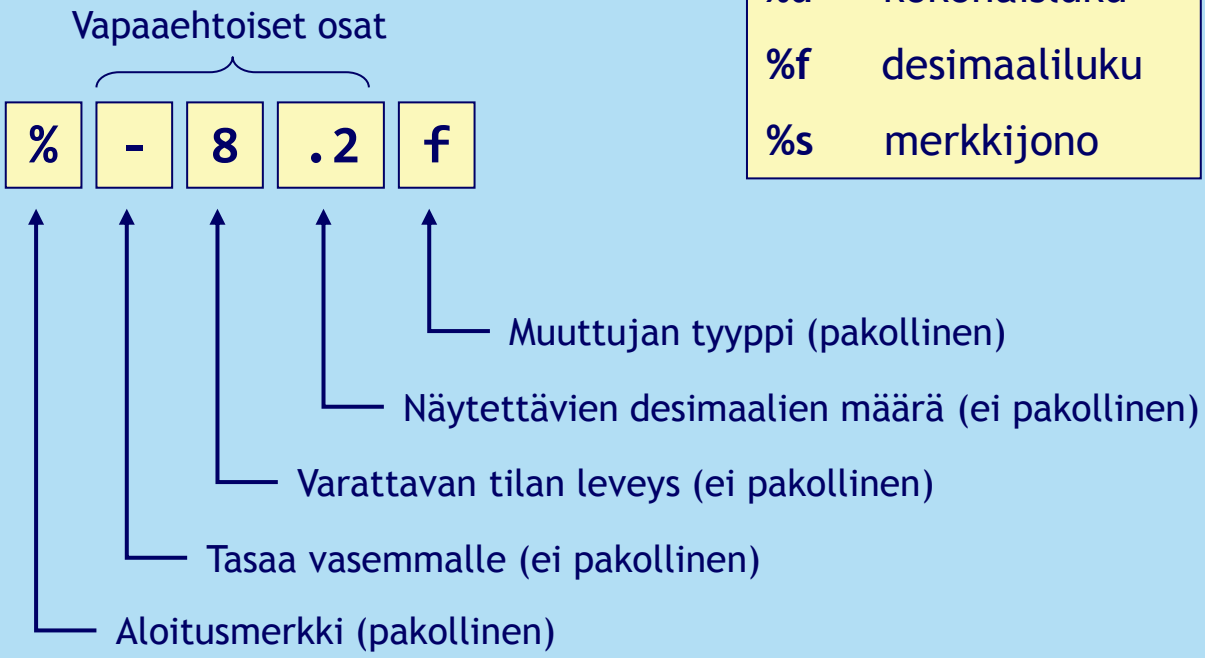
```
luku1 = 2
luku2 = 3
print "Ensimmainen luku on %d ja toinen luku on %d" % (luku1, luku2)
```

- Tekstin sekaan liitettävän muuttujan paikka merkitään prosenttimerkillä alkavalla muotoilukoodilla
- Rivin lopussa kerrotaan prosenttimerkin jälkeen ne muuttujat tai lausekkeet, joiden arvot sijoitetaan muotoilukoodien tilalle

Muotoilukoodit

- Muotoilukoodilla voi vaikuttaa tulostettavan muuttujan muotoiluun

%d	kokonaisluku
%f	desimaaliluku
%s	merkkijono



Muotoiluesimerkkejä

Koodi:

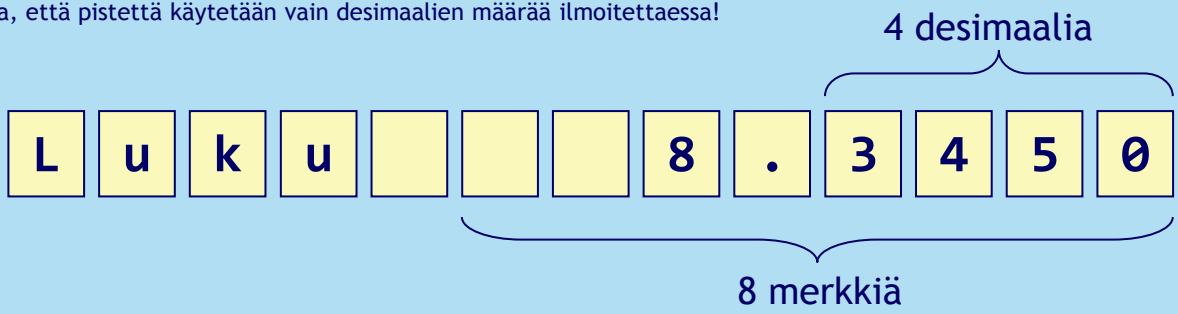
```
luku1 = 2
luku2 = 4.5643567

print "Luku 1 on %d" % luku1
print "Luku 2 on %f" % luku2
print "Luku 2 on noin %.2f" % luku2
print "Luvut ovat %d ja %.2f" % (luku1, luku2)
```

Tuloste:

```
Luku 1 on 2
Luku 2 on 4.5643567
Luku 2 on noin 4.56
Luvut ovat 2 ja 4.56
```

```
luku = 8.345
print "Luku %8.4f" % luku
```

Huomaa, että pistettä käytetään vain desimaalien määrää ilmoitettaessa!



Ohjelman rakenne

- Ohjelma voi koostua useista eri funktioista, joita kutsutaan ohjelman sisältä
- Yleensä ohjelmissa on vähintään yksi funktio: pääohjelma eli **`main`**

```
def main():
  ...


main()
```

- Sisennysten kanssa pitää olla tarkkana!

Funktiot

- Funktiot ovat ohjelman sisällä olevia pieniä osia, jotka suorittavat jonkin tehtävän
- Funktio määritellään sanalla **`def`** ja sitä kutsutaan funktion nimellä

Funktion määrittely ja kutsuminen:

```
def sano_hei():
  print "Hei maailma!"


sano_hei()
```

Muista sulkeet funktion nimen perään!

Funktiot

- Funktiolla voi olla nolla, yksi tai useita parametreja, joita voi käsitellä funktiossa muuttujien tavoin
- Funktiosta on mahdollista palauttaa yksi tai useita arvoja **`return`**-käskyllä
- Funktiossa voi olla useita **`return`**-käskyjä, mutta niistä kuitenkin suoritetaan vain yksi, minkä jälkeen funktiosta poistutaan välittömästi

Erilaisia funktioita

Ei parametreja eikä paluuarvoa:

```
def sano_hei():
  print "Hei maailma!"

sano_hei()
```

Pelkkä paluuarvo:

```
def kysy_nimi():
  return raw_input("Kuka olet?\n")

nimi = kysy_nimi()
```

Yksi parametri:

```
def sano_jotakin(teksti):
  print teksti

sano_jotakin("Tulosta tama!")
```

Useampi parametri:

```
def laske_summa(luku1, luku2):
  print "Summa on", luku1 + luku2

laske_summa(3, 6)
```

Parametreja ja yksi paluuarvo:

```
def laske_tulo(luku1, luku2):
  return luku1 * luku2

tulo = laske_tulo(3, 5)
```

Parametreja ja useampi paluuarvo:

```
def laske_summa_tulo(luku1, luku2):
  return luku1 + luku2, luku1 * luku2

summa, tulo = laske_summa_tulo(2, 7)
```

Funktioiden käyttäminen

- Funktiota voi kutsua ohjelmasta useita kertoja eri parametreilla
- Jos funktio tarvitsee tietoja muualta ohjelmasta, arvot pitää yleensä välittää parametreina funktion sisälle
- Funktioiden sisällä voi olla rajaton määrä normaaleita koodirivejä ja uusia funktiokutsuja

Funktioiden käyttäminen

- Funktion paluuarvon voi tallettaa muuttujaan, tai sitä voi käyttää suoraan, jos samaa arvoa ei tarvita muualla myöhemmin

Tulosta tarvitaan monessa kohdassa:

```
def kerro_kolmella(luku):
  return luku * 3

tulos = kerro_kolmella(5)
if tulos < 12:
  print "Tulos on alle 12"
elif tulos < 16:
  print "Tulos on alle 16"
elif tulos < 24:
  ...
```

Tulosta tarvitaan vain kerran:

```
def kerro_kolmella(luku):
  return luku * 3

print "Vastaus on", kerro_kolmella(5)
```

```
def kerro_kolmella(luku):
  return luku * 3

tulos = kerro_kolmella(5)
print "Vastaus on", tulos
```

Funktioesimerkki

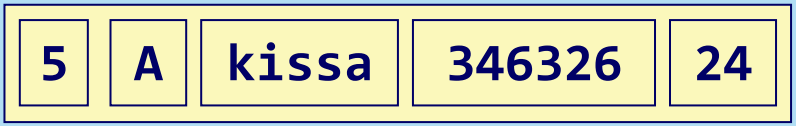
Onko luku parillinen?

```
def onko_parillinen(luku):
  if luku % 2 == 0:
    return True
  else:
    return False

def main():
  luku = int(raw_input("Anna jokin kokonaisluku\n"))
  if onko_parillinen(luku):
    print "Antamasi luku on parillinen"
  else:
    print "Antamasi luku on pariton"

main()
```

Lista

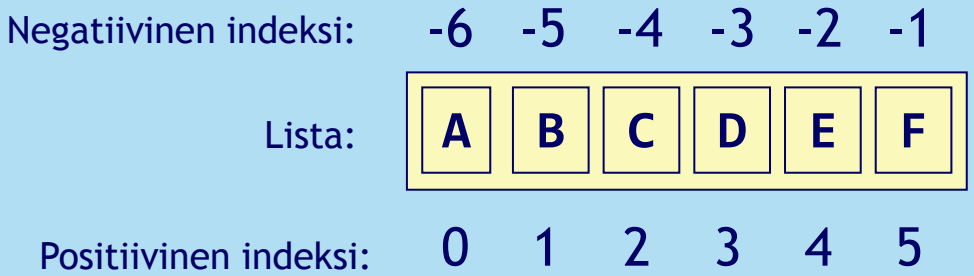
- Lista on tietorakenne, johon voidaan tallettaa tietoa käsiteltäväksi
- Esimerkki: Kansio, johon voidaan lisätä ja ottaa pois papereita, sekä lisäksi papereita voidaan laskea, selailla ja järjestellä
- Listaan talletetaan tietoa alkioina, jotka voivat olla erityyppisiä

Lista, jonka sisällä on viisi alkiota:



Listan indeksointi

- Jokaisella listan alkiolla on oma indeksi, jonka avulla alkioon voi viitata



- Listasta voidaan valita myös useampia alkioita kerralla

```
lista[0] = A      lista[1:3] = [B, C]
lista[3] = D      lista[:3] = [A, B, C]
lista[-5] = B     lista[2:] = [C, D, E, F]
```

Listan toiminnot

Luo tyhjä lista:

```
elaintarha = []
```

Luo uusi lista ja tietty määrä alkioita:

```
elaintarha = ["papukaija"] * 8
```

Lisää uusi alkio tiettyyn kohtaan:

```
elaintarha.insert(3, "kirahvi")
```

Onko alkio listassa:

```
if "undulaatti" in elaintarha:
  ...
```

Hae alkion paikka eli indeksi listassa:

```
nro = elaintarha.index("kenguru")
```

Käännä listan järjestys:

```
elaintarha.reverse()
```

Muuta alkiota:

```
elaintarha[5] = "seepra"
```

Luo uusi lista ja lisää siihen alkioita:

```
elaintarha = ["karhu", "leijona"]
```

Lisää uusi alkio listan loppuun:

```
elaintarha.append("kameli")
```

Poista alkio listasta:

```
elaintarha.remove("leijona")
```

Toistuvien alkioiden määrä listassa:

```
kpl = elaintarha.count("tiikeri")
```

Lajittele lista:

```
elaintarha.sort()
```

Alkioiden kokonaismäärä:

```
elaimia = len(elaintarha)
```

Hae tietty alkio:

```
elain = elaintarha[4]
```

Listan läpikäynti

- Listassa olevat alkiot voi käydä yksitellen läpi **`for`**-silmukan avulla

```
elaintarha = ["karhu", "leijona", "kameli"]
for elain in elaintarha:
  print elain
```

- **`for`**-silmukan muuttujassa on vuorotellen jokainen listan alkio alkaen listan alusta
- Listan alkioiden määrää ei saa muuttaa tällaisen silmukan sisällä!

Listan läpikäynti

- Listassa olevat alkiot voi käydä läpi **`for`**-silmukalla myös indeksien avulla

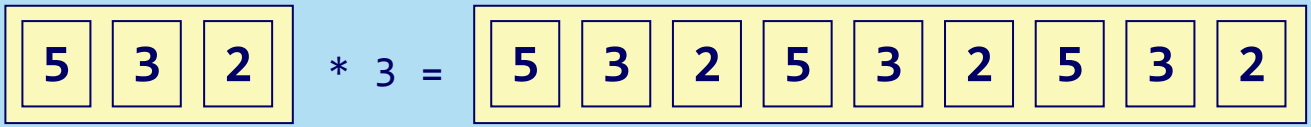
```
elaintarha = ["karhu", "leijona", "kameli"]
for indeksi in range(len(elaintarha)):
  print elaintarha[indeksi]
```

- Tästä on hyötyä, jos alkion indeksinumeroa tarvitaan silmukan sisällä

Listan toistaminen

- Listasta voi luoda kopion *-operaattorilla, jolloin luodaan uusi lista, joka sisältää halutun määrän kopioita alkuperäisestä listasta

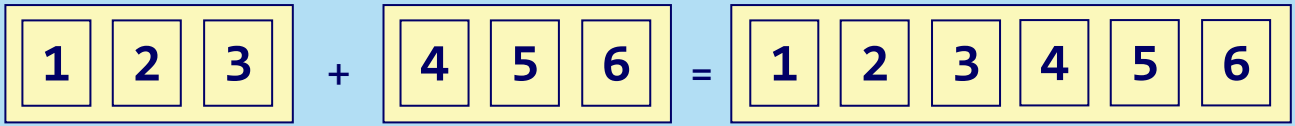
```
numerolista = [5, 3, 2]
toistettu_lista = numerolista * 3
```



Listojen yhdistäminen

- Kaksi tai useampia listoja voidaan liittää yhdeksi uudeksi listaksi +-operaattorilla

```
numerot1 = [1, 2, 3]
numerot2 = [4, 5, 6]
numerot = numerot1 + numerot2
```



Merkkijono listana

- Merkkijonon voi kuvitella eräänlaiseksi listaksi, joka koostuu yksittäisistä merkeistä
- Merkkijono toimii listan tavoin, kun halutaan käydä merkkijono läpi tai viitata merkkeihin
- Komentoja, jotka alkavat listan nimellä ja pisteellä, ei voi käyttää! (esim. `lista.insert` tai `lista.reverse`)
- Merkkijonon merkkejä ei voi muokata!
- Merkkijonossa alkiot ovat merkkejä: vaikka merkki olisi numero, pitää se tyyppimuuntaa luvuksi tarvittaessa!

Merkkijono listana

```
sana = "Python"
print "Sanassa on", len(sana), "kirjainta"
if "y" in sana:
  print "Sanassa on kirjain y"
```

```
sana = raw_input("Anna jokin sana\n")
print "Sanan kirjaimet ovat:"
for kirjain in sana:
  print kirjain
```

```
sana = "Ohjelmointi"
print sana[5]   # m
print sana[-2]  # t
print sana[5:]  # mointi
print sana[1:3] # hj
print sana[:3]  # Ohj
```

Laske positiivisen kokonaisluvun numerot yhteen:

```
luku = raw_input("Anna kokonaisluku\n")
if int(luku) >= 0:
  summa = 0
  for merkki in luku:
    summa += int(merkki)
  print "Luvun numeroiden summa on", summa
```

Merkkijonojen yhdistäminen

- Merkkijonoja voi toistaa ja yhdistellä samalla tavalla kuin listoja

```
alku = "Ohjelmointi "
loppu = "on kivaa!"
lause = alku + loppu
```

```
sana = "hei"
print sana * 2   # Tulostaa heihei
```

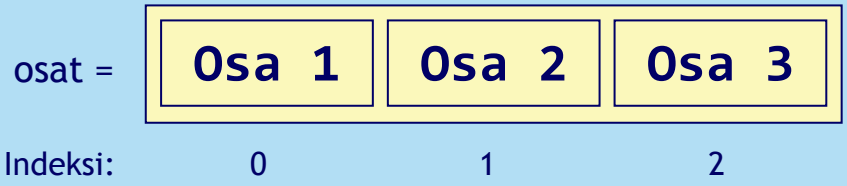
- Yhdisteltävien osien pitää olla merkkijonoja, tarvittaessa tehdään tyyppimuunnos

```
alku = "Ulkona on "
lampotila = 22
loppu = " astetta."
lause = alku + str(lampotila) + loppu
```

Merkkijonon pilkkominen

- Merkkijonosta voi tehdä **`split`**-käskyllä uuden listan, jonka alkioina ovat merkkijonon tietyllä merkillä erotetut osat

```
tiedot = "Osa 1/Osa 2/Osa 3"
osat = tiedot.split("/")
print osat[2]    # Tulostaa Osa 3
```



Laske lukujen summa:

```
lasku = "2+15+3+8"
osat = lasku.split("+")
summa = 0
for osa in osat:
    summa += int(osa)
print summa
```

- Alkiot ovat merkkijonoja sisällöstä riippumatta!

Kirjainkoon vaihtaminen

- Merkkijonon kaikki kirjaimet voi nopeasti vaihtaa isoiksi tai pieniksi kirjaimiksi

Vaihda kirjaimet isoiksi:

```
mjono = "Muuta isoiksi kirjaimiksi"
print mjono.upper()
```

Vaihda kirjaimet pieniksi:

```
mjono = "Muuta pieniksi kirjaimiksi"
print mjono.lower()
```

- Komennot eivät muuta alkuperäistä merkkijonoa, vaan palauttavat uuden muunnetun merkkijonon, joka pitää sijoittaa jonnekin tai käyttää heti!
- Toiminnot ovat käteviä, jos halutaan vertailla merkkijonoja vain kirjaimien, ei kirjaimien koon perusteella (not case-sensitive)

Vertaile merkkijonoja vain kirjaimien perusteella:

```
mjono1 = raw_input("Anna 1. merkkijono\n")
mjono2 = raw_input("Anna 2. merkkijono\n")
if mjono1.upper() == mjono2.upper():
  print "Merkkijonot ovat samat!"
```

Kirjaimien vaihtaminen

- Merkkijonosta voi vaihtaa kaikki tietyt kirjaimet tai merkkijonon osat toisiksi

Vaihda j-kirjaimet k-kirjaimiksi:

```
mjono = "joulu"
uusi = mjono.replace("j", "k")
```

- Komento muuttaa kaikki merkkijonosta löytyvät esiintymät, ei vain ensimmäistä.
- Korvaava osa voi olla myös tyhjä merkkijono, jolloin esiintymät poistetaan. Parametrit voivat olla myös useamman merkin mittaisia.
- Komento ei muuta alkuperäistä merkkijonoa, vaan palauttaa uuden muunnetun merkkijonon, joka pitää sijoittaa jonnekin tai käyttää heti!

Sanakirja

- Sanakirja on tietorakenne, joka sisältää avain-arvo -pareja
- Sanakirjasta saadaan haettua nopeasti avainta vastaava arvo
- Arvoihin voi viitata vain avaimen avulla
- Avain voi olla esimerkiksi luku tai merkkijono, arvo voi olla mitä hyvänsä tyyppiä
- Yksi avain voi olla sanakirjassa vain kerran

Sanakirjan toiminnot

Luo tyhjä sanakirja:

```
varasto = {}
```

Avaimien kokonaismäärä:

```
avaimia = len(varasto)
```

Luo sanakirja ja lisää siihen avain-arvo -pareja:

```
varasto = {"avain" : "arvo", "tuoleja" : 12}
```

Lisää uusi avain-arvo -pari tai muokkaa vanhaa arvoa:

```
varasto["avain"] = "arvo"
```

```
varasto["poytia"] = 7
```

Onko avain sanakirjassa:

```
if "tuoleja" in varasto:
  ...
```

Poista avain (ja sen arvo) sanakirjasta:

```
del varasto["poytia"]
```

Hae avainta vastaava arvo:

```
maara = varasto["tuoleja"]
```

Näissä esimerkeissä on käytetty vain merkkijonoja ja kokonaislukuja. Avaimet voivat olla myös lukuja ja arvot mitä hyvänsä tyyppiä.

Arvojen paikalle voi sijoittaa normaaliin tapaan myös muuttujan.

Sanakirjan läpikäynti

- Sanakirjan avaimet voi käydä yksitellen läpi **`for`**-silmukan avulla

```
for avain in sanakirja:
  print avain
```

- **`for`**-silmukan muuttujassa on vuorotellen jokainen sanakirjan avain
- Avaimilla ei ole järjestystä, joten läpikäynti-järjestyksestä ei pidä olettaa mitään!

Sanakirjan läpikäynti

- Myös sanakirjan avain-arvo -parit voi käydä **`for`**-silmukalla helposti läpi

```
for avain, arvo in sanakirja.iteritems():
  print "Avainta", avain, "vastaa arvo", arvo
```

- Avainta vastaavaa arvoa ei tarvitse nyt hakea erikseen sanakirjasta kuten tässä:

```
for avain in sanakirja:
  print "Avainta", avain, "vastaa arvo", sanakirja[avain]
```

Tiedostot

- Tiedosto avataan **`open`**-komennolla

```
tiedosto = open("kirje.txt", "r")
```

`r`	vain luku
`w`	kirjoitus
`a`	kirjoita loppuun

Tila, johon tiedosto avataan

Tiedoston nimi

- Tiedosto suljetaan **`close`**-komennolla, kun tiedostoa ei enää käytetä

```
tiedosto.close()
```

Tiedoston lukeminen

- Tiedoston voi lukea rivi kerrallaan **`for`**-silmukalla

```
tiedosto = open("kirje.txt", "r")
for rivi in tiedosto:
  rivi = rivi.rstrip()
  print rivi
tiedosto.close()
```

- Jokainen rivi päättyy rivinvaihtoon, jonka **`rstrip`**-komento poistaa
- Rivit ovat merkkijonoja sisällöstä riippumatta!

Tiedostoon kirjoittaminen

- Tiedostoon voi kirjoittaa merkkijonoja **`write`**-komennolla

```
tiedosto = open("malli.txt", "w")
tiedosto.write("Tallennetaan jotakin")
tiedosto.close()
```

- Komennon parametrin tulee olla merkkijono, tarvittaessa käytetään tyyppimuunnosta

Poikkeukset

- Ohjelma ei saa kaatua, jos käyttäjä antaa esimerkiksi kirjaimia ja ohjelma odottaa lukua
- Tätä varten voidaan määritellä poikkeuksia (eli virhetilanteita) käsittelevä rakenne
- Poikkeukset käsitellään **`try`**-rakenteilla, jotka sisältävät yhden **`try`**-osan ja vähintään yhden **`except`**-osan

```
try:
  luku = int("abc")
except ValueError:
  print "Nyt tapahtui virhe!"
```

Poikkeukset

- **`try`**-osaa suoritetaan rivi kerrallaan normaalisti
- Jos ja vain jos tapahtuu virhe, hypätään virhettä vastaavaan **`except`**-osaan
- **`try`**-osaan ei palata enää takaisin, mikäli sieltä on poistuttu

(Huomaa kuitenkin, että jos `try`-rakenne on silmukan sisällä, suoritetaan rakenne seuraavalla kierroksella uudelleen alusta normaalisti)

```
try:
  print "Seuraavalla rivilla tapahtuu virhe"
  luku = int("abc")
  print "Tata riviä ei tulosteta koskaan"
except ValueError:
  print "Nyt tapahtui virhe!"
print "Ohjelman suoritus jatkuu tasta"
```

Poikkeukset

- **except**-osia voi olla useita eri virhetyyppejä varten ja **except**-osassa voi olla myös useita virhetyyppejä

```
try:
  tiedosto = open("virhe.txt", "r")
  lista = [int("abc")] * 5
  lista[8] = 4
  tiedosto.close()
except (ValueError, IOError):
  print "Tyyppimuunnos- tai tiedostovirhe!"
except IndexError:
  print "Virheellinen indeksi!"
```

- **try**-rakenteita voi sijoittaa myös sisäkkäin, tällöin virhe käsitellään vain sisimmässä mahdollisessa **except**-osassa, eikä sitä havaita ulommissa **try**-rakenteissa

Poikkeukset

- Funktiokutsun aikana tapahtuva virhe voidaan käsitellä myös siellä, mistä funktiota kutsutaan.

```
def kysy_luku():
  luku = int(raw_input("Anna kokonaisluku:\n"))
  return luku

def main():
  try:
    luku = kysy_luku()
    print luku * 5
  except ValueError:
    print "Muunnosvirhe, ohjelma paattyy!"

main()
```

Tätä voi hyödyntää näppärästi, jos ohjelma halutaan lopettaa funktiossa tapahtuneen virheen seurauksena.

Tyypillisiä poikkeuksia

- **`ValueError`**: tyyppimuunnos merkkijonosta luvuksi ei onnistunut
- **`IOError`**: avattavaa tiedostoa ei löydy tai sen käsittelemisessä tapahtui virhe
- **`IndexError`**: indeksi ei ole sallitulla välillä
- **`KeyError`**: avainta ei ole sanakirjassa
- **`ZeroDivisionError`**: koodissa on yritetty jakaa nollalla

Oliot

- Oliot ovat keino kuvata ja käsitellä ohjelman sisällä erilaisia yksinkertaistettuja malleja, jotka liittyvät usein todellisiin asioihin
  - Esimerkki: Olio voisi kuvata esimerkiksi pankkitiliä, autoa, opiskelijaa, tai koordinaatistoon asetettua viivaa
- Olion sisällä on tallennettuna kaikki kyseiseen olioon liittyvä tieto, jota voidaan lukea ja muokata olion toimintojen avulla
  - Esimerkki: Pankkitiliä kuvaavassa oliossa voisi olla tallennettuna tilin saldo, jonka voi lukea ja muuttaa

Oliot

- Ohjelmaan kirjoitetaan luokka, joka kuvaa olion piirteet
- Oliot ovat ilmentymiä luokasta, eli yhdestä luokasta voidaan luoda rajaton määrä itsenäisiä olioita ohjelmaan
- Esimerkki: Oliot ovat kuin robotteja. Ne toimivat kaikki täsmälleen samalla ennalta määritellyllä tavalla, mutta ne toimivat täysin toisistaan riippumattomasti ja niillä on kaikilla oma muisti.

Oliot

- Oliota kuvaavassa luokassa määritellään olion kentät ja metodit
- Kentät vastaavat muuttujia, mutta niiden sisältö on oliokohtainen
- Metodit vastaavat funktioita, mutta ne suoritetaan aina tietyn olion sisällä

Luokan määrittely

- Luokka määritellään avainsanalla **`class`** tiedoston uloimmalla tasolla
- Metodit määritellään luokan sisälle avainsanalla **`def`**

```
class Tuote:
  def __init__(self, nimi, hinta):
    self.__nimi = nimi
    self.__hinta = hinta

  def hae_hinta(self):
    return self.__hinta
```

Kentät

- Kentät ovat tavallisia muuttujia, jotka voivat sisältää numeroita, merkkijonoja, listoja jne...
- Kaikilla samantyyppisillä olioilla on samannimiset kentät, mutta niiden sisältö on oliokohtainen
- Kentät luodaan _ _init_ _-metodissa
- Kenttiin pitää viitata luokan sisällä aina sanan **`self`** avulla

```
def korota_hintaa(self, korotus):
  self._hinta += korotus
```

- Yleensä kentän nimi aloitetaan kahdella alaviivalla, jotta kentän arvoa ei voi muokata suoraan muualta kuin tästä luokasta

Metodit

- Metodit toimivat lähes täysin samalla tavalla kuin funktiot
- Metodin määrittelyssä on vain aina oltava ensimmäisenä parametrina **`self`**, joka viittaa siihen olioon, jossa metodia suoritetaan

```
class Tuote:
  def __init__(self, nimi, hinta):
    self.__nimi = nimi
    self.__hinta = hinta

  def hae_hinta(self):
    return self.__hinta
```

Metodit

- Pakollisen **`self`**-parametrin avulla voi käsitellä olion kenttiä
- Kaikki metodit (myös _ _init_ _ ja _ _str_ _) saavat sisältää täysin vapaan määrän normaaleita koodirivejä

```
class Tuote:
  def __init__(self, nimi, hinta):
    self.__nimi = nimi
    self.__hinta = hinta

  def hae_hinta(self):
    return self.__hinta

  def aseta_hinta(self, hinta):
    self.__hinta = hinta
```

_ _init_ _

- Jokaisessa luokassa on yleensä metodi _ _init_ _, joka suoritetaan automaattisesti uutta oliota luotaessa (kaksi alaviivaa peräkkäin sanan molemmin puolin)
- Tässä metodissa määritellään kaikki luotavan olion kentät ja asetetaan niille alkuarvot

```
class Tuote:
  def __init__(self, nimi, hinta):
    self.__nimi = nimi
    self.__hinta = hinta
```

- _ _init_ _-metodiin voi lisätä **`self`**-parametrin jälkeen omia parametreja, jotka annetaan oliota luotaessa

Olion luominen

- Olio luodaan sijoittamalla viittaus siihen esimerkiksi muuttujaan, listaan tai sanakirjaan
- Olio luodaan antamalla luokan nimi ja suluissa _ _init_ _-metodissa määritellyt parametrit

```
tuote = Tuote("Karkkipussi", 2.20)
```

- HUOM! **`self`**-parametri jätetään tässä kohdassa kokonaan huomiotta, aivan kuten sitä ei olisi metodin määrittelyssä lainkaan
- Jos luokka on määritelty eri tiedostossa kuin sitä käyttävä ohjelma, lue myös **`import`**-käskystä sivulta 68

Metodien kutsuminen

- Olion metodeita kutsutaan metodin nimellä kuten funktioitakin, mutta sitä ennen tulee viittaus olioon ja piste

- Olioon voi viitata esimerkiksi muuttujan avulla

```
karkkipussi = Tuote("Karkkipussi", 2.20)
hinta = karkkipussi.hae_hinta()
print "Tuote maksaa %.2f euroa" % hinta
karkkipussi.aseta_hinta(hinta * 1.2)
```

- HUOM! **`self`**-parametri jätetään tässäkin kokonaan huomiotta, aivan kuten sitä ei olisi metodin määrittelyssä lainkaan (Parametri on itse asiassa jo ennen metodin nimeä)

Metodien kutsuminen

- Luokassa voidaan määritellä myös, että metodin sisällä kutsutaan saman tai jonkin toisen olion metodia

```
tuote1 = Tuote("Halpa", 2.00)
tuote2 = Tuote("Halvempi", 1.50)
if tuote1.onkoHalvempiKuin(tuote2):
  print "Tuote 1 on halvempi"
else:
  print "Tuote 1 ei ole halvempi"
```

```
class Tuote:

  ...

  def haeHinta(self):
    return self.__hinta

  def onkoHalvempiKuin(self, toinen):
    omahinta = self.haeHinta()
    toisenhinta = toinen.haeHinta()
    if omahinta < toisenhinta:
      return True
    else:
      return False
```

_ _str_ _

- Luokkaan voi määritellä myös _ _str_ _-metodin, joka suoritetaan automaattisesti, kun olio halutaan tulostaa **`print`**-käskyllä
- Metodi palauttaa jonkin merkkijonon

```
class Tuote:
  def __init__(self, nimi, hinta):
    self.__nimi = nimi
    self.__hinta = hinta

  def __str__(self):
    return "%s maksaa %.2f euroa." % (self.__nimi, self.__hinta)
```

```
kirja = Tuote("Kirja", 13.60)
print kirja   # Kirja maksaa 13.60 euroa.
```

`import`-käsky

- Luokka voidaan määritellä omaan tiedostoon ja sitä käyttävä ohjelma toiseen tiedostoon
- Tällöin luokka pitää tuoda ohjelman käyttöön **`import`**-käskyllä ohjelman alussa ensimmäisellä rivillä
- Tähän on kaksi tapaa:

```
import tuote
def main():
  lehti = tuote.Tuote("Lehti", 4.50)
```

Tässä luokka Tuote on määritelty tiedostossa tuote.py ja olio luodaan aina komentamalla tuote.Tuote(...)

```
from tuote import *
def main():
  lehti = Tuote("Lehti", 4.50)
```

Tässä luokka Tuote on määritelty myös tiedostossa tuote.py, mutta olion voi luoda suoraan luokan nimen perusteella

Opetusmonisteen lisäksi osoitteessa
`http://docs.python.org/` on paljon hyödyllistä materiaalia