# Ohjelmoinnin perusteet Pythonilla

© Teemu Sirkiä, 2023

Päivitetty 3.9.2023

# Yleistä

- Materiaali sisältää lähinnä Aalto-yliopiston Ohjelmoinnin peruskurssi Y1:n harjoitustehtävissä tarvittavia keskeisiä asioita
- Vastaavat tiedot löytyvät huomattavasti laajemmin kurssin oppimateriaalista

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)
```

# Laskutoimitukset

- Käytössä on tavalliset laskutoimitukset

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

Katkaisevan jakolaskun lopputulos on jaettavien lukujen tyypistä riippuen kokonaisluku tai desimaaliluku, jonka desimaaliosa on aina nolla.

```
10 / 4 = 2.5
10 // 4 = 2
10.0 // 4 = 2.0
```

```
tulos = 2 + 3
```

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

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

```
tulos = tulos + 2
```

```
tulos += 2
```

# Näppäimistön lukeminen

- Komennolla **input** voi pyytää käyttäjää antamaan tietoja ohjelmalle

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

- Vastaus palautetaan aina merkkijonona, vaikka käyttäjä antaisikin luvun

Merkkijono `\n` `input`-käskyn lopussa tarkoittaa rivinvaihtoa. Näin kysymys ja vastaus menevät eri riveille.

# Tyyppimuunnokset

- **input**-käsky palauttaa käyttäjän syötteen aina merkkijonona, joten ennen kuin käyttäjältä saatuja tietoja voi käyttää laskutoimituksissa, tulee tehdä tyyppimuunnos merkkijonosta luvuksi

```
luku = input("Anna luku, kerron sen kahdella\n")
tulos = float(luku) * 2
print("Luku kaksinkertaisena on", tulos)
```

**float** tarkoittaa desimaalilukua, **int** kokonaislukua

# Tyyppimuunnokset

- Muunnokseen on monta erilaista tapaa:

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

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

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

Tyyppimuunnoksen voi yhdistää kätevästi myös **input**-käskyyn, jolloin se ei unohdu:

```
luku1 = float(input("Anna 1. luku\n"))
luku2 = float(input("Anna 2. luku\n"))
tulo = luku1 * 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 saman lukuarvon useaan kohtaan koodia, voi käyttää vakion nimeä
- Vakioiden nimet kirjoitetaan isoilla kirjaimilla

```
KILOHINTA = 5.4
PAKKAUSKULUT = 3.2


paino = float(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 käyttötarkoituksiin

```
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(input("Anna ensimmainen luku\n"))
while luku != 0:
  if luku > 0:
    laskuri += 1
  luku = int(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)
```

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

`range(toistoja)`

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

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

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

`range(alaraja, yläraja+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("Luvut ovat", luku1, "ja", luku2)
```

- Muotoilukoodien avulla muuttujien paikat voidaan kirjoittaa suoraan tekstin sekaan

```
luku1 = 2
luku2 = 3
print(f"Luvut ovat {luku1} ja {luku2}")
```

Käytä näistä tavoista vain toista, älä sekoita niitä samaan tulostuskäskyyn.

# Tulostuksen muotoilu

```
luku1 = 2
luku2 = 3.2567
print(f"Luvut ovat {luku1} ja {luku2:.2f}")
```

- Muotoilua sisältävän merkkijonon alussa ennen sitaattimerkkiä tulee olla kirjain `f` osoittamassa, että merkkijono muotoillaan. Sitä ei pidä sotkea itse muotoilukoodissa olevaan `f`-kirjaimeen, joka osoittaa desimaaliluvun muotoilua.

- Tekstin sekaan liitettävän muuttujan paikka merkitään muotoilukoodin sisältävillä aaltosulkeilla. Aaltosulkeisiin kirjoitetaan tulostettavan muuttujan nimi sekä kaksoispisteen jälkeen tarvittaessa tieto, miten tuloste halutaan muotoilla

# Muotoilukoodit

- Muotoilukoodilla voi vaikuttaa tulostettavan muuttujan muotoiluun

Valinnaiset muotoilumääreet

```
{ nimi : < 8 .2 f }
```

- `{` Aloitusmerkki
- `<` Tasaa luku vasemmalle
- `8` Varattavan tilan leveys
- `.2` Näytettävien desimaalien määrä
- `f` Tyyppikirjain
- `}` Lopetusmerkki

| | |
|---|---|
| d | kokonaisluku |
| f | desimaaliluku |
| s | merkkijono |

| | |
|---|---|
| `<` | tasaa vasemmalle |
| `>` | tasaa oikealle |
| ^ | keskitä |

Mikäli arvoa ei halua muotoilla mitenkään, voi aaltosulkeiden sisällä käyttää pelkkää muuttujan nimeä, jonka haluaa tulostaa.

# Muotoiluesimerkkejä

Koodi:

```
luku1 = 2
luku2 = 4.5643567

print(f"Luku 1 on {luku1:d}")
print(f"Luku 1 on {luku1}")
print(f"Luku 2 on noin {luku2:.2f}")
print(f"{luku1} ja {luku2:.2f}")
```

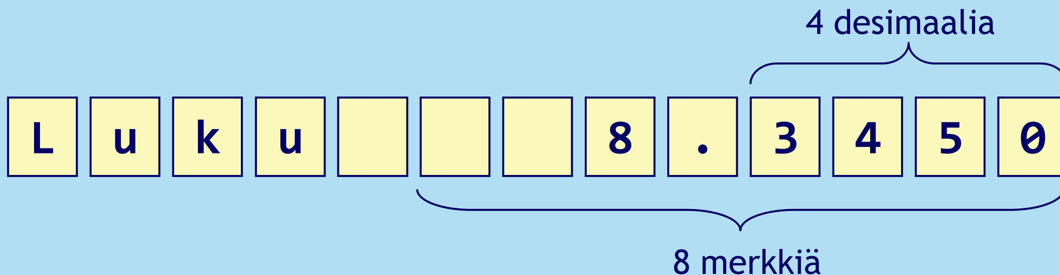
Tuloste:

```
Luku 1 on 2
Luku 1 on 2
Luku 2 on noin 4.56
2 ja 4.56
```

Kun tulostetaan kokonaislukuja, on tyyppikirjaimen **d** käyttäminen tarpeetonta, jos luvun leveyttä ei haluta määritellä. Tällöin riittää ilmoittaa tulostettavan muuttujan nimi.

```
luku = 8.345
print(f"Luku {luku:8.4f}")
```

Huomaa, että pistettä käytetään vain desimaalien määrää ilmoitettaessa!



# Taulukon tulostaminen

```python
luku = 3.45678
for i in range(3):
  print(f"{i+1:3d}. {i:<2d} {luku*i:7.2f}")
```

```
  1. 0     0.00
  2. 1     3.46
  3. 2     6.91
```

3 merkkiä leveä sarake kokonaisluvulle

2 merkkiä leveä sarake kokonaisluvulle, tasattu vasemmalle

7 merkkiä leveä sarake, jossa on aina kaksi desimaalia

`{:3d}` `{:<2d}` `{:7.2f}`

Taulukko tulostetaan ilmoittamalla sarakkeiden leveydet. Kaikki aaltosulkeiden ulkopuolella olevat merkit tulostuvat sellaisenaan, tässä esimerkissä piste ja välilyönnit sarakkeiden välissä.

Pelkän muuttujan nimen lisäksi hakasulkeissa voi olla myös yksinkertaisia laskutoimituksia ym., jonka tulos halutaan tulostaa ilman, että arvo tallennetaan ensin erilliseen muuttujaan.

# 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! Kaikki funktioon kuuluva koodi pitää sisentää funktion sisään

# 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 funktiota kutsuttaessa!

# 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():
  vastaus = input("Kuka olet?\n")
  return vastaus
nimi = kysy_nimi()
```

Yksi parametri:

```
def sano_jotakin(teksti):
  print(teksti)

sano_jotakin("Tulosta tama!")
```

Useampi parametri:

```
def laske_summa(luku1, luku2):
  print("Summa:", 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(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
- Vertaus: Lista on kuin kansio, johon voidaan lisätä ja ottaa pois papereita, sekä lisäksi papereita voidaan laskea, selailla ja järjestellä. Jokaisella kansion sivulla on sivunumero.
- Listaan talletetaan tietoa alkioina, jotka voivat olla erityyppisiä

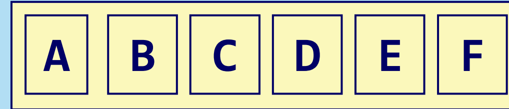
Lista, jonka sisällä on viisi alkiota:

# Listan indeksointi

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

Negatiivinen indeksi: -6 -5 -4 -3 -2 -1

Lista:




Positiivinen indeksi: 0 1 2 3 4 5

- 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

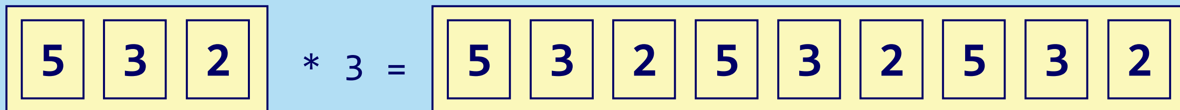
```
elaintarha = ["karhu", "leijona", "kameli"]
for indeksi in range(len(elaintarha)):
  print(elaintarha[indeksi])
```

- Tästä on hyötyä, jos alkion indeksinumeroa tarvitaan silmukan sisällä johonkin

# Listan toistaminen

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

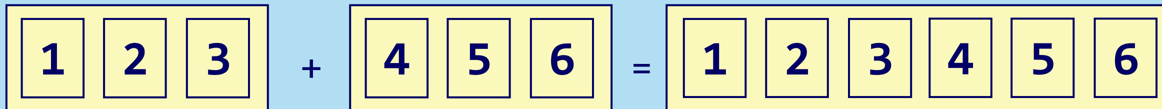
```
numerolista = [5, 3, 2]
toistettu_lista = numerolista * 3
```



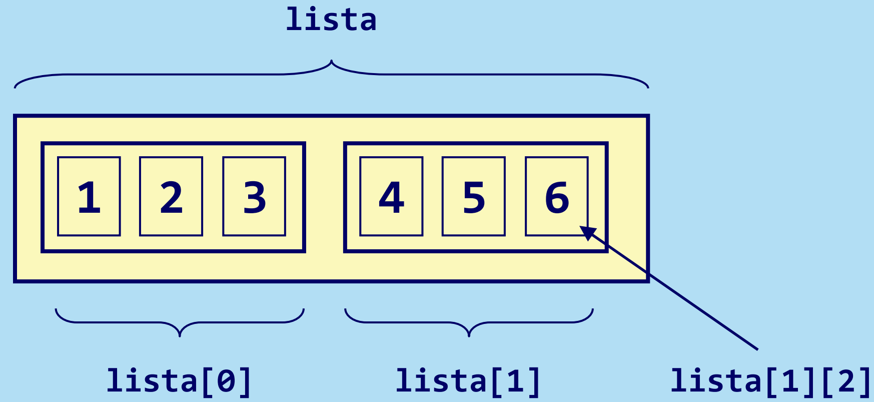
# Listojen yhdistäminen

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

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



# Lista listan sisällä




```
lista = [[1, 2, 3], [4, 5, 6]]
print(lista)          # [[1, 2, 3], [4, 5, 6]]
print(lista[1])       # [4, 5, 6]
print(lista[1][2])    # 6
```

# Merkkijono listana

- Merkkijonon voi kuvitella eräänlaiseksi listaksi, joka koostuu yksittäisistä merkeistä
- Merkkijono toimii listan tavoin, kun halutaan käydä merkkijono läpi tai viitata merkkeihin
- Useimpia komentoja, jotka alkavat listan nimellä ja pisteellä, ei voi kuitenkaan käyttää! (esim. `insert` tai `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 = 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 = 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
```

osat =

| Osa 1 | Osa 2 | Osa 3 |
|---|---|---|

Indeksi: 0 1 2

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 = input("Anna 1. merkkijono\n")
mjono2 = 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
- Vertaus: Sanakirja on kuin lokerikko, jossa jokaisessa lokerossa on nimitarra. Esineet laitetaan tietyn-nimiseen lokeroon, josta esine myös löydetään, kun tiedetään, minkä nimisestä lokerosta sitä etsitään.
- Arvoja voi hakea vain avaimen avulla
- Avain voi olla esimerkiksi luku tai merkkijono, arvo voi olla mitä hyvänsä tyyppiä
- Sanakirjassa ei voi olla kahta samaa avainta

# Sanakirjan toiminnot

Luo tyhjä sanakirja:

```
hinnat = {}
```

Luo sanakirja ja lisää siihen avain-arvo -pareja:

```
hinnat = {"kahvi" : 1.8, "pulla" : 2.2}
```

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

```
hinnat["kaakao"] = 2.6
```

Onko avain sanakirjassa:

```
if "tee" in hinnat:
```

Poista avain (ja sen arvo) sanakirjasta:

```
del hinnat["pulla"]
```

Hae avainta vastaava arvo:

```
hinta = hinnat["kahvi"]
```

Avaimien kokonaismäärä:

```
tuotteita = len(hinnat)
```

Tuotteen nimi on tässä avain ja tuotteen hinta sitä vastaava arvo. Näin tuotteen hinta voidaan hakea helposti tuotteen nimen avulla.

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

Avaimen tai arvon paikalla voi olla normaaliin tapaan myös muuttuja.

# 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 läpikäyntijärjestyksestä ei pidä olettaa mitään!

# Sanakirjan läpikäynti

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

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

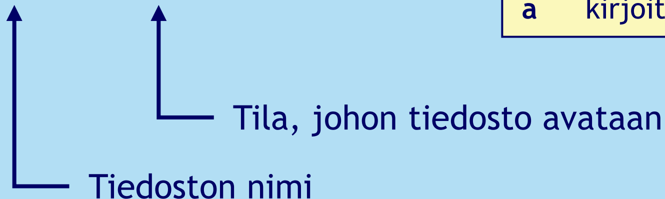
- Avainta vastaavaa arvoa ei tarvitse nyt hakea erikseen sanakirjasta kuten tässä:

```
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 |

- 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\n")
tiedosto.write("Toinen rivi")
tiedosto.close()
```

- Komennon parametrin tulee olla merkkijono, tarvittaessa käytetään tyyppimuunnosta
- Mikäli annettua tiedostoa ei ole olemassa, luodaan uusi tiedosto

# Poikkeukset

- Ohjelman suorituksen aikana voi aiheutua erilaisia virhetilanteita
- Tätä varten voidaan määritellä virhetilanteita (eli poikkeuksia) 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 rivia 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, OSError):
  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(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
- `OSError`: avattavaa tiedostoa ei löydy tai sen käsittelemisessä tapahtui virhe
- `IndexError`: indeksi ei ole sallitulla välillä
- `KeyError`: avainta ei ole sanakirjassa
- `ZeroDivisionError`: ohjelmassa on yritetty jakaa nollalla

Muista, ettei kaikkia tilanteita kannata käsitellä poikkeuksilla. Usein on esimerkiksi helpompaa tarkistaa `if`-käskyllä, onko sanakirjassa haluttua avainta kuin käsitellä vastaava poikkeus.

# 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 olion sisällä olevia muuttujia, jotka voivat sisältää lukuja, 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 luokan ulkopuolella olevasta koodista

# 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 kuin sitä ei olisi metodin määrittelyssä lainkaan
- Jos luokka on määritelty eri tiedostossa kuin sitä käyttävä ohjelma, lue myös **import**-käskystä sivulta 70

# 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(f"Tuote maksaa {hinta:.2f} euroa")
karkkipussi.aseta_hinta(hinta * 1.2)
```

- HUOM! **`self`**-parametri jätetään tässäkin kokonaan huomiotta, aivan kuin sitä ei olisi metodin määrittelyssä lainkaan

# Metodien kutsuminen

- Metodin sisällä voidaan kutsua saman tai jonkin toisen olion metodeita

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

**onko_halvempi**-metodi voi selvittää oman hintansa lisäksi myös parametrina annetun toisen **Tuote**-olion hinnan kutsumalla tämän olion **kerro_hinta**-metodia

```
class Tuote:

  ...

  def kerro_hinta(self):
    return self.__hinta

  def onko_halvempi(self, toinen):
    oma_hinta = self.kerro_hinta()
    toisen_hinta = toinen.kerro_hinta()
    if oma_hinta < toisen_hinta:
      return True
    else:
      return False
```

# __str__

- Luokkaan voi määritellä **__str__**-metodin, joka suoritetaan automaattisesti, kun olio halutaan tulostaa **print**-käskyllä
- Metodin tulee palauttaa aina merkkijono. Apuna voi käyttää merkkijonojen muotoilua.

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

  def __str__(self):
    return f"{self.__nimi} maksaa {self.__hinta:.2f} euroa."
```

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

# import-käsky

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

```
import tuote


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

Tässä luokka **Tuote** on määritelty tiedostossa **tuote.py** ja olio luodaan aina ilmoittamalla moduulin ja luokan nimi

```
from tuote import *


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

Tässä luokka **Tuote** on määritelty myös tiedostossa **tuote.py**, mutta olion voi luoda pelkällä luokan nimellä

Osoitteessa `http://docs.python.org/` on paljon hyödyllistä lisämateriaalia.