

Petteri Mäkelä

# **Ohjelmoinnin perusteet ja IoT-sovellukset Python-ohjelmointikielellä**

# SISÄLTÖ

SISÄLTÖ .....	2
1 Johdanto .....	5
2 Ohjelmoinnin perusteet .....	6
2.1 Python-ohjelmointikieli ja kehitysympäristöt .....	6
2.2 Ensimmäinen sovellus .....	6
2.3 Syöttö ja tulostus .....	6
2.4 Laskutoimitukset .....	8
2.5 Valintarakenteet .....	8
2.6 While-lause .....	9
2.7 For-lause .....	10
2.8 Tulostuksen muotoilu .....	10
2.9 Funktiot .....	11
2.10    Satunnaisluvut .....	12
2.11    Tehtäviä .....	12
3 Listat .....	14
3.1 Listan esittely .....	14
3.2 Listan toiminnot .....	14
3.3 Tietojen lukeminen tiedostosta listaan .....	16
3.4 Merkkijonot ja listat .....	17
3.5 Split .....	18
3.6 Merkkijonon kirjainten vaihtaminen .....	18
3.7 Tehtäviä .....	18
4 Olio-ohjelmointi .....	20
4.1 Luokat ja oliot .....	20
4.2 Oliosunnittelu .....	20
4.3 Luokan määrittely .....	21
4.4 Olion luominen .....	21
4.5 __str__-metodi .....	22
4.6 Luokan määrittelemine toiseen tiedostoon .....	23
4.7 Tehtäviä .....	24

5	Sanakirja.....	25
5.1	Sanakirjan perustoiminnot.....	25
5.2	Sanakirjan läpikäyminen .....	26
5.3	Listat sanakirjan arvoina .....	26
5.4	Oliot sanakirjassa .....	29
6	Kuvaajien piirtäminen.....	32
6.1	NumPy- ja Matplotlib-moduulien asentaminen .....	32
6.2	NumPy-moduuli.....	32
6.3	Kuvaajien piirtäminen Matplotlib-moduulin avulla .....	34
6.4	Tiedostosta lukeminen .....	36
6.5	Esimerkki: Pythonin listojen tulostaminen .....	38
6.6	Tehtäviä .....	39
7	JSON ja sarjallistaminen .....	41
7.1	Sarjallistaminen (serialisointi).....	41
7.2	Sarjallistuksen purkaminen (deserialisointi) .....	43
8	Socket-ohjelmointi.....	44
8.1	TCP/IP-protokolla ja IP-osoite.....	44
8.2	Socket-yhteyden muodostaminen .....	45
8.3	Asiakas .....	45
8.4	Palvelin .....	46
8.5	Tiedon välitys JSON-muodossa .....	48
8.6	Harjoituksia .....	49
9	Python Flask ja REST API .....	50
9.1	REST API.....	50
9.2	Postman.....	50
9.3	Flask .....	51
9.4	Template .....	54
9.5	Esimerkki .....	56
9.6	POST -metodin käyttö HTML-sivulta.....	58
9.7	REST API.....	59
9.8	Requests.....	62
9.9	Tehtävä .....	63
9.10	Socket.io .....	64

10MQTT.....	65
10.1 Julkaisija .....	66
10.2 Tilaaja .....	66

# 1 Johdanto

Python-ohjelmointikieli on helppo oppia. Pythonissa on yksinkertainen syntaksi ja monipuoliset valmiit tietorakenteet. Python-ohjelmointikieli on tullut viime aikoina suosituksi myös siksi, että siinä on monipuoliset kirjastot tieteelliseen laskentaan ja tekoälyn sovelluksiin.

Tämä opas on tarkoitettu niille, jotka ovat opetelleet jo ohjelmoinnin perusteet jollakin muulla ohjelmointikielellä. Oppaassa kerrataan ohjelmoinnin perusasiat vain niiltä osin, miten Python poikkeaa muista yleisistä ohjelmointikielistä, kuten Java, C# ja C. Ohjelmoinnin perusteiden opiskeluun Python-ohjelmointikielellä löytyy monia muita täydellisimpiä oppaita, myös suomen kielellä.

Oppaan toisena tarkoituksena opastaa lukijaa tekemään yksinkertaisia IoT-sovelluksia Python-kielellä. Oppaassa käsitellään viestien välitys MQTT:llä, REST API ja web-pohjaisen sovelluksen tekeminen Python Flask -ohjelmointikielellä.

Seinäjoella 10.1.2020

Petteri Mäkelä

[petteri.makela@seamk.fi](mailto:petteri.makela@seamk.fi)

## 2 Ohjelmoinnin perusteet

### 2.1 Python-ohjelmointikieli ja kehitysympäristöt

Python on tulkattava ohjelmointikieli. Pythonia pidetään helppona oppia ja sitä jotkut suosittelevat ensimmäiseksi ohjelmointikieleksi. Pythonia käytetään usein komentoriviltä suoritettavissa skripteissä. Python on yleinen ohjelmointikieli myös tieteellisessä laskennassa (kirjastot NumPy ja SciPy). Python on yleistynyt viime vuosien aikana verkkosivustojen ohjelmointikielenä (esimerkiksi Django, Flask)

Pythonista on käytössä kaksi versiota 2.x ja 3.x, jotka eivät ole keskenään yhteensopivia. Tässä oppaassa käytetään versiota 3. Pythonin voi ladata osoitteesta <https://www.python.org/downloads/>. Kehitysympäristöksi(IDE) on useita vaihtoehtoja kuten PyCharm, Visual Studio, Eclipse, Netbeans ja **Visual Studio Code**.

### 2.2 Ensimmäinen sovellus

Kirjoita seuraava rivi tiedostoon hello.py.

```
print("Hello world!")
```

Aja ohjelma samasta kansioista, johon tallensit tiedoston.

```
py hello.py
```

Ohjelma tulostaa Hello world!.

### 2.3 Syöttö ja tulostus

Tutustutaan syötteiden lukemiseen ja tulostukseen esimerkin avulla. Tehdään ohjelma, joka kysyy käyttäjältä nimen ja syntymäajan. Käyttäjän annettua nämä tiedot ohjelma tulostaa henkilön nimen ja iän.

Tee ohjelma alla olevan mallin mukaan ja kokeile sitä.

```

import datetime

# user input
firstName = input("first name: ")
lastName = input("last name: ")
sBirthYear = input("birth year: ")

# convert to int
birthYear = int(sBirthYear)

# current date
today = datetime.date.today()

# calculate age
age = today.year - birthYear

# output
print(firstName + " " + lastName + " is " + str(age) + " old")

```

Edellä olevassa esimerkissä päivämäärän käsittelyyn liittyvät funktiot saadaan käyttöön kirjoittamalla tiedoston alkuun `import datetime`. Nykyinen ajanhetki saadaan funktion `datetime.date.today()`-funktion avulla.

Huomaa, että Python ohjelmoinnissa ei käytetä puolipistettä lauseen lopussa, kuten esimerkiksi Java-ohjelmoinnissa. Lause päättyy rivinvaihtoon.

Käyttäjän syöte luetaan funktiolla `input`. Input-funktiolle annetaan parametrina käyttäjälle näytettävä kehote. Input-palauttaa käyttäjän antaman syöteen merkkijonona. Konsolille tulostetaan funktiolla `print`.

Python ohjelmoinnissa muuttujille ei määritellä tietotyyppiä esittelyn yhteydessä. Muuttujan tyyppi määräytyy sen saaman arvon mukaan. Pythonissa on käytössä samankaltaiset tietotyypit kuin muissakin ohjelmointikielissä. Edellisessä esimerkissä muuttujat `firstName`, `lastName` ja `sBirthYear` ovat merkkijonotyyppisiä, koska `input`-funktio palauttaa merkkijonon.

Tyypimuunnos merkkijonosta kokonaisluvuksi tehdään funktiolla `int`. Funktio `float` muuttaa merkkijonon desimaaliluvuksi. Muunnos kokonaisluvusta merkkijonoksi tehdään funktiolla `str`. Kommentit alkavat merkillä `#`.

## 2.4 Laskutoimitukset

Laskutoimitukset tapahtuvat Pythonissa samalla tavoin kuin muissakin ohjelmointikielissä. Python-kielessä on kuitenkin joitakin poikkeuksia. Esimerkiksi kahden kokonaisluvun jakolasku palauttaa desimaaliluvun:  $1/3 = 0.3333333$ .

Desimaaliluvun katkaisevassa laskutoimituksessa käytetään kahta jakomerkkiä:  $5//3 = 1$ . Potenssiin korotus tehdään seuraavasti:  $2**8 = 256$ .

Vakiot merkitään isoilla kirjaimilla:  $SCALE = 1.5$ .

## 2.5 Valintarakenteet

Ehtolauseen merkintätapa poikkeaa muista ohjelmointikielistä.

```
if age < 50:
    print(firstName + " you are ")
    print("young")
elif age < 60:
    print("middle-age")
else:
    print("old")
```

Huomaa kaksoispisteet if-, elif- ja else-rivien perässä. Huomaa myös, että ehdon ei tarvitse olla suluissa.

Python-ohjelmoinnissa lohkoja ei rajata aaltosulkeilla kuten muissa ohjelmointikielissä. Sen sijaan tyhjällä merkillä rivin alussa (eli sisennyksellä) on merkitystä. Sisennykset määrittelevät mihin loogiseen joukkoon rivi kuuluu.

Alla olevassa ohjelmassa on esimerkkejä vertailuoperaattoreiden käytöstä. Huomaa esimerkistä myös not-, and- ja or-operaattoreiden käyttö.



```

# vertailuoperaattoreita
if luku1 > luku2:
    print("luku1 on suurempi kuin luku2")

if luku1 != luku2:
    print("luvut ovat erisuuria")

if luku1 <= 0:
    print("luku1 on pienempi tai yhtäsuuri kuin 0")

# ehdon voi kääntää sanalla not
if not luku1 < 3:
    print("luku1 ei ole pienempi kuin 3")

# ehtoja voi yhdistellä JA- ja TAI-operaattoreilla
if luku1 > 4 and luku1 < 8 or luku1 > 10:
    print("luku1 on välillä 5...7 tai suurempi kuin 10")

# Pythonissa voi tehdä myös näin
if 4 < luku1 < 8:
    print("luku1 on välillä 5...7")

```

## 2.6 While-lause

Alkuehtoinen toistolause eli while-lause tehdään seuraavalla tavalla:

```

counter = 0
while counter < 3:
    print(counter)
    counter+=1
print("loop was executed ", counter, " times")

```

Huomaa, että while-rivin perässä on kaksoispiste ja sulkuja ei käytetä toistoehdon ympärillä. Laskurin kasvattaminen yhdellä tehdään lauseella `counter+=1`. Muista ohjelmointikielistä tuttua `i++`-tyyppistä merkintätapaa ei ole käytössä. Aivan kuten ehtolauseen tapauksessa sisennyksillä on merkitystä. Toistettavaan lohkoon kuuluvat saman verran sisennetyt lauseet. Edellisessä esimerkissä viimeistä riviä ei toisteta, koska sitä ei ole sisennetty.

## 2.7 For-lause

Pythonin for-lause poikkeaa paljon muista ohjelmointikielistä.

```
# numbers 0-4
for i in range(5):
    print(i)

# numbers 4-7
for i in range(4, 8):
    print(i)

# numbers 0, 2, 4, 6, 8
for i in range(0, 10, 2):
    print(i)
```

For-lauseessa ei ole näkyvissä laskurimuuttujan kasvatusta, vaan toistettava väli ja askel annetaan range-funktiolle parametrina. Huomaa, että väli on puoliavoin: alku kuuluu väliin, mutta loppu ei.

## 2.8 Tulostuksen muotoilu

Python-kielen tulostuksen muotoilu jonkin verran poikkeaa muista ohjelmointikielistä. Linkissä <http://www.cs.hut.fi/~ttsirkia/Python.pdf> löytyy hyviä esimerkkejä tulostuksen muotoiluun.

Print-lauseessa muuttujat voidaan erotella pilkuilla:

```
print("Luvut ovat", luku1, "ja", luku2)
```

Sama tulostus saadaan myös muotoilukoodia käyttämällä:

```
print("Luvut ovat {:d} ja {:d}".format(luku1, luku2))
```

Desimaaliluku voidaan muotoilla seuraavalla tavalla:

```
luku = 2.2453653532
print("Tulos on {:>6.2f}".format(luku))
```

Muotoilukoodi koostuu seuraavista osista:

- { : aloitusmerkkivara
- > tasaus oikealle
- 6 varattavan tilan leveys
- .2 tulostettavien desimaalin määrä
- f tyyppin tunnus (float)
- } lopetusmerkki

Taulukkomainen tulostus syntyy seuraavasti:

```
saldo = 100
korkoprosentti = 0.05
for kuukausi in range(3):
    korko = saldo * korkoprosentti
    saldo += korko
    print("Kuukausi {:>2d} Korko {:>8.2f} Saldo {:>8.2f}"
          .format(kuukausi, korko, saldo))
```

Tulostus:

```
Kuukausi 0 Korko      5.00 Saldo    105.00
Kuukausi 1 Korko      5.25 Saldo    110.25
Kuukausi 2 Korko      5.51 Saldo    115.76
```

## 2.9 Funktiot

Ohjelma kannattaa jakaa useampaan funktioon. Yleensä ohjelmissa on vähintään yksi funktio pääohjelma eli main. Funktio määritellään avainsanalla def ja sitä kutsutaan funktion nimellä. Kaikki funktion toteutukseen kuuluva koodi pitää sisentää funktion sisään.

Seuraava esimerkki koostuu main-funktiosta ja funktiosta, joka laskee pallon tilavuuden. Huomaa, että pääohjelmaa kutsutaan uloimmalta sisennystasolta.

```

import math

def pallonTilavuus(sade):
    tilavuus = 4/3 * math.pi * sade ** 3
    return tilavuus

def main():
    r = float(input("Anna pallon sade: "))
    V = pallonTilavuus(r)
    print("Pallon tilavuus on {:.3f}".format(V))

# main-funktion kutsu
main()

```

## 2.10 Satunnaisluvut

Ohjelmissa tarvitaan usein satunnaisuutta. Satunnaislukuja voidaan generoida Pythonissa random-kirjaston avulla. Alla on esimerkki satunnaislukujen käyttämisestä.

```

import random

# satunnainen kokonaisluku väliltä 0 - 10
luku = random.randint(1, 10)
print(luku)

# satunnainen liukuluku välillä 0 - 1
x = random.random()
print(x)

# satunnainen liukuluku välillä -5 <= x <= 5
y = random.random() * 10 - 5
print(y)

# 10 satunnaislukua
for i in range(10):
    luku = random.randint(1, 6)
    print(luku, end=" ")

```

## 2.11 Tehtäviä

Tässä luvussa on tehtäviä ohjelmoinnin perusteisiin liittyen.

1. Tee ohjelma, joka kysyy käyttäjältä lukuja, kunnes käyttäjä antaa negatiivisen luvun. Ohjelma laskee lukujen keskiarvon.

2. Tee ohjelma, joka tulostaa muunnostaulukon senttimetreistä tuumiksi välillä 10 – 100 senttimetriä 10 senttimetrin välein.
3. Tee ohjelma, joka tulostaa kaikki ASCII-merkit ja niiden lukuarvot kokonaislukuina ja heksadesimaalilukuina.
4. Tee ohjelma, joka tulostaa tasalyhenteisen lainan tiedot kuukausittain. Ohjelma kysyy käyttäjältä lainan määrän, lyhennyksen suuruuden ja korkoprosentin. Ohjelma tulostaa kuukausittain kuukausierän, koron ja jäljellä olevan lainan määrän. Käytä muotoilukodeja niin, että tulostus on siisti.
5. Tee ohjelma, joka etsii kokeilemalla funktion  $f(x) = 2x^2 + 2x + 8$  pienimmän arvon.
6. Tee ohjelma, joka tulostaa pallon lentoradan x- ja y-koordinaatit sekunnin välein (vino heittoliike fysiikasta)

## 3 Listat

### 3.1 Listan esittely

Listat ovat Pythonissa monipuolisempia kuin joissakin muissa ohjelmointikielissä. Lista esitellään näin:

```
lista = []
```

Seuraavassa esimerkissä on esitelty valmiiksi määritelty lista ja tulostettu se kahdella eri tavalla.

```
hajyt = ["Isotalo", "Rannajarvi", "Pukkila"]  
# foreach  
for hajy in hajyt:  
    print(hajy)  
  
# indeksin avulla  
for i in range(len(hajyt)):  
    print("Hajy nro ", i + 1, " on ", hajyt[i])
```

Listan sisältö tulostetaan ensin iteroivalla for-lauseella (joissakin ohjelmointikielissä foreach). Tässä tapauksessa lista käydään aina kokonaisuudessaan läpi. Toisessa tulostuksessa lista käydään läpi indeksin avulla. Tässä vaihtoehdossa voidaan listasta käydä vain haluttu alue läpi.

### 3.2 Listan toiminnot

Seuraavassa esimerkissä on esitelty listan toimintoja.

```

lista = [5,3,6,7,9,8,1]

# uuden alkion lisääminen listan loppuun
lista.append(4)
# alkion lisääminen tiettyyn paikkaan
lista.insert(1, 2)
# listan järjestäminen
lista.sort()
# listan järjestyksen kääntäminen
lista.reverse()
# listan pituus
lukumaara = len(lista)

# listan tulostus
for item in lista:
    print(item)

# onko alkio listassa
if 6 in lista:
    print("Listassa on alkio 6")

# alkion paikka listassa (haku)
print(lista.index(6))

```

Listaan voidaan tallentaa myös erityyppisiä alkioita. Listan indeksointi alkaa nol-  
lasta, kuten useimmissa ohjelmointikielissä.

Listasta voi luoda kopion \*-operaattorilla. Tällöin tehdään uusi lista, joka sisältää  
annetun määrän kopioita alkuperäisestä listasta.

```

lista = [3, 4, 5]
uusilista = lista * 3
print(uusilista)

```

Uuden listan alkiot ovat [3, 4, 5, 3, 4, 5, 3, 4, 5].

Listoja voidaan yhdistää +-operaatiolla.

```

lista1 = [3, 4, 5]
lista2 = [6, 7, 8]
uusilista = lista1 + lista2
print(uusilista)

```

Uuden listan alkiot ovat nyt [3, 4, 5, 6, 7, 8].

Listan sisällä voi olla myös toinen lista.

```
lista = [[1, 2, 3],[4, 5, 6]]
print(lista)      # [[1, 2, 3], [4, 5, 6]]
print(lista[0])   # [1, 2, 3]
print(lista[1])   # [4, 5, 6]
print(lista[0][2]) # 3
```

### 3.3 Tietojen lukeminen tiedostosta listaan

Tiedosto avataan open-komennolla.

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

Ensimmäinen parametri on tiedoston nimi polkuineen. Toisessa parametrissa määritellään, avataanko tiedosto kirjoittamista (w), lukemista (r) vai lisäämistä varten (a).

Tiedosto suljetaan close-komennolla.

```
tiedosto.close()
```

Tiedosto luetaan rivi kerrallaan seuraavasti.

```
tiedosto=open("nimet.txt","r")
for rivi in tiedosto:
    rivi=rivi.rstrip() # poistaa rivinvaihdon rivin lopusta
    print(rivi)
tiedosto.close()
```

Rivi luetaan aina merkkijonona.

Tiedostoon voidaan kirjoittaa merkkijonoja write-komennolla.

```
tiedosto=open("data.txt","w")
tiedosto.write("Ensimmäinen rivi\n")
tiedosto.write("Toinen rivi")
tiedosto.close()
```



### 3.4 Merkkijonot ja listat

Merkkijonon voi ajatella yksittäisistä merkeistä koostuvaksi listaksi. Merkkijonon merkkeihin voi siis viitata samalla tavalla hakasulkeilla kuin listan alkioihin.

```
sana = "Ohjelmointi on kivaa"

ekakirjain = sana[0]
print(ekakirjain) # O

for i in range(len(sana)):
    print(sana[i], end = ",")
print()
# O,h,j,e,l,m,o,i,n,t,i, ,o,n, ,k,i,v,a,a,

for kirjain in sana:
    print(kirjain, end = " ")
print()
# O h j e l m o i n t i   o n   k i v a a
```

Huomaa edellisestä esimerkistä myös print-funktion end-parametrin käyttö, jolla vältetään rivinvaihto ja asetetaan erotinmerkki.

Useimpia listaan liittyviä algoritmeja ei voi kuitenkaan käyttää merkkijonojen yhteydessä (esimerkiksi insert). Merkkijonon merkkejä ei voi myöskään muuttaa.

Merkkijonon osaan voi viitata näin:

```
merkkijono = "Python"
print(merkkijono[:2]) # Py
print(merkkijono[2:4]) # th
print(merkkijono[1:]) # ython
```

Merkkijonoja voi yhdistää muista ohjelmointikielistä tutulla tavalla +-operaattorilla. Kaikkien yhdisteltävien osien on oltava merkkijonoja. Muut tietotyypit on siis muutettava merkkijonoksi str-funktiolla.

### 3.5 Split

Merkkijono voidaan pilkkoa split-funktiolla. Funktio split palauttaa uuden listan, jonka alkioina ovat annetulla erotinmerkillä erotetut osat. Syntyvän listan alkiot ovat aina merkkijonoja.

```
rivi = "Hannu Hanhi 10000"
osat = rivi.split(" ")
etunimi = osat[0] # Hannu
sukunimi = osat[1] # Hanhi
palkka = float(osat[2]) # 10000
```

### 3.6 Merkkijonon kirjainten vaihtaminen

Merkkijonon sisältöä voi muokata replace-funktiolla:

```
sana = "Puu"
uusisana = sana.replace("P", "K")
print(uusisana) # Kuu
```

Replace-funktio ei muuta alkuperäistä merkkijonoa, vaan palauttaa uuden muokattun merkkijonon. Replace korvaa kaikki merkkijonosta löytyneet korvattavat merkit, ei pelkästään ensimmäistä. Replace toimii myös useammalle merkille:

```
sana = "Ohjelmointi"
uusisana = sana.replace("ointi", "a")
print(uusisana) # Ohjelma
```

Jos korvaava merkkijono on tyhjä, poistetaan korvattavat esiintymät.

### 3.7 Tehtäviä

1. Tee ohjelma, joka kysyy käyttäjältä nimiä, kunnes käyttäjä antaa tyhjän merkkijonon. Nimet talletetaan listaan. Ohjelma järjestää listan aakkosjärjestykseen ja tulostaa lopulta tyhjän listan.

2. Tee ohjelma, joka lukee desimaalilukuja tiedostosta. Kukin luku on tallennettu omalle rivilleen. Ohjelman alussa on esitelty lista. Tiedostoa luetaan rivi kerrallaan. Kullakin rivillä oleva luku muunnetaan numeroksi ( $\text{luku} = \text{float}(\text{rivi})$ ) ja talletetaan listaan. Kun tiedosto on luettu, ohjelma laskee lukujen summan, keskiarvon, keskihajonnan ja mediaanin.
3. Tee ohjelma, joka laskee samalla rivillä annettujen lukujen summan.
4. Tee ohjelma, joka pyytää käyttäjää antamaan lauseen, jossa on useita sanoja. Ohjelma tulostaa kunkin sanan ensimmäisen kirjaimen isolla.
5. Tee ohjelma, joka lukee tekstitiedoston. Ohjelma laskee kuinka monta merkkiä 'a', 'e', 'i', 'o' ja 'u' siinä on.
6. Tee ohjelma, joka simuloi nopanheittoa. Ohjelma generoi siis satunnaislukuja väliltä 1 – 6. Ohjelma kysyy käyttäjältä, kuinka monta kertaa noppaa heitetään. Ohjelma tulostaa nopanheittojen tulosten jakauman.
7. Tee ohjelma, jossa simuloidaan kahden nopan heittoa. Ohjelma tulostaa nopanheiton tulosten jakauman.
8. Tee ohjelma, joka lukee tekstiä tiedostosta. Ohjelma laskee, kuinka monta kertaa siinä esiintyy kukin pieni kirjain. Vain merkit 'a' – 'z' lasketaan.

## 4 Olio-ohjelmointi

### 4.1 Luokat ja oliot

Olio-ohjelmoinnissa rakennuspalikoita eivät ole vain muuttujat ja metodit, vaan suuremmat kokonaisuudet eli luokat (class) ja niiden ilmentymät oliot (object). Olio on jokin ympäristöstään erottuva kokonaisuus, jolla on oma identiteetti, sisäinen rakenne, käytös ja viitteet ympäristöönsä. Oliot kommunikoivat keskenään lähettämällä toisilleen viestejä.

Luokka määrittelee samankaltaisten olioiden rakenteen ja toiminnan. Luokan toiminnot määritellään metodeissa. Luokan ominaisuudet (rakenne) määritellään jäsenmuuttujissa eli attribuuteissa. Ominaisuudet ovat usein suojattu ulkopuolisilta. Ominaisuuksia käsitellään yleensä metodeilla. Olion tila voidaan vaihtaa metodien välityksellä.

Luokka on myös tietotyyppi, samoin kuin ovat myös perustietotyypit. Luokasta voidaan luoda useita olioita, samalla tavalla, kun esimerkiksi float-tietotyypistä voidaan luoda useita muuttujia.

Jokainen olio on jonkin luokan ilmentymä. Luokka määrittelee olion rakenteen ja käyttäytymisen. Ohjelma luo ajon aikana olioita, joilla on olion luokassa määritellyt jäsenmuuttujat ja metodit.

### 4.2 Oliosuunnittelu

Oliosuunnittelu on mallintamista, jossa luokka vastaa jotakin yleiskäsitettä. Esimerkiksi luokka Työntekijä voisi olla yrityksen työntekijän malli, jossa on kaikille työntekijä yhteisiä ominaisuuksia ja toimintoja.

Työntekijä-luokan ominaisuuksia eli attribuutteja tai jäsenmuuttujia voisivat olla etunimi, sukunimi, osoite ja palkka. Luokan toimintoja eli metodeja voisivat olla työntekijän tietojen tulostaminen ja palkan korotus.

Työntekijä-luokasta voidaan luoda useita olioita, esimerkiksi työntekijät Jaakko, Kari ja Matti. Kullakin näistä työntekijöistä on luokan Työntekijä määrittämät ominaisuudet. Kuhunkin työntekijään voidaan myös kohdistaa luokan määrittämiä toimintoja.

### 4.3 Luokan määrittely

Luokka määritellään avainsanalla `class`. Määrittelyn on oltava tiedoston uloimmalla tasolla.

```
class Piste:
    def __init__(self, nimi, x, y):
        self.nimi = nimi
        self.x = x
        self.y = y

    def etaisyyys(self, toinenPiste):
        dx = self.x - toinenPiste.x
        dy = self.y - toinenPiste.y
        return math.sqrt(dx ** 2 + dy ** 2)
```

Kentät eli jäsenmuuttujat määritellään `__init__`-metodissa. `__init__`-metodi muistuttaa muiden ohjelmointikielten konstruktoria. Kenttiin täytyy viitata luokan sisällä aina sanan `self` avulla. Jos kentän nimi alkaa kahdella alaviivalla, kentän arvoon ei voi viitata suoraan luokan ulkopuolelta.

Metodit määritellään avainsanalla `def`. Metodit on määritelty sisennyksien avulla luokkaan kuuluviksi. Metodin määrittelyssä on oltava ensimmäisenä parametrina `self`. `Self` viittaa siihen olioon, jossa metodia suoritetaan.

### 4.4 Olion luominen

Edellä esitelty luokka `Piste` on siis itse määritelty tietotyyppi. Seuraavassa esimerkissä luodaan kaksi oliota `pisteA` ja `pisteB`.

```

class Piste:
    def __init__(self, nimi, x, y):
        self.nimi = nimi
        self.x = x
        self.y = y

    def etaisyys(self, toinenPiste):
        dx = self.x - toinenPiste.x
        dy = self.y - toinenPiste.y
        return math.sqrt(dx ** 2 + dy ** 2)

pisteA = Piste("A", 2, 3)
pisteB = Piste("B", 5, 6)

etaisyysAB = pisteA.etaisyys(pisteB)

```

Sisennykset määrittävät, että oliot pisteA ja pisteB luotiin luokan Piste ulkopuolella (pääohjelmassa).

Olion luomisessa annetaan parametrina `__init__` -metodissa määritellyt tiedot. Self-parametri jätetään kuitenkin huomiotta.

Kun olio on luotu, sen jäseniin voidaan viitata. Edellisessä esimerkissä oliolle pisteA kutsutaan metodia etaisyys. Parametrina on olio pisteB.

#### 4.5 `__str__`-metodi

Luokkaan voidaan määritellä metodi `__str__`, joka vastaa esimerkiksi Java-ohjelmointikielen `toString`-metodia. `__str__`-metodi suoritetaan automaattisesti, kun olion tiedot halutaan tulostaa.

Metodi `__str__`:n tulee palauttaa aina merkkijonon.

```

class Piste:
    def __init__(self, nimi, x, y):
        self.nimi = nimi
        self.x = x
        self.y = y

    def etaisyyys(self, toinenPiste):
        dx = self.x - toinenPiste.x
        dy = self.y - toinenPiste.y
        return math.sqrt(dx ** 2 + dy ** 2)

    def __str__(self):
        return self.nimi + " " + str(self.x) + " " + str(self.y)

pisteA = Piste("A", 2, 3)
print(pisteA) # A 2 3

```

#### 4.6 Luokan määrittelyminen toiseen tiedostoon

Luokka määritellään usein omassa tiedostossaan (moduulissaan). Luokkaa käyttävä ohjelma on silloin eri tiedostossa. Tällöin luokan määrittely tuodaan luokkaa käyttävän ohjelman tietoon import-käskyllä.

Import-käsky voidaan antaa kahdella eri tavalla:

- `import piste`. Luokka `Piste` on määritelty tiedostossa `piste.py`. Olion luomisessa täytyy ilmoittaa myös sen moduulin nimi, missä luokka on määritelty.

```

import piste

def main():
    pisteA = piste.Piste("A", 2, 3)
    print(pisteA) # A 2 3

main()

```

- `from piste import *`. Luokka `Piste` on myös määritelty tiedostossa `piste.py`. Olion luodaan pelkällä olion nimellä ilmoittamatta moduulin nimeä.

```

from piste import *

def main():
    pisteA = Piste("A", 2, 3)
    print(pisteA) # A 2 3

main()

```

#### 4.7 Tehtäviä

1. Määrittele luokka Kortti. Lisää luokkaan jäsenmuuttujat maa ja arvo sekä `__init__`- ja `__str__`-metodit. Luo pääohjelmassa korttipakan kortit ja lisää ne listaan.
2. Määrittele luokka Korttipakka. Jäsenmuuttujana on lista pakan korteista. Metodit ovat `luoPakka`, `sekoita`, `jarjesta` ja `jaaKortit`.
3. Tee ohjelma, joka lukee pisteiden tietoja tiedostosta (nimi, x, y). Ohjelma laskee pisteiden kautta kuljetun polun pituuden. Määrittele ensin luokka Piste. Tee pääohjelmassa tyhjä lista pisteitä varten. Lue tiedostoa rivi kerrallaan. Pilko rivi `split`-funktioilla. Luo Piste-luokan olio ja anna sille parametrina riviltä luetut tiedot. Lisää Piste-olio listaan. Käy lopuksi lista läpi ja laske polun pituus.



## 5 Sanakirja

Sanakirja (*engl. dictionary*) on tietorakenne, joka sisältää avain-arvo -pareja. Sanakirjasta saadaan haettua arvo nopeasti avaimen perusteella. Avain on usein merkijono tai kokonaisluku. Avaimen on oltava yksikäsitteinen, eli sanakirjassa ei saa esiintyä samaa avainta kahteen kertaan. Arvo voi olla mitä tahansa tietotyyppiä, esimerkiksi olio.

### 5.1 Sanakirjan perustoiminnot

Seuraavassa esimerkissä on näytetty, miten sanakirja luodaan. Alkiot lisätään sanakirjaan yksinkertaisesti sijoittamalla arvo tietylle avaimelle.

```
# luodaan sanakirja
henkilot = {}
# lisäään sanakirjaan avain-arvo -pareja
henkilot["176-167"] = "Kovanaama"
henkilot["176-671"] = "Koljatti"
henkilot["176-761"] = "Kepuli"
# vanhaa arvoa voidaan myös muokata
henkilot["176-761"] = "Kaappi-Kake"
```

Seuraavassa esimerkissä on näytetty joitain sanakirjan perustoimintoja:

```
# haetaan avainta vastaa arvo
henkilo = henkilot["176-671"]
# katsotaan, onko avainta vastaava arvo sanakirjassa
if ("176-761" in henkilot):
    print("löytyi")
# poistetaan avain ja arvo sanakirjasta
del henkilot["176-761"]
# alkioiden määrä
henkiloita = len(henkilot)
```

Sanakirja alkioineen voidaan muodostaa myös näin seuraavasti. Tässä pressure, temperature ja humidity ovat avaimia.

```
measurements = { 'pressure':1024, 'temperature':275, 'humidity':33.0 }
```

## 5.2 Sanakirjan läpikäyminen

Sanakirjan avaimet voidaan käydä läpi for-lauseen avulla:

```
for avain in henkilot:  
    print(avain)
```

Sanakirja perustuu tietorakenteeseen nimeltä hajautustaulu (*engl. hash table*). Hakeminen hajautustaulusta on nopeaa, mutta avaimet eivät tulostu järjestyksessä.

Avain-arvo -parit voidaan käydä läpi for-lauseella seuraavasti:

```
for avain, arvo in henkilot.items():  
    print("avain:", avain, "arvo:", arvo)
```

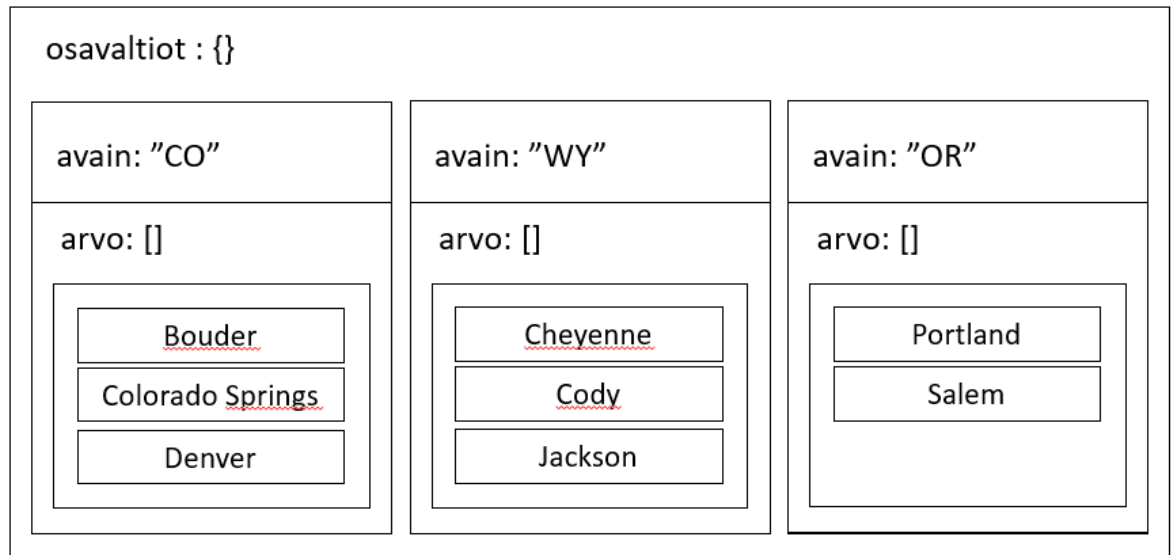
Avain-arvo -parit voidaan käydä läpi myös näin:

```
for avain in henkilot:  
    print("avain:", avain, "arvo:", henkilot[avain])
```

Tämä tapa on kuitenkin hitaampi, koska arvo täytyy hakea erikseen joka kierroksella.

## 5.3 Listat sanakirjan arvoina

Tehdään seuraavaksi ohjelma, joka lajittelee kaupungit osavaltioittain listoihin. Ohjelmassa luodaan sanakirja osavaltioista, jonka avaimina on osavaltioiden tunnukset. Avainta vastaavina arvoina on listat, jotka sisältävät tähän osavaltioon kuuluvat kaupungit.



Tietorakenne koostuu siis sanakirjasta, joka sisältää listoja alla yllä kuvan mukaisesti.

Kaupunkien tiedot luetaan tiedostosta. Tiedostossa ensimmäisessä sarakkeessa on kaupungin nimi, sitten on puolipiste erotinmerkinä ja sitten osavaltion tunnus.

```

Denver;CO
Salem;OR
Boulder;CO
Jackson;WY
Portland;OR
Colorado Springs;CO
Cheyenne;WY
Gold Hill;CO
Cody;WY

```

Kopioi edellä olevat rivit tekstitiedostoon cities.txt.

Tee sitten ohjelma tiedostoon kaupungit.py. Luo ensin sanakirja ja tee sitten tiedoston lukeminen:

```
# avataan tiedosto lukemista varten
tiedosto = open("cities.txt")

# luodaan sanakirja osavaltioita varten
osavaltiot = {}

# käydään tiedosto läpi
for rivi in tiedosto:
    rivi = rivi.rstrip() # rivinvaihdon poisto
    print(rivi)
tiedosto.close()
```

Palastele sitten rivi split-funktiolla. Ensimmäisessä osassa on kaupunki ja toisessa osassa on osavaltion tunnus. Tarkista sitten, löytyykö sanakirjasta rivillä ollut osavaltion tunnus. Jos tunnusta ei löytynyt, lisää se sanakirjaan. Seuraavaksi lisää rivillä ollut kaupunki tunnusta vastaavaan sanakirjan listaan.

```
# käydään tiedosto läpi
for rivi in tiedosto:
    rivi = rivi.rstrip()
    # palastellaan rivi osiin, syntyy lista rivin osista
    osat = rivi.split(';')
    kaupunki = osat[0]
    tunnus = osat[1]

    # tarkista löytyykö tunnus sanakirjasta
    if not tunnus in osavaltiot:
        # jos ei löydy, luo uusi alkio sanakirjaan.
        # Arvoksi tulee tyhjä lista
        osavaltiot[tunnus] = []

    # lisää kaupunki sanakirjassa olevaan listaan tunnuksen mukaan
    osavaltiot[tunnus].append(kaupunki)
tiedosto.close()
```

Tulosta seuraavaksi osavaltio-sanakirjan sisältö.

```
for avain, kaupunkilista in osavaltiot.items():
    print("Osavaltio:", avain)
    print("Kaupungit:")
    for kaupunki in kaupunkilista:
        print(" - ", kaupunki)
    print()
```

Tulostus näyttää tältä:

```

Osavaltio: CO
Kaupungit:
- Denver
- Boulder
- Colorado Springs
- Gold Hill

```

```

Osavaltio: OR
Kaupungit:
- Salem
- Portland

```

```

Osavaltio: WY
Kaupungit:
- Jackson
- Cheyenne
- Cody

```

## 5.4 Oliot sanakirjassa

Tehdään seuraavaksi ohjelma, joka lukee paikkatietoja tiedostosta. Tiedostossa on yhdellä rivillä kunkin paikan tiedot: tunnus (id), nimi, leveyspiiri ja pituuspiiri. Ohjelma lukee tiedoston rivi riviltä. Kullakin rivillä olevaa paikkatietoa kohti luodaan Paikka-luokan olio. Tämä olio tallennetaan sanakirjaan arvoksi. Paikkatiedon tunnusta käytetään avaimena.

Kopioi seuraavat rivit tiedostoon paikat.txt.

```

1;Espoo;60.206376371;24.656728549
2;Hamina;60.569688146;27.197900579
3;Joensuu;62.602079226;29.759679275
4;Helsinki;60.166640739;24.943536799
5;Kauhava;63.100920012;23.065244367
6;Kouvola;60.866825214;26.705598153
7;Lapua;62.970168655;23.007570333
8;Rovaniemi;66.502790259;25.728478856
9;Utsjoki;69.907828523;27.026541275
10;Vaasa;63.092588875;21.615874122
11;Seinäjoki;62.786662897;22.842279603
12;Savonlinna;61.869802657;28.878498290

```

Tee seuraavaksi luokka Paikka tiedostoon paikka.py.

```
class Paikka:
    def __init__(self, id, nimi, latitude, longitude):
        self.id = id
        self.nimi = nimi
        self.latitude = latitude
        self.longitude = longitude
        self.altitude = 0

    def __str__(self):
        return str(self.id) + " " + self.nimi + " " + \
            str(self.latitude) + " " + str(self.longitude)
```

Huomaa, miten merkkijonon yhdistämisessä tarvittava lause on saatu jaettua kahdelle riville kenoviivaa käyttämällä.

Tee sitten pääohjelma tiedostoon paikkatiedot.py. Alusta main-funktiossa sanakirja paikkatiedoille ja lisää myös tiedoston lukeminen. Kokeile, että saat tiedoston luetua.

```
from paikka import *

def main():
    # huom. encoding='utf-8' skandinaavisia merkkejä varten
    tiedosto = open("paikat.txt", "r", encoding='utf-8')

    paikat = {}

    for rivi in tiedosto:
        # poista rivinvaihto
        rivi = rivi.rstrip()
        print(rivi)

    tiedosto.close()
main()
```

Muokkaa sitten for-lauseen sisältöä seuraavasti. Pilko rivi merkkijonolistaksi ja poimi paikan tunnus, nimi ja koordinaatit listasta. Luo sitten Paikka-luokan olio ja lisää se sanakirjaan. Paikan tunnus toimii avaimena.

```

paikat = {}

for rivi in tiedosto:
    # poista rivinvaihto
    rivi = rivi.rstrip()
    # pilko rivi, poimi osat ja tee tarvittavat tyyppimuunnokset
    osat = rivi.split(';')
    tunnus = int(osat[0])
    nimi = osat[1]
    latitude = float(osat[2])
    longitude = float(osat[3])
    # luo olio
    paikka = Paikka(tunnus, nimi, latitude, longitude)
    # lisää paikka sanakirjaan
    paikat[paikka.id] = paikka

tiedosto.close()

```

Testataan seuraavaksi, että sanakirja muodostui oikein tulostamalla sanakirjan sisältö.

```

tiedosto.close()

for avain, paikka in paikat.items():
    print(paikka)

```

Sanakirjasta voi nyt hakea paikkaan liittyvät tiedot avaimen perusteella.

```

# haetaan paikka
paikanId = int(input("anna paikan id: "))

if paikanId in paikat:
    print("Paikan tiedot: ", paikat[paikanId])

```

## 6 Kuvaajien piirtäminen

Tässä luvussa perehdytään kuvaajien piirtämiseen NumPy- ja Matplotlib-moduuleja hyödyntäen.

### 6.1 NumPy- ja Matplotlib-moduulien asentaminen

Moduulit asennetaan komentoriviltä tai Visual Studio Coden terminaalista pip-ohjelman avulla:

- `pip install numpy`
- `pip install matplotlib`

Vaihtoehtoisesti asennuksen voi tehdä näin: `py -m pip install numpy`.

### 6.2 NumPy-moduuli

Kuvaajien piirtämisessä hyödynnetään usein NumPy-kirjastoa. NumPy-kirjastossa on useita numeeriseen laskentaan sopivia työkaluja:

- taulukko (ndarray), jolla voi kuvata esimerkiksi vektoreita ja matriiseja. Taulukot voivat olla myös moniulotteisia.
- funktioita vektorien käsittelyyn ja matriisilaskentaan, matemaattiset perusfunktiot, tilastolliset funktiot

NumPyn taulukot ovat numeerisessa laskennassa huomattavasti nopeampia kuin Pythonin listat.

NumPy-moduuli tuodaan omaan ohjelmaan import-käskyllä: **`import numpy as np`**. Alla olevassa ohjelmassa on esimerkkejä NumPy-tilukoiden käsittelystä.



```

import numpy as np

# vektorin luominen
vektori = np.array([1, 2, 3, 4])

# taulukon alkioihin viittaaminen tapahtuu kuten listoilla
print(vektori[2])

# 2x2 matriisi tehdään seuraavasti
matriisi = np.array([[1, 2],[6, 8]])
print(matriisi[0, 0])

v0 = np.zeros(5) # [0, 0, 0, 0, 0]
v1 = np.arange(1, 6) # [1, 2, 3, 4, 5]
v2 = np.arange(2, 11, 2) # [2, 4, 6, 8, 10]
v3 = np.linspace(0.1, 0.5, 5) # [ 0.1,  0.2,  0.3,  0.4,  0.5]

```

NumPy sopii hyvin myös matriisilaskentaan. Alla olevassa ohjelmassa on esimerkkejä NumPy:n käytöstä matriisilaskentaan.

```

import numpy as np # pip ir

# vaakavektori
v = np.array([[1,2,3]])
print(v) # [[1 2 3]]

# pystyvektori
x = np.array([[4],[5],[6]])
print(x)    # [[4]
             # [5]
             # [6]]

# matriisien kertolasku
vx = v@x
print(vx)   # [[32]]

xv = x@v
print(xv)   # [[ 4  8 12]
             # [ 5 10 15]
             # [ 6 12 18]]

```

### 6.3 Kuvaajien piirtäminen Matplotlib-moduulin avulla

Matplotlib-moduuli sisältää monipuoliset työkalut kuvaajien piirtämiseen. Matplotlibissä on kaksi erilaista tapaa kuvaajien käsittelyyn. Ensimmäinen tapa on käsitellä kuvaajia ja niiden osia ”oliomaisesti”. Toinen tapa on tehdä kuvaajia samaan tapaan kuin Matlabissa käyttämällä pyplot-kirjastoa. Tässä oppaassa käytetään jälkimmäistä tapaa.

Kuvaajia piirrettäessä ohjelmaan pitää aina tuoda matplotlib.pyplot-moduuli import-käskyllä. Yleensä tarvitaan myös NumPy-moduuli. Näistä moduuleista käytetään usein lyhenteitä np ja plt.

```
import numpy as np
import matplotlib.pyplot as plt
```

Piirretään ensin vektorin y arvot kuvaajaan.

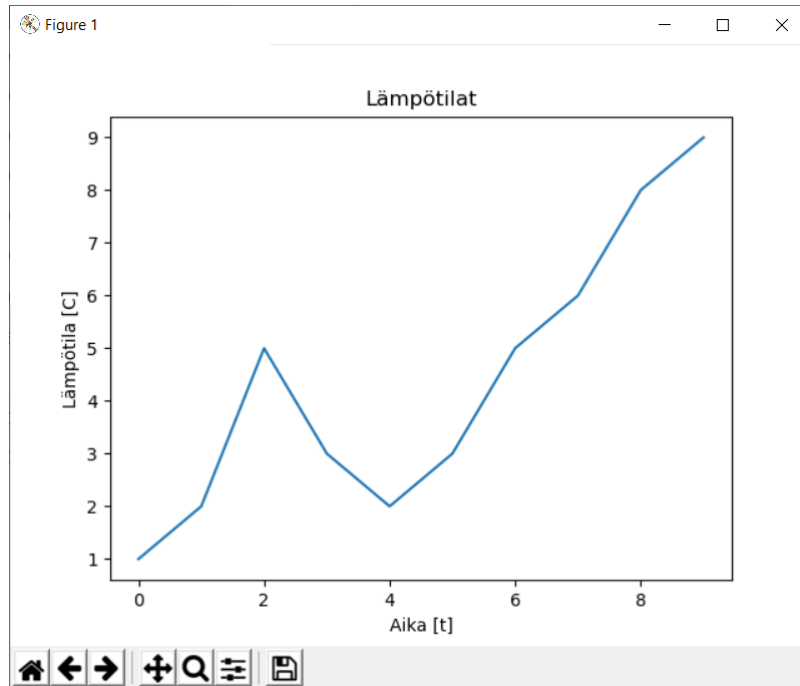
```
import numpy as np
import matplotlib.pyplot as plt

y = np.array([1,2,5,3,2,3,5,6,8,9])

plt.plot(y)
plt.title("Lämpötilat")
plt.xlabel("Aika [t]")
plt.ylabel("Lämpötila [C]")
plt.show()
```

Edellisessä esimerkissä plt.plot()-funktiolle annettiin parametriksi vain yksi vektori, joka sisältää y:n arvot. Tällöin kuvaaja piirretään siten, että x-akselin arvot tulevat y-vektorin alkioden järjestysnumeron mukaan. Sama tulos saataisiin funktiokutsulla plt.plot(x, y), jossa vektorilla x olisi arvot 0, 1, 2, ... n-1.

Kuvaajan ja akselien otsikot voidaan asettaa funktioilla `plt.title()`, `plt.xlabel()` ja `plt.ylabel()`. Tässä esimerkissä funktio `plt.show()` avaa uuden ikkunan, jossa näytetään kuvaaja.



Seuraavassa esimerkissä on tulostettu funktion  $f(x) = \sin(x)$  arvot tietyllä välillä.

```
import numpy as np
import matplotlib.pyplot as plt

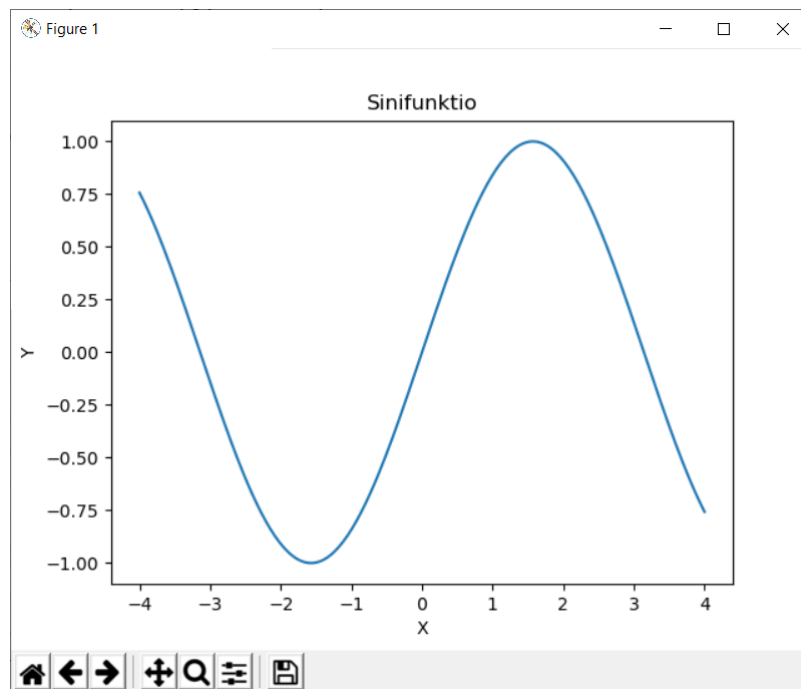
n = 256
# tehdään vektori x, jossa on 256 arvoa välillä -4...4
x = np.linspace(-4, 4, n)
print(x) # [-4. -3.96862745 -3.9372549 -3.90588235 ...
print(x.shape) # (11,)

# lasketaan sinifunktion arvot vektorin x arvoilla
y1 = np.sin(x)

# plotataan y:n arvot kohdissa x
plt.plot(x, y1)
plt.title("Sinifunktio")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

Edellisessä esimerkissä vektori  $x$  tehdään funktion `np.linspace()` avulla. Vektori  $x$  sisältää 256 tasaisesti kasvavaa arvo välillä  $[-4, 4]$ .

Lauseessa  $y = \sin(x)$  tehdään uusi vektori, jossa  $y_i = \sin(x_i)$ . Funktio `plt.plot(x, y)` tulostaa  $y$ :n arvot vektorin  $x$  määrittelemissä kohdissa. Huomaa, että vektorien  $x$  ja  $y$  täytyy olla tässä saman pituiset.



## 6.4 Tiedostosta lukeminen

Taulukon arvot voidaan lukea myös suoraan tekstitiedostosta funktion `np.loadtxt()` avulla. Tekstitiedosto voi sisältää useita rivejä ja sarakkeita. Seuraavassa esimerkissä tiedostossa on kuitenkin vain yksi sarake, jossa kukin luku on tallennettu omalle rivilleen. Tekstitiedoston alku näyttää tältä:

```
3.5
2.9
3.2
3.7
3.2
3.6
3.5
2.9
3
```

Oletetaan, että tiedoston luvut kuvaavat jonkin kappaleen nopeuden x-komponentteja, jotka on mitattu kahden sekunnin välein. Tulostusta varten täytyy siis generoida mittausajanhetkiä kuvaava vektori x, joka on saman pituinen kuin vektori y, ja jossa arvot kasvavat kahden sekunnin välein. Tässä esimerkissä vektori x on tehty funktion `np.arange()` avulla.

```
import numpy as np
import matplotlib.pyplot as plt

# Tiedostossa on mitattuja nopeuksia.
# Kullakin rivillä on yksi nopeusarvo
y = np.loadtxt("numbers.txt")

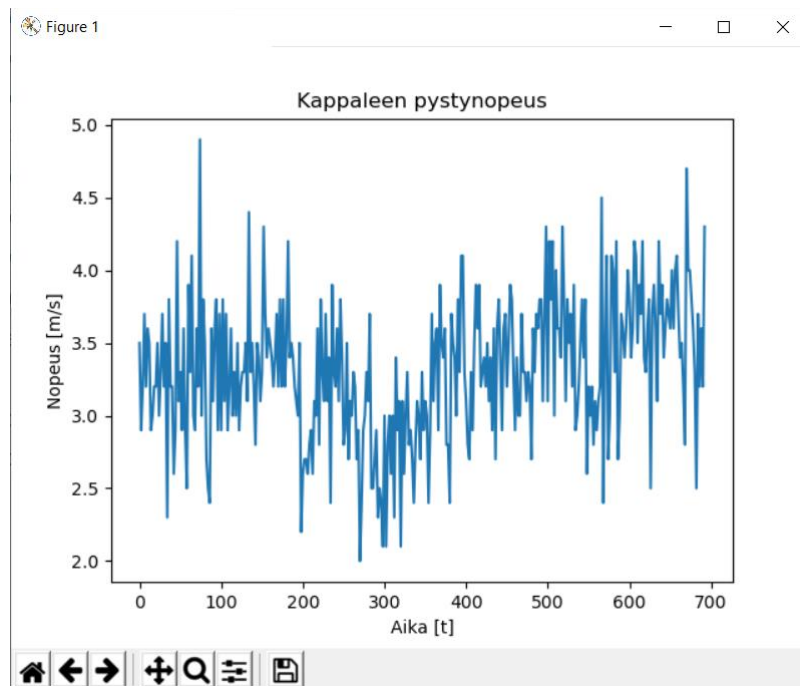
# Tehdään tasaisesti kasvavia arvoja sisältävä vektori x,
# joka on saman pituinen kuin vektori y
x = np.arange(0, y.size)
# Oletetaan, että nopeudet on mitattu kahden sekunnin välein
x = 2 * x

plt.plot(x, y)
plt.title("Kappaleen pystynopeus")
plt.xlabel("Aika [t]")
plt.ylabel("Nopeus [m/s]")
plt.show()
```

Edellä kuvattu ohjelma voidaan tehdä aivan hyvin myös siten, että tiedosto luetaan rivi riviltä ja arvot lisätään tavanomaiseen Python-kielen listaan. Lista voidaan tulostaa `plt.plot()`-funktiolla suoraan tai se voidaan muuntaa `np.array()`-muotoon ennen tulostusta.

```
nopeudet = []
tiedosto = open("numbers.txt", "r")
for rivi in tiedosto:
    rivi = rivi.rstrip()
    nopeus = float(rivi)
    nopeudet.append(nopeus)
# muutetaan lista np.arrayksi
y = np.array(nopeudet)
```

Edellisissä esimerkeissä tehty kuvaaja näyttää nyt tältä:



## 6.5 Esimerkki: Pythonin listojen tulostaminen

Seuraavassa esimerkissä piirretään funktion  $f(x) = x^2 + 2x + 1$  kuvaaja.

Funktion arvot tietyllä välillä voidaan laskea esimerkiksi while-silmukassa ja tallentaa x:n ja y:n arvot tavanomaiseen listaan:

```
xvalues = []
yvalues = []

x = -6
while x <= 4:
    y = x ** 2 + 2 * x + 1
    xvalues.append(x)
    yvalues.append(y)
    x += 0.5
```

Toinen vaihtoehto on hyödyntää NumPy-kirjaston poly1d-luokkaa:

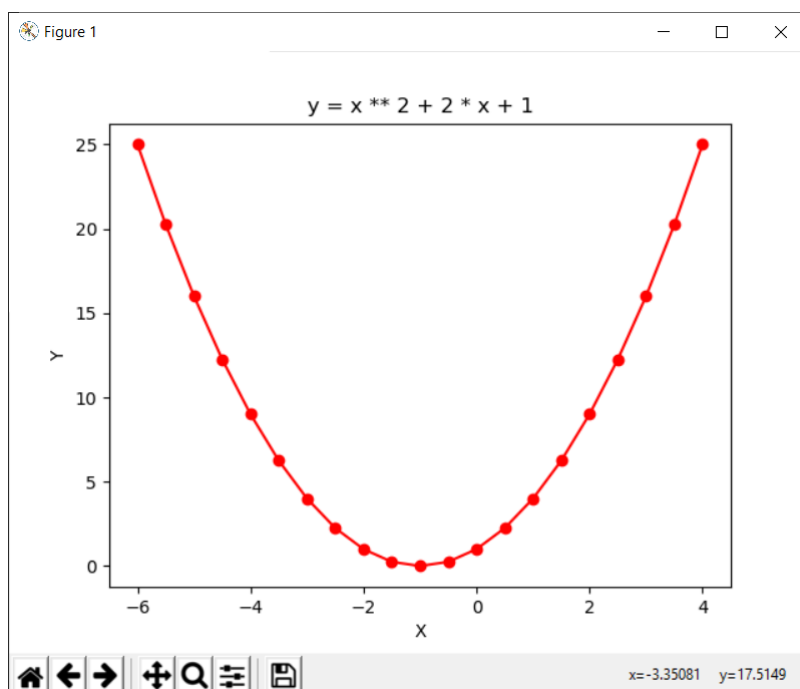
```
# määritellään polynomi  $x^2 + 2x + 1$ 
pol = np.poly1d([1, 2, 1])

# tehdään x-vektori
xvalues = np.arange(-6, 4, 0.5)

# lasketaan polynomifunktion arvot
yvalues = pol(xvalues)
```

Molemmissa tapauksissa kuvaaja piirretään samalla tavalla.

```
# tulostus punaisella värillä ja pallukoilla
plt.plot(xvalues, yvalues, 'o-', color = 'red')
plt.title("y = x ** 2 + 2 * x + 1")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



## 6.6 Tehtäviä

1. Tulosta samaan kuvaan seuraavat funktiot välillä -10 – 10 yhden askeleen välein:

-  $f(x) = 2x + 1$

-  $f(x) = 2\cos(2x)$

2. Tee ohjelma, joka tulostaa pallon lentoradan xy-koordinaatistossa.
3. Tee ohjelma, joka havainnollistaa kaaviossa annuiteettilainan lyhentymistä.



## 7 JSON ja sarjallistaminen

Olio täytyy sarjallistaa ennen kuin se voidaan tallentaa levyille tai siirtää toiseen järjestelmään tietoverkon yli. Sarjallistamisessa olio tai jokin muu tietorakenne kirjoitetaan (*serialize*) tavuina tulostusvirtaan ja luetaan (*de-serialize*) tavuina syötevirrasta. Sarjallistaminen voidaan tehdä monella tavalla. Kirjoittava ohjelma voi sarjallistaa olion tiedot binäärimuodossa tai tekstinä useassa eri muodossa. Lukevan ohjelman täytyy tietää, missä muodossa tieto on sarjallistettu, että se osaa purkaa sarjallistamisen.

JSON on yleinen tekstipohjainen tiedoston esittämismuoto, jota käytetään myös sarjallistamisessa. Useimmissa ohjelmointikielissä on käytössä kirjastot, joiden avulla olioiden ja listojen sarjallistaminen ja sarjallistamisen purkaminen käy helposti. Koska tietorakenteiden esittämistapa JSON-muodossa on standardoitu, voivat eri ohjelmointikielillä tehdyt ohjelmat välittää tietoa toisilleen JSON-muodossa.

### 7.1 Sarjallistaminen (serialisointi)

Sanakirjan sarjallistaminen on helppoa. Seuraava ohjelma tulostaa sanakirjan tiedot JSON-muodossa tallennettuna.

```
import json

mittaus = {'paine': 1024, 'lämpötila': 275, 'kosteus': 50}
# sarjallistaminen niin, että syntyy rivinvaihdot ja sisennykset
mittausJson = json.dumps(mittaus, indent=True)
print(mittausJson)
```

Tulostus näyttää tältä:

```
{
  "paine": 1024,
  "lämpötila": 275,
  "kosteus": 50
}
```

Myös lista on helppo sarjallistaa. Seuraavassa esimerkissä tehdään useampi mittaus-sanakirja jotka lisätään listaan. Lista sarjallistetaan ja tulostetaan konsolille ja tiedostoon.

```
import json

# lista mittauksista
mittaukset = []

# muokataan mittauksia ja lisätään ne sanakirjaksi
for i in range(3):
    # mittausolio sanakirjana. vaihdellaan arvoja
    mittaus = {'aika': i, 'paine': 1024+i, 'lampotila': 275-i, 'kosteus': 50+i}
    mittaukset.append(mittaus)

# sarjallistaminen niin, että syntyy rivinvaihdot ja sisennykset
mittauksetJson = json.dumps(mittaukset, indent=True)
print(mittauksetJson)

# kirjoitetaan mittaukset vielä tiedostoon
tiedosto = open("mittaukset.json", "w")
tiedosto.write(mittauksetJson)
tiedosto.close()
```

Listan tulostus JSON-muodossa näyttää tältä:

```
[
  {
    "aika": 0,
    "paine": 1024,
    "lampotila": 275,
    "kosteus": 50
  },
  {
    "aika": 1,
    "paine": 1025,
    "lampotila": 274,
    "kosteus": 51
  },
  {
    "aika": 2,
    "paine": 1026,
    "lampotila": 273,
    "kosteus": 52
  }
]
```

Myös itse tehdyn luokan olio voidaan sarjallistaa. Python ei kuitenkaan voi tietää, miten kaikki itse määritellyt tietotyypit pitäisi sarjallistaa. Tällöin säännöt sarjallistamiselle täytyy kertoa itse (tätä ei neuvota tässä oppaassa). Jos luokassa on kuitenkin vain yksinkertaisia tietotyyppejä, onnistuu itse määritellyn luokan olion sarjallistaminen varsin helposti. Funktiossa `json.dumps` täytyy kertoa, että olio sarjallistetaan kuten sanakirja. Tämä tapahtuu kutsumalla oliolle `__dict__`-jäsentä sarjallistamisen yhteydessä.

```
olioJson = json.dumps(olio.__dict__)
```

Alla on esimerkki itse tehdyn luokan sarjallistamisesta.

```
import json

class Henkilo:
    def __init__(self, nimi, palkka):
        self.nimi = nimi
        self.palkka = palkka

miihkali = Henkilo('miihkali', 1000)

miihkaliJson = json.dumps(miihkali.__dict__)
print(miihkaliJson) # {"nimi": "miihkali", "palkka": 1000}
```

## 7.2 Sarjallistuksen purkaminen (deserialisointi)

Tiedostoon tallennettu JSON-muotoinen lista sanakirjoista voidaan jäsentää näin.

```
import json

# avataan tiedostoi lukemista varten
tiedosto = open("mittaukset.json", "r")
# deserialisoidaan
mittaukset = json.load(tiedosto)
# tulostetaan listan alkiot
print(mittaukset)
# tulostetaan listan tiedot alkioittain
for mittaus in mittaukset:
    print(mittaus['aika'], mittaus['lampotila'], \
          mittaus['paine'], mittaus['kosteus'])
tiedosto.close()
```

## 8 Socket-ohjelmointi

Asiakas/palvelin-arkkitehtuuri on yleinen ratkaisu verkko-ohjelmoinnissa. Palvelin-sovellus on yleensä palvelinkoneella toimiva sovellus, joka odottaa asiakaskoneella olevien asiakassovellusten pyyntöjä. Kun asiakas ottaa yhteyden palvelimeen, saa se käyttöönsä palvelimen toteuttamat palvelut. Palvelin voi palvella yleensä useaa asiakasta yhtä aikaa.

Tässä oppaassa tutustutaan asiakas/palvelinohjelmointiin aluksi socketien avulla. Asiakas- ja palvelin-ohjelmien välille luodaan ensin TCP-yhteys. Yhteyden avaamisen jälkeen voidaan siirtää halutun protokollan mukaista tietoa asiakkaan ja palvelimen välillä. Ohjelmointiympäristö tarjoaa tuen ainoastaan yhteyden avaamiselle käytölle ja sulkemiselle, mutta kommunikoinnin hallinta jää ohjelmoijan huoleksi.

### 8.1 TCP/IP-protokolla ja IP-osoite

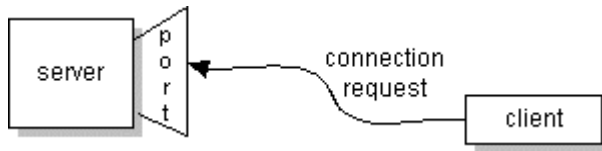
TCP/IP-protokolla koostuu kahdesta eri protokollasta TCP (Transmission Control Protocol) ja IP (Internet Protocol). TCP-protokolla vastaa tiedon luotettavasta siirtämisestä kohteiden välillä. TCP on yhteydellinen protokolla. IP-protokolla hoitaa pakettien reitittämisen internetissä. IP-protokollaan liittyy nelitavuinen IP-osoite. Omaan koneeseen voi viitata IP-osoitteella 127.0.0.1 (localhost).

Yleensä tietokone on yhteydessä verkkoon vain yhdellä yhteydellä. Tätä yhteyttä tarvitsevat kuitenkin samaan aikaan monet eri ohjelmat. Samassa IP-osoitteessa olevat palvelut erotetaan portin avulla. Kutakin palvelua varten varataan yksi tai useampi portti. Portit 0 – 1023 on varattu tiettyyn käyttöön.

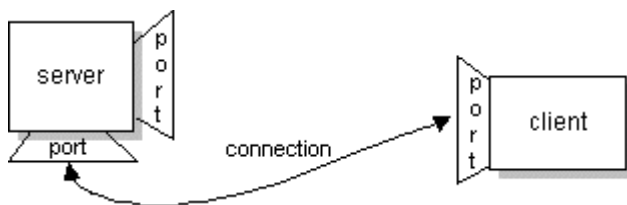
IP-osoitetta käytetään koneen tunnistamisessa. Porttiosoitetta käytetään tunnistamaan haluttu prosessi tässä koneessa. IP-osoitteen ja portin yhdistelmää kutsutaan socket-osoitteeksi. Socket-osoite voi olla esimerkiksi muotoa: 127.0.0.1:8000.

## 8.2 Socket-yhteyden muodostaminen

Kommunikointi alkaa siten, että asiakas valitsee vapaan portin, ottaa yhteyden palvelimen tunnettuun porttiin ja odottaa vastausta.



Palvelin vastaa asiakkaan yhteydenottoon ja antaa asiakkaan käyttöön jonkin omista vapaista porteistaan. Palvelinportti vapautuu odottamaan seuraavaa asiakasta.



Pythonissa socket-moduuli, jota voi käyttää verkko-ohjelmointiin. Socket-ohjelmointi Pythonilla on hyvin samankaltaista, kuin muillakin ohjelmointikielillä.

## 8.3 Asiakas

Alla on esimerkki asiakas-ohjelmasta. Aluksi asiakas-ohjelmassa luodaan socket-olio. Funktio `socket.gethostname` palauttaa oman koneen IP-osoitteen. Oletuksena on tässä, että palvelin on samassa koneessa kuin asiakas.

```
import socket

# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name (127.0.0.1)
host = socket.gethostname()

port = 9999
```

```

# connection to hostname on the port.
s.connect((host, port))

# write command
s.send("LKM".encode('ascii'))

# Receive no more than 1024 bytes
msg = s.recv(1024)

s.close()
print (msg.decode('ascii'))

```

Seuraavaksi asiakas avaa socket-yhteyden palvelinkoneen haluttuun porttiin connect-funktiolla. Parametriksi annetaan palvelinohjelman IP-osoite ja portti.

Tässä esimerkissä oletetaan, että palvelimelle pitää antaa ensin komento, johon se vastaa. Asiakas lähettää komennon "LKM" palvelimelle send-funktiolla ja jää odottamaan vastausta. Kun vastaus saapuu, asiakas lukee sen recv-funktiolla. Kun socket-yhteyttä ei enää tarvita, sulkee asiakas yhteyden close-funktiolla.

Tässä esimerkissä data lähetetään ja vastaanotetaan tietovirtana (socket.SOCK\_STREAM socketin alustuksessa). Tiedot lähetetään ja vastaanotetaan tavuina. Merkkijonot muunnetaan tavulistaksi encode-funktiolla lähetyksessä. Vastaanotossa tavuvirta muutetaan merkkijonoksi decode-funktiolla.

## 8.4 Palvelin

Alla on palvelimen koodi. Socket-olio luodaan samalla tavalla kuin asiakkaan puolella. Funktiossa bind määritellään palvelinohjelman portti, jota kuunnellaan. Samalla pitää antaa myös palvelimen IP-osoite, joka saadaan funktiolla gethostname. Palvelinsovellus kuuntelee omaa palvelinporttiaan ja avaa asiakkaille sen kautta yhteyksiä.

```

import socket

# create a socket object
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9999

# bind to the port
serversocket.bind((host, port))

# queue up to 5 requests
serversocket.listen(5)

i = 0
while True:
    # establish a connection
    clientsocket, addr = serversocket.accept()
    i += 1
    print("Got a connection from %s" % str(addr))

    msg = "EMPTY"
    viesti = clientsocket.recv(1024)
    viesti = viesti.decode('ascii')
    if viesti == "LKM":
        msg = str(i)

    clientsocket.send(msg.encode('ascii'))
    clientsocket.close()

```

Alustusten jälkeen palvelinohjelma jää ikuisen silmukkaan odottamaan uutta yhteyspyyntöä (accept). Kun yhteyspyyntö saadaan, luodaan uusi socket tätä asiakasta varten.

Tässä esimerkissä oletetaan, että asiakas lähettää ensin komennon, jonka palvelin lukee recv-funktiolla. Jos komento on "LKM", vastaa palvelin asiakkaalle, miten monta asiakasta palvelimella on ollut. Viesti lähetetään asiakkaalle send-funktiolla. Lopuksi yhteys lopetetaan close-funktiolla.

Usein kutakin asiakasta varten luodaan uusi säie. Näin palvelinohjelma vapautuu heti palvelemaan uutta asiakasta.

## 8.5 Tiedon välitys JSON-muodossa

Edellisessä esimerkissä asiakkaan palvelimelle välittämä komento ja palvelimen palauttama vastaus oli pelkkää tekstiä. Usein tieto kannattaa välittää kuitenkin JSON-muodossa. JSONia käytettäessä täytyy tiedoston alkuun muistaa lisätä `import json`.

Alla on näytetty osa asiakkaan koodista, kun komento ja vastaus ovat json-muodossa.

```
# connection to hostname on the port.
s.connect((host, port))

# create command
command = { 'command': 'LKM' }
# serialize command
strCommand = json.dumps(command)

# send command
s.send(strCommand.encode('ascii'))

# Receive no more than 1024 bytes
msg = s.recv(1024)

s.close()

# deserialize answer
strAnswer = msg.decode('ascii')
answer = json.loads(strAnswer)
print(strAnswer)
```

Palvelimen koodin loppuosa näyttää tältä:

```
i = 0
while True:
    # establish a connection
    clientsocket, addr = serversocket.accept()

    i+=1
    print("Got a connection from %s" % str(addr))

    in_msg = clientsocket.recv(1024)
    in_msg = in_msg.decode('ascii')
```



```

# luetaan JSON-muotoinen viesti
cmd = json.loads(in_msg)

out_msg = {}
if (cmd['command'] == 'LKM'):
    # tehdään vastaus
    out_msg = { 'LKM': i}
# sarjallistetaan vastaus JSON-muotoon
out_msg = json.dumps(out_msg)

clientsocket.send(out_msg.encode('ascii'))
clientsocket.close()

```

## 8.6 Harjoituksia

1. Lisää edelliseen palvelinohjelmaan toiminto, joka palauttaa palvelimen kellon-ajan (Komento "TIME"). Muuta myös asiakasohjelmaa vastaavasti. Siirrä data JSON-muodossa.
2. Tee palvelinohjelma, joka lähettää vastauksia asiakkaalle jatkuvasti, kun yhteys on avattu.

## 9 Python Flask ja REST API

### 9.1 REST API

HTTP (Hypertext Transfer Protocol eli hypertekstin siirtoprotokolla) on protokolla, jota asiakasohjelmat ja www-palvelimet käyttävät tiedonsiirtoon. HTTP-protokollaa käytettäessä asiakasohjelma avaa TCP-yhteyden palvelimelle ja lähettää pyynnön (request). Palvelin vastaa lähettämällä vastauksen pyyntöön (response). Vastaus voi olla esimerkiksi HTML-sivu tai JSON-dokumentti. Usein asiakasohjelma on selain.

HTTP-protokollassa on useita metodeja, joista yleisimmät ovat:

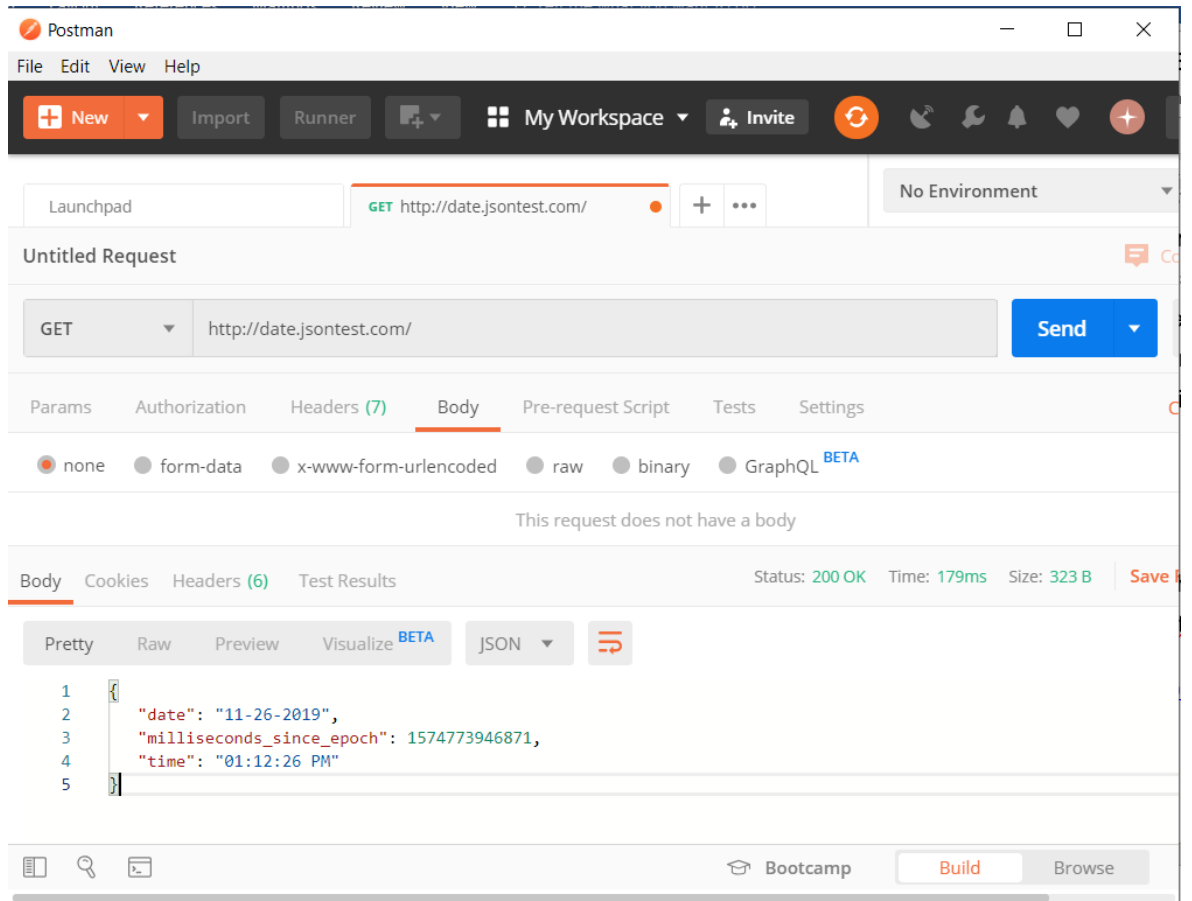
- GET: käytetään resurssin hakua varten (esimerkiksi verkkosivun hakuun)
- POST: tietojen lähettäminen palveluun
- HEAD: sivun otsikkotietojen hakeminen
- PUT: tiedon tallettaminen
- DELETE: tiedon poistaminen

REST (Representational State Transfer) on HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen. REST keskittyy datan siirtämiseen resurssien välillä. REST:ssä data on kommunikaatiossa avainasemassa operaatioiden sijaan. REST:issä kaikki palvelut nähdään resursseina (Resource), joilla on yksilöitävissä oleva URI (Uniform Resource Locator)-tunniste ja yhtenäinen standardi HTTP (Hypertext Transfer Protocol)-protokollan mukainen kutsurajapinta.

### 9.2 Postman

HTTP-pyyntöjen tekemistä REST-pohjaiseen palveluun voi kokeilla helposti Postman-ohjelman avulla, joka on ladattavissa internetistä (download postman).

Lataa Postman ja lähetä sillä GET-pyyntö osoitteeseen <http://date.jsontest.com>.



Palvelu palauttaa päivämäärän ja ajan JSON-muodossa.

### 9.3 Flask

Flask on helppokäyttöinen mikrosovelluskehys web-sovellusten tekemiseen. Flask asennetaan pip-työkalulla: `pip install flask --user`.

Tehdään ensin yksinkertainen Flask-sovellus. Tee uusi hakemisto "hello" testisovellusta varten ja siirry tähän hakemistoon. Asenna Flask terminaalista komennolla `pip install flask --user`. Tee uusi tiedosto `hello.py` ja kirjoita siihen alla oleva ohjelma:

```

from flask import Flask
app = Flask(__name__)

@app.route('/')
def helloworld():
    return ("Hello World")

if __name__ == '__main__':
    app.run(debug = True)

```

Aja ohjelma ja avaa selain. Anna osoiteriville osoite localhost:5000. Selaimelle tulostuu teksti "Hello World".

Funktio route() on dekoraattori, jossa määritellään funktio, jota kutsutaan parametrina annetulla URLilla. Edellisessä esimerkissä kutsutaan funktiota helloworld(), kun URL on '/' eli palvelun kotisivu.

Seuraavassa esimerkissä route()-dekoraattorille on annettu parametrina sääntö '/hello'. Kun käyttäjä valitsee osoitteen http://localhost:5000/hello, kutsutaan funktiota helloworld().

```

from flask import Flask
app = Flask(__name__)

@app.route('/hello')
def helloworld():
    return ("Hello World")

if __name__ == '__main__':
    app.run(debug = True)

```

URL voi sisältää myös muuttujia, jotka välitetään route()-dekoraattorin parametrin kautta lisäämällä siihen muuttujaosa.

```

from flask import Flask
app = Flask(__name__)

@app.route('/hello/<name>')
def helloworld(name):
    return ("Hello {:s}".format(name))

if __name__ == '__main__':
    app.run(debug = True)

```

Kun selaimen osoiteriville annetaan URL <http://localhost:5000/hello/Petteri>, tulostuu teksti 'Hello Petteri'.

Muuttujaosa on oletuksena string-tyyppiä, mutta se voidaan määritellä myös int-, float- ja path-tyyppiseksi. Seuraavassa esimerkissä on dekoraattori, jossa muuttujaosa on int-tyyppinen.

```

from flask import Flask
app = Flask(__name__)

@app.route('/hello/<name>')
def helloworld(name):
    return ("Hello {:s}".format(name))

@app.route('/number/<int:id>')
def numberinput(id):
    return ("ID {:d}".format(id))

if __name__ == '__main__':
    app.run(debug = True)

```

Funktiota `url_for()` voi käyttää URL:n rakentamiseen dynaamisesti, mitä on havainnollistettu seuraavassa esimerkissä.

```

from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/admin')
def adminuser():
    return ('Hello administrator')

@app.route('/user/<name>')
def normaluser(name):
    return ("Hello {:s}".format(name))

@app.route('/user/<username>')
def hellouser(username):
    if username == 'admin':
        return redirect(url_for('adminuser'))
    else:
        return redirect(url_for('normaluser', guenamest = username))

if __name__ == '__main__':
    app.run(debug = True)

```

## 9.4 Template

Flask-ohjelmoinnissa voi käyttää Jinja2-templatea HTML-sivun näyttämiseen. Templaten avulla voidaan sisällyttää Python-koodia HTML-koodin sisälle.

Tee templates-hakemistoon tiedosto hello.html ja kopioi sinne alla oleva HTML-koodi.

```

<!doctype html>
<html>
  <body>
    <h1>Hello {{ name }}!</h1>
  </body>
</html>

```

Tee sitten seuraava Flask-ohjelma:

```

from flask import Flask, render_template
app = Flask(__name__)

@app.route('/user/<username>')
def hello_name(username):
    return render_template('hello.html', name = username)

if __name__ == '__main__':
    app.run(debug = True)

```

Yllä olevassa esimerkissä kopioidaan URLissa annettu username-muuttuja HTML-sivun h1-elementin sisälle. Kokeile ohjelmaa antamalla selaimen osoiteriville localhost:5000/user/Nimi.

Sanakirjan sisältö voidaan näyttää templatien avulla HTML-taulukossa seuraavasti:

```

<!doctype html>
<html>
  <body>
    <table border = 1>
      {% for key, value in result.items() %}
        <tr>
          <th> {{ key }} </th>
          <td> {{ value }} </td>
        </tr>
      {% endfor %}
    </table>
  </body>
</html>

```

For-lauseen ympärillä on merkintä {%...%} ja muuttujien arvojen ympärillä merkintä {{...}}.

Sanakirjan näyttämistä voi kokeilla seuraavalla ohjelmalla:

```

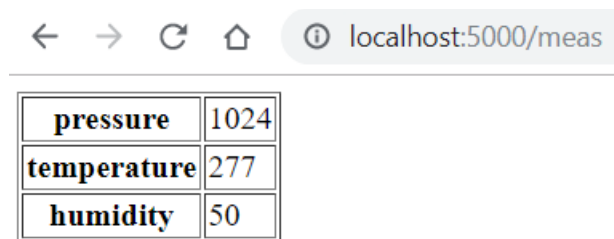
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/meas')
def result():
    measurement = {'pressure':1024,'temperature':277,'humidity':50}
    return render_template('table.html', result = measurement)

if __name__ == '__main__':
    app.run(debug = True)

```

Tuloksena syntyvä HTML-sivu näyttää nyt tältä:



The screenshot shows a web browser window with the address bar displaying 'localhost:5000/meas'. Below the address bar, there is a table with three rows and two columns. The first row has 'pressure' and '1024'. The second row has 'temperature' and '277'. The third row has 'humidity' and '50'.

pressure	1024
temperature	277
humidity	50

## 9.5 Esimerkki

Tehdään ohjelma, joka tulostaa alla olevan listan tiedot www-sivulle. Lista sisältää sanakirjoja polkupyörien tiedoista.

```

products = [
    { 'id' : '1', 'name' : 'Gary Fisher', 'category' : 'MTB', 'price' : '1000'},
    { 'id' : '2', 'name' : 'Trek Domane', 'category' : 'Road', 'price' : '2000'},
    { 'id' : '3', 'name' : 'Canyon', 'category' : 'MTB', 'price' : '2000'},
    { 'id' : '4', 'name' : 'Specialized', 'category' : 'CX', 'price' : '1000'},
]

```

Tee ensin tiedosto result.html templates-kansioon:



```

<!doctype html>
<html>
  <body>
    <table border = 1>
      {% for p in result %}
        <tr>
          <th> {{ p['id'] }} </th>
          <td> {{ p['price'] }} </td>
          <td> {{ p['name'] }} </td>
        </tr>
      {% endfor %}
    </table>

  </body>
</html>

```

HTML-sivulla näytetään listan result alkiot, jotka ovat sanakirjoja. Lista käydään läpi for-lauseessa.

Tee uusi tiedosto products.py ja kopioi siihen alla oleva ohjelmakoodin runko.

```

import json
from flask import Flask, render_template, request, Response
app = Flask(__name__)

products = [
    { 'id' : '1', 'name' : 'Gary Fisher', 'category' : 'MTB', 'price' : '1000'},
    { 'id' : '2', 'name' : 'Trek Domane', 'category' : 'Road', 'price' : '2000'},
    { 'id' : '3', 'name' : 'Canyon', 'category' : 'MTB', 'price' : '2000'},
    { 'id' : '4', 'name' : 'Specialized', 'category' : 'CX', 'price' : '1000'},
]
if __name__ == '__main__':
    app.run(debug = True)

```

Lisää seuraavaksi dekoraattori ja funktio, joka näyttää listan tiedot HTML-sivulla taulukossa.

```

# avaa sivun result.html ja näyttää tuotteet siinä
@app.route('/products2/')
def get_all_products2():
    return render_template('result.html', result = products)

```

HTML-sivu näytetään, kun käyttäjä antaa selaimessa osoitteen localhost:5000/products2. Funktio get\_all\_products2() näyttää HTML-sivun result.html.

## 9.6 POST -metodin käyttö HTML-sivulta

Oletusarvoisesti Flask käyttää GET-metodia. Katsotaan seuraavaksi esimerkkiä, jossa siirretään tietoa HTML-lomakkeelta palvelimelle. Tällöin käytetään POST-metodia.

Tee ensin templates kansio ja sinne tiedosto user.html. Kopioi seuraava koodi tiedostoon.

```
<html>
  <body>
    <form action = "http://localhost:5000/user" method = "post">
      <p>Enter Name:</p>
      <p><input type = "text" name = "username" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>
```

Form-elementin method-argumentissa on määritelty, että käytetään POST-metodia.

Tee seuraavaksi alla oleva ohjelma:

```

from flask import Flask, redirect, url_for, request, render_template
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('user.html')

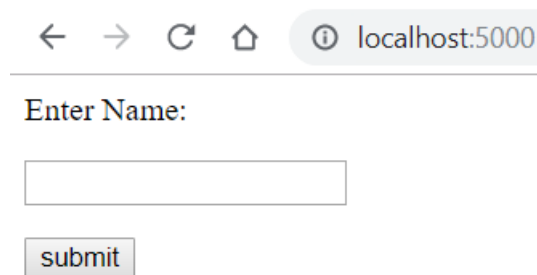
@app.route('/success/<name>')
def success(name):
    return ("Hello {:s}".format(name))

@app.route('/user', methods = ['POST'])
def userinput():
    user = request.form['username']
    return redirect(url_for('success', name = user))

if __name__ == '__main__':
    app.run(debug = True)

```

Kun käyttäjä siirtyy selaimessa osoitteeseen localhost:5000, tulee näytölle alla oleva lomake.



Tällöin kutsutaan funktiota `index()`, joka näyttää html-sivun `user.html`. Kun käyttäjä antaa lomakkeella nimen ja painaa submit-painiketta, kutsutaan funktiota `userinput()`. Vastaavassa dekoraattorissa on määritelty URL  `'/user'` ja HTTP-metodiksi POST (vrt. HTML-sivu).

## 9.7 REST API

Yleisimmät HTTP-metodit (GET, PUT, POST ja DELETE) vastaavat CRUD operaatioita:

- POST: tietojen lähettäminen palveluun (Create)

- GET: käytetään resurssin hakua varten (Read)
- PUT: tiedon päivittäminen (Update)
- DELETE: tiedon poistaminen (Delete)

Tehdään seuraavaksi ohjelma, joka toteuttaa CRUD-operaatiot. Aloitetaan aiemmasta harjoituksesta `products.py`, joka näyttää aluksi tältä. Tiedosto `result.html` on `templates`-kansiossa.

```
import json
from flask import Flask, render_template, request, Response
app = Flask(__name__)

products = [
    { 'id' : '1', 'name' : 'Gary Fisher', 'category' : 'MTB', 'price' : '1000'},
    { 'id' : '2', 'name' : 'Trek Domane', 'category' : 'Road', 'price' : '2000'},
    { 'id' : '3', 'name' : 'Canyon', 'category' : 'MTB', 'price' : '2000'},
    { 'id' : '4', 'name' : 'Specialized', 'category' : 'CX', 'price' : '1000'},
]
# avaa sivun result.html ja näyttää tuotteet siinä
@app.route('/products2/')
def get_all_products2():
    return render_template('result.html', result = products)

if __name__ == '__main__':
    app.run(debug = True)
```

Ohjelmaan lisätään funktiot, joilla voi lisätä uusia tietoja sekä poistaa ja päivittää niitä. Lisäksi tehdään funktioita, joilla voi hakea tietoja. Näitä funktioita voi kutsua REST API:n kautta erilaisista asiakasohjelmista.

Lisätään seuraavaksi dekoraattori ja funktio, joka palauttaa tuotteiden tiedot JSON-muodossa asiakkaalle, kun asiakas tekee GET-pyyynnön osoitteeseen `localhost:5000/products`. Vastauksen lähettämisessä käytetään apuna `Response`-luokkaa.

```
# palauttaa tuotteet JSON-muodossa
@app.route('/products/')
def get_all_products():
    s = json.dumps(products, indent=True)
    resp = Response(s, status=200, mimetype='application/json')
    return resp
```

Yksittäisen tuotteen tiedot voidaan hakea seuraavasti:

```
# palauttaa yksittäisen tuotteen tiedot id:n perusteella
@app.route('/products/<int:id>')
def get_product(id):
    s = json.dumps(products[id], indent=True)
    resp = Response(s, status=200, mimetype='application/json')
    return resp
```

Tuotteen lisääminen tehdään POST-metodin avulla. Alla oleva ohjelma poimii JSON-muodossa olevat tiedot pyynnöstä ja lisää tiedot taulukkoon. POST-metodi palauttaa lisätyn tietueen asiakkaalle.

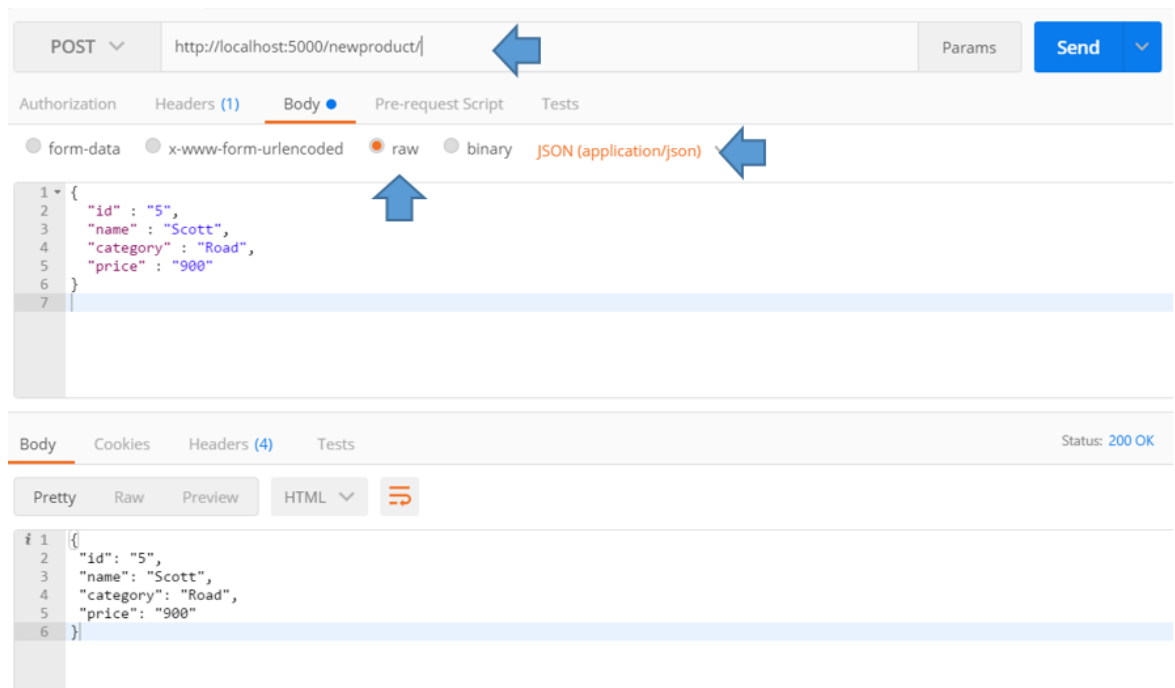
```
@app.route('/newproduct/', methods = ['POST'])
def new_product():
    # lisää uusi tuote taulukkoon
    p = request.get_json(force=True)
    products.append(p)
    return json.dumps(p, indent=True)
```

Lisää vielä toiminnot tietueen poistamiseksi (DELETE) ja päivittämiseksi (UPDATE).

Kokeile seuraavaksi lisätä uusi tuote Postman-ohjelman avulla:

- Metodi: POST
- URL: <http://localhost:5000/newproduct>
- Body
- raw
- JSON (application/json)

Asetukset ovat myös seuraavassa kuvassa.



Tarkista vielä, että uusi tuote näkyy HTML-sivulla.

## 9.8 Requests

Tehdään seuraavaksi konsolisovellus, joka kutsuu vastaavia HTTP-metodeja kuin selain. Tämä sovellus kutsuu GET- ja POST-metodeja Python-ohjelmasta hyödyntäen Requests-kirjastoa.

HTTP-pyyynnön lähettäminen Requests-kirjaston avulla on helppoa. Ensin asennetaan Requests-kirjasto pip:n avulla:

```
pip install requests --user
```

Seuraavassa ohjelmassa on esimerkki, jossa on tehty POST- ja GET-pyynnöt konsolisovelluksesta.

```
import json
import requests

product = {
    "id" : "111",
    "name" : "Specialized",
    "category" : "MTB",
    "price" : "500"
}

# sarjallista
s = json.dumps(product)

# lähetä uuden tuotteen tiedot palvelimelle POST:lla
response = requests.post('http://localhost:5000/newproduct/', data = s)
print(response)

# hae kaikki tuotteet GET:llä
response = requests.get('http://localhost:5000/products/')
print(response.json())
```

Käynnistä edellä tehty palvelinohjelma products.py. Käynnistä sitten asiakasohjelma.

## 9.9 Tehtävä

Tee ensin palvelinohjelma, joka näyttää HTML-sivulla mittautustietoja taulukossa. Tee sitten asiakasohjelma, joka lähettää mittautustietoja palvelinohjelmalle sekunnin välein.

Ota mallia asiakasohjelmaan Products-harjoituksesta.

Esimerkki asiakasohjelman mittautusten generoinnista on alla.

```
import json
import requests
import math

i = 0

while i < 20:
    measurement = {}
    measurement["id"] = i
    measurement["pressure"] = 1024 * math.sin(i/10.0)
    measurement["temperature"] = 300 + math.cos(i/5.0)
    measurement["humidity"] = 33 + math.cos(i/6.0)

    # muunna mittaus json-muotoon ja lähetä POST:lla

    i += 1
```

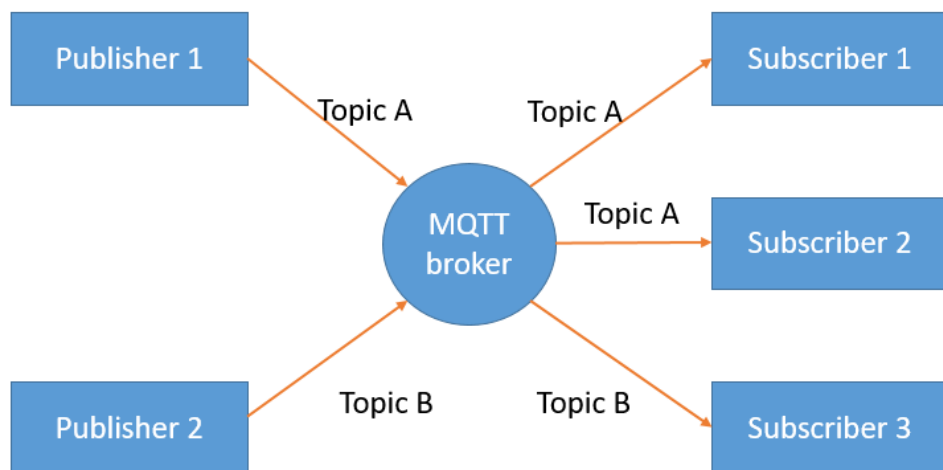
## 9.10 Socket.io



## 10MQTT

MQTT-protokollaa (Message Queue Telemetry Transport) käytetään yleisesti IoT-sovelluksissa. Nimensä mukaisesti se sopii telemetriadatan keräämiseen, etämittaukseen ja valvontaan. MQTT-protokollaa käytetään myös monissa IoT-alustoissa datan vastaanottamiseen.

MQTT perustuu julkaisija-tilaaja -malliin (publisher/subscriber). Datat tuottajat ovat julkaisijoita. Tilaajia voi olla useita ja ne voivat tilata monen julkaisijan dataa. Julkaisija ja tilaaja eivät ole suoraan yhteydessä toisiinsa, vaan niiden välissä on MQTT-välityspalvelin eli broker. Julkaisijan ja tilaajan ei tarvitse siis olla tietoisia toisistaan.



Yllä olevassa kuviossa julkaisija 1 lähettää viestiä Topic A ja julkaisija 2 viestiä Topic B MQTT-brokerille. Tilaajat 1 ja 2 tilaavat MQTT-brokerilta viestin Topic A. Tilaaja 3 tilaa viestin Topic B. MQTT ei sisällä viestijonoja, joten datan tilaajan on oltava alusta asti kuuntelemassa välityspalvelinta, jos se haluaa vastaanottaa julkaisijan lähettämän datan alusta saakka.

MQTT-protokollaa testattaessa voi käyttää valmiita testipalvelimia, kuten [broker.hivemq.com](https://broker.hivemq.com). MQTT brokerin voi asentaa myös omalle tietokoneelle (MQTTbox, MQTT.fx).

## 10.1 Julkaisija

Alla on esimerkki ohjelmasta, joka toimii julkaisijana (publisher). Ohjelma generoi mittausdataa, sarjallistaa sen JSON-muotoon ja lähettää datan MQTT brokerille.

```
import paho.mqtt.client as mqtt
import time, json, math

#broker_address="test.mosquitto.org"
broker_address="broker.hivemq.com"

client = mqtt.Client("MeasurementPublisher") # luo uusi asiakas

client.connect(broker_address) # avaa yhteys brokerille
print("ok")

for i in range(10):
    measurement = { }
    measurement['id'] = i
    measurement['pressure'] = 1024 + math.sin(i/10.0)
    measurement['temperature'] = 300 + 3 * math.cos(i/5.0)
    measurement['humidity'] = 33 + math.cos(i/6.0)

    # muunna json-muotoon ja lähetä MQTT brokerille
    s = json.dumps(measurement)
    # "meas" on topic ja s on lähetettävä daat
    client.publish("meas", s)
    print(s)
    time.sleep(2)
```

## 10.2 Tilaaaja

Alla on esimerkki ohjelmasta, joka toimii tilaajana (subscriber).

```
import paho.mqtt.client as mqtt
import time

# callback-funktio, jota kutsutaan, kun uusi tilattu viesti saapuu
def on_message(client, userdata, message):
    print("message received " ,str(message.payload.decode("utf-8")))
    print("message topic=",message.topic)
    print("message qos=",message.qos)
    print("message retain flag=",message.retain)

broker_address="broker.hivemq.com"

client = mqtt.Client("MeasurementSubscriber") # avaa yhteys brokerille
client.on_message=on_message # asetetaan callback-funktio

client.connect(broker_address) #avataa yhteys brokerille
client.loop_start() # aloitetaan kuuntelusilmukka

# tilataan viesti, jonka topic on "meas"
client.subscribe("meas")

time.sleep(100) # odotetaan 100 s
client.loop_stop() # lopetetaan kuuntelu
```