Funksjoner og if-tester

Ole Christian Lingjærde, Institutt for Informatikk, UiO

6 September - 12 September, 2021 (Del 2 av 2)

Forrige forelesning på en foil

- **Lister:** for å lagre flere verdier i ett dataobjekt
- **Funksjoner:** navngi kodebiter slik at de kan brukes flere steder
- **Funksjonskall:** kalle på funksjon = anvende funksjon
- **Argumenter og returverdier:** input og output til funksjoner

Tema i dag

- Mer om funksjoner
- Assert-tester
- Testfunksjoner

Quiz 1

Hva skriver dette programmet ut?

```
def f():
  print("Hello world")

f()
```

Quiz 2

Hva skriver dette programmet ut?

```
def f():
    verdi = 15
    return verdi
print(f())
```

Quiz 3

Hva skrives ut her?

```
def f(x):
    y = 2*x - 1
    return y

lengde = 15
print(f(lengde))
```

Quiz 4

Hva skrives ut her?

```
def f(x):
    y = 2*x - 1
    return y

def g(x):
    verdi = 2*f(x)
    return verdi

dist = 5
print(g(dist))
```

Quiz 5

Hva skriver dette programmet ut?

```
def partall(n):
    a = list(range(0,n,2))
    return a

def g(k,n):
    verdi = partall(n)[k]
    return verdi

print(g(3,20))
```

Quiz 6

Klarer du å se hva denne funksjonen gjør?

```
def f(n):
    v = 1
    for i in range(2,n+1):
        v = v * i
    return v
```

Et kall på en funksjon må ha riktig antall argumenter

Anta at vi har definert en funksjon

```
def fnk(x, y, z):
  return x*y*z
```

Når vi skal bruke funksjonen senere i samme program, må vi passe på å få med riktig antall argumenter i kallet:

```
print(fnk())              # Gir feilmelding
print(fnk(3))             # Gir feilmelding
print(fnk(3, 0))          # Gir feilmelding
print(fnk(3, 0, 5))       # OK
print(fnk(3, 0, 5, 1))    # Gir feilmelding
```

Argumenter med standard-verdier (default values)

Det er et unntak fra regelen på forrige slide. Når vi definerer en funksjon kan vi velge å gi en standard (default) verdi til noen av inputargumentene:

```
def fnk(x, y, z=0, w=2):
    return x + y + z + w
```

Her har de to siste argumentene fått standard-verdier. Vi kan da selv velge om vi vil gi verdier til disse argumentene når vi bruker funksjonen:

```
verdi = fnk(3)                  # Gir feilmelding
verdi = fnk(3, 5.2)             # OK (nå blir z=0 og w=2)
verdi = fnk(3, 5.2, 0.1)        # OK (nå blir z=0.1 og w=2)
verdi = fnk(3, 5.2, 0.1, 18)    # OK (nå blir z=0.1 og w=18)
```

VIKTIG: Argumentene med standard-verdier må stå sist i listen over input-argumenter.

Å navngi argumenter i et funksjonskall

Vi har sett at vi kan oppgi argumentverdier når vi *definerer* funksjoner. Omvendt kan vi også oppgi argumentnavn når vi *bruker* funksjoner:

```
def f(x,y):
    return x**2 - 2*x*y + y**2

z = f(3, 4)          # Ingen argumentnavn
z = f(3, y=4)        # Ett argumentnavn
z = f(x=3, y=4)      # To argumentnavn
z = f(y=4, x=3)      # To argumentnavn
```

VIKTIG: Når vi oppgir argumentnavn må vi sørge for at disse står sist i listen over inputverdier til funksjonen.

Eksempel

Anta at vi har en funksjon av t, med parametre A, a og ω som i praksis ofte vil ha verdiene $A = a = 1$ og $\omega = 2\pi$:

$$f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)$$

Mulig implementasjon i Python:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)
```

Her har vi gitt A, a og ω standard-verdier. Det gir oss mange muligheter når vi kaller på funksjonen:

```
v1 = f(0.2)                          # Bare oppgi t
v2 = f(0.2, omega=1)                 # Endre omega
v3 = f(0.2, omega=1, A=2.5)          # Endre omega og A
v4 = f(A=5, a=0.1, omega=1, t=1.3)   # Endre alle parametre
v5 = f(0.2, 1, 2.5)                  # Endre A og a
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)
r = h(0, 1)
r = h(0, 1, 2)
r = h(x=0, 1, 2)
r = h(0, y=1)
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    # Ikke lovlig (mangler y)
r = h(0, 1)
r = h(0, 1, 2)
r = h(x=0, 1, 2)
r = h(0, y=1)
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)
r = h(x=0, 1, 2)
r = h(0, y=1)
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)
r = h(0, y=1)
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # Ikke lovlig (navngitte må komme sist)
r = h(0, y=1)
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            #Ikke lovlig
r = h(0, y=1)               #Lovlig
r = h(0, 1, z=3)
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            #Ikke lovlig
r = h(0, y=1)               #Lovlig
r = h(0, 1, z=3)            # Lovlig
r = h(0, 0, x=0)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            #Ikke lovlig
r = h(0, y=1)               #Lovlig
r = h(0, 1, z=3)            # Lovlig
r = h(0, 0, x=0)            # Ikke lovlig (x angis to ganger)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                    #Ikke lovlig
r = h(0, 1)                 #Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            #Ikke lovlig
r = h(0, y=1)               #Lovlig
r = h(0, 1, z=3)            # Lovlig
r = h(0, 0, x=0)            # Ikke lovlig
r = h(z=0, x=1)             # Ikke lovlig (y mangler)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                # Ikke lovlig
r = h(0, 1)             #Lovlig
r = h(0, 1, 2)          # Lovlig
r = h(x=0, 1, 2)        # Ikke lovlig
r = h(0, y=1)           #Lovlig
r = h(0, 1, z=3)        # Lovlig
r = h(0, 0, x=0)        # Ikke lovlig
r = h(z=0, x=1)         # Ikke lovlig
r = h(z=0, x=1, y=2)    #Lovlig
```

Funksjoner kan ha funksjoner som argumenter

Et argument til en funksjon kan selv være en funksjon.

```
def fnk(g):
  return g(0)

from math import cos
verdi = fnk(cos)
print(f"Cos(0) = {verdi}")
```

VIKTIG: Når vi utfører `fnk(cos)` så kaller vi på funksjonen `fnk` med *selve funksjonen* `cos` som inputargument, *ikke* verdien til `cos(x)` for en bestemt `x`.

Eksempel

Eksempel: Vi kan estimere den annenderiverte til en funksjon f i et gitt punkt x med denne formelen:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

hvor $h > 0$ er et lite tall (f.eks. h=0.000001).

Her er en funksjon som beregner den annenderiverte i et punkt x med formelen over:

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/(h*h)
    return r
```

En liten digresjon

Funksjonen vi nettopp definerte, har standard-verdien `h=1E-6`. Er det en grunn til å velge $h = 0.000001$ i stedet for en lavere eller høyere verdi?

- Matematisk blir approksimasjonen bedre når h blir mindre.
- Men vi også ta hensyn til avrundingsfeil.
- Noen numeriske problemer er mer følsomme for avrundingsfeil enn andre, så i praksis blir det ofte litt prøving og feiling for å finne en god h-verdi.

Effekten av å endre h

For å studere effekten av å endre `h` skriver vi et lite program:

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r

def g(t):
    return t**(-6)

# Beregn g''(t) for mindre og mindre verdier av h:
for k in range(1,14):
    h = 10**(-k)
    print (f"h = {h:.0e}: {diff2(g, 1, h):.5f}")
```

Output ($g''(1) = 42$)

```
h = 1e-01: 44.61504
h = 1e-02: 42.02521
h = 1e-03: 42.00025
h = 1e-04: 42.00000
h = 1e-05: 41.99999
h = 1e-06: 42.00074
h = 1e-07: 41.94423
h = 1e-08: 47.73959
h = 1e-09: -666.13381
h = 1e-10: 0.00000
```

For små h-verdier dominerer avrundingsfeil

For $h < 10^{-8}$ er resultatene helt feil!

- **Problem 1:** for veldig små h subtraherer vi tall som er nesten like; dette gir avrundingsfeil.
- **Problem 2:** for små h deler vi avrundingsfeilen på et veldig lite tall (h^2), og dette forsterker feilen.

Mulig løsning: bruke desimaltall med flere sifre

- Python har en (langsom) flyttalls datatype (`decimal.Decimal`) hvor antall sifre er vilkårlig stort.
- Bruk av 25 sifre gir nøyaktige resultater for $h \leq 10^{-13}$

I praksis er det sjelden behov for høyere nøyaktighet.

Lambda-funksjoner

Noen ganger trenger vi å lage funksjoner som bare beregner enkle uttrykk. Da er *lambda-funksjoner* ofte et nyttig alternativ til vanlige funksjonsdefinisjoner.

Eksempel: funksjonen

```
def f(x,y):
    return x**2 - y**2
```

kan defineres på én linje med lambda-konstruksjonen:

```
f = lambda x, y: x**2 - y**2
```

Lambdafunksjoner kan brukes direkte som argumenter i funksjoner:

```
z = g(lambda x, y: x**2 - y**2, 4)
```

Funksjoner bør dokumenteres

Vi kan lage en *doc string* som plasseres rett etter funksjons-headeren og i triple anførselstegn.

```
def C2F(C):
    """Konverter Celsiusgrader (C) til Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Beregn koeffisientene a og b i uttrykket for en rett
    linje y = a*x + b sompasserer gjennom (x0,y0) og (x1,y1)

    x0, y0: første punkt (floats).
    x1, y1: andre punkt (floats).
    return: a, b (floats) for linjen (y=a*x+b).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```

assert: en spesiell form for test

Noen ganger ønsker man å stoppe programkjøringen og gi en feilmelding dersom en bestemt betingelse ikke er oppfylt. For dette formål har vi `assert`.

```
assert x > 0, "x må være positiv"
```

Når setningen over utføres, skjer følgende:

- Hvis $x > 0$ så skjer ingenting
- Hvis $x \leq 0$ så stopper programmet med feilmeldingen "x må være positiv"

Testfunksjoner

Vi skal nå se hvordan `assert` brukes i praksis. Anta at du har skrevet en funksjon som beregner et eller annet og returnerer svaret. Hvordan kan vi vite om svaret er det vi ønsker?

Teststrategi:

- Anta at vi har laget en funksjon `f(x)` og at vi ønsker å vite at kallet `y = f(x)` gir riktig svar.
- Vi lager en *testfunksjon* i Python som kaller på `f(x)` med noen utvalgte verdier for `x` der vi vet hva output `y` skal være (vi har fasiten).
- Hvis output ikke stemmer med fasiten, skal testfunksjonen gi en feilmelding.

Krav til testfunksjoner

En testfunksjon vil *ikke* gi noe utskrift på skjermen hvis funksjonen som testes består alle testene. Hvis en av testene feiler, vil det komme en feilmelding (`AssertionError`).

Regler for testfunksjoner:

- Hvis navnet på funksjonen som skal testes er `XXX` så skal navnet på testfunksjonen være `test_XXX`.
- En testfunksjon har ingen argumenter
- En testfunksjon må ha en `assert success, message` setning, hvor `success` er `True` hvis testen består og `False` ellers. Siste argument `message` kan droppes.

Eksempel 1

```
# Funksjon som skal finne produktet av alle verdiene i en liste
def prod(a):
    result = 1
    for e in a:
            result = result * e
    return result

# Vi lager en testfunksjon:
def test_prod():
    """Testfunksjon for prod."""
    a = [3,1,5]                 # Eksempel på inputverdi for a
    computed = prod(a)          # Faktisk output fra prod
    expected = 15               # Forventet output fra prod
    success = (computed == expected)
    message = "prod gir feil svar!"
    assert success, message

# Vi kaller på testfunksjonen:
test_prod()
```

Eksempel 2

I forrige eksempel testet vi bare om `prod` fungerte for én bestemt liste `[3,1,5]`. Vi kan bedre testen ved å legge inn flere test-cases:

```
def prod(a):
    result = 1
    for e in a:
            result = result * e
    return result

# Vi lager en testfunksjon:
def test_prod():
    """Testfunksjon for prod."""
    inputs = [[3,1,5], [2,4,5], [1,0]]   # Inputlister
    answers = [15, 40, 0]                # Ønsket output
    for i in range(len(inputs)):
        computed = prod(inputs[i])
        expected = answers[i]
        success = (computed == expected)
        message = "prod gir feil svar"
        assert success, message

# Vi kaller på testfunksjonen:
test_prod()
```

Eksempel 3

En liten variant av forrige eksempel er å endre litt på løkken og bruke `zip` til å plukke ut matchende input- og answer-verdier:

```
def prod(a):
    result = 1
    for e in a:
            result = result * e
    return result

# Vi lager en testfunksjon:
def test_prod():
    """Testfunksjon for prod."""
    inputs = [[3,1,5], [2,4,5], [1,0]]  # Inputlister
    answers = [15, 40, 0]                # Ønsket output
    for inp, expected in zip(inputs, answers):
        computed = prod(inp)
        success = (computed == expected)
        message = "prod gir feil svar"
        assert success, message

# Vi kaller på testfunksjonen:
test_prod()
```

Eksempel 4

En svakhet med alle testfunksjonene vi har sett på til nå, er at de tester om `computed` og `expected` er *helt identiske*. I praksis kan det være små avrundingsfeil i `computed` som gjør at `computed == expected` blir False selv om svaret egentlig er riktig.

For å unngå dette, kan vi i stedet sjekke om avstanden mellom `computed` og `expected` er veldig liten.

```
def test_prod():
    """Testfunksjon for prod."""
    inputs = [[3.1, 1.1, 5.2], [2.2, 4.0, 5.9]]  # Inputlister
    answers = [17.732, 51.92]                     # Ønsket output
    tolerance = 1e-10
    for inp, expected in zip(inputs, answers):
        computed = prod(inp)
        success = abs(computed-expected) < tolerance
        message = "prod gir feil svar"
        assert success, message

# Vi kaller på testfunksjonen:
test_prod()
```

Hvordan kjøres testfunksjoner i praksis?

- **Fra programmet:** Vi kan legge inn et kall på testfunksjonen i samme program som funksjonen er definert.
- **Fra ipython:** Hvis vi kjører Python interaktivt via ipython kan vi kalle på testfunksjonen derfra.
- **Fra kommandolinjen:** Hvis vi står på samme filkatalog (directory) som programfilen, kan vi gi kommandoen `pytest prog.py` for å kjøre alle testfunksjoner i programfilen `prog.py`.
- **Forenklet testing:** Vi kan gi kommandoen `pytest` (uten argumenter) for å kjøre alle testfunksjoner i alle Python-programmer med navn som starter på `test_`.

Enhetstesting

I praksis må vi teste store programmer for feil på flere ulike måter. En måte er å sjekke at hver enkelt funksjon i programmet fungerer (droppes i praksis for de aller enkleste funksjonene). Det kalles enhetstesting. Programmet må da passere alle enhetstestene.

Sluttkommentar om testfunksjoner

- Å bevise at et program oppfører seg korrekt for alle tenkelige input er generelt svært vanskelig.
- Å vise at programmet oppfører seg korrekt for *noen* inputverdier er et skritt på veien og kan ofte være tilstrekkelig.
- Det at en suksessfull test ikke gir noe output kan være irriterende - men bruker du pytest så får du beskjed om hvilke testfunksjoner som er kjørt.

Lokale variabler i en funksjon

Variabler som defineres inni en funksjon kalles *lokale variabler*. De finnes bare så lenge funksjonen utføres og er usynlige på utsiden av funksjonen.

```
g = 4                                   # Global variabel
print(g)

def f(t):
    g = 9.81                            # Lokal variabel
    v0 = 0.15                           # Lokal variabel
    return v0*t - 0.5*g*t**2

print(f(1))
print(g)
```

Merk at de to print(g)-setningene vil skrive ut det samme. Tilordningen til `g` inni funksjonen `f(t)` endrer ikke den globale variabelen `g`.

Funksjonsargumenter er lokale variabler

Også argumentene i en funksjon er lokale variabler. De er kun synlige på innsiden av funksjonen.

```
def g(s,t):
    c0 = 1.05
    res = (s+t+c0)**2
    return res

y = g(3.5, 5.0)
```

Vi kan tenke oss at kallet `g(3.0, 5.0)` fører til at disse setningene utføres:

```
s = 3.5
t = 5.0
c0 = 1.05
res = (s+t+c0)**2
return res
```

Her er `s`, `t`, `c0` og `res` lokale variabler i funksjonen.

Globale variabler

- Variabler definert utenfor funksjoner kalles globale.
- Globale variabler er også synlige inni funksjoner.
- Eksempel:

```
def skrivut():
    print(f"alpha = {alpha:g}")

alpha = 10          # Global variabel
skrivut()           # Utskrift: alpha = 10
```

Lokale variabler

- Variabler definert inni funksjoner kalles lokale.
- De er kun synlige inni funksjonen.
- De eksisterer bare når funksjonen utføres!
- Eksempel:

```
def skrivut():
    alpha = 0.6323                  # LOKAL variabel
    print(alpha)                    # Skriver ut alpha

skrivut()                           # Utskrift: 0.6323
print(alpha)                        # FEILMELDING
```

Lokale variabler kan skjule globale variabler

- Vær varsom hvis en lokal og global variabel har samme navn!
- Bare den lokale variabelen er synlig inni funksjonen
- Bare den globale variabelen er synlig utenfor funksjonen
- Eksempel:

```
def g(t):
    alpha = 1.0                 # LOKAL variabel
    return alpha * t

alpha = 100000                  # GLOBAL variabel
print(g(2))                     # Utskrift: 2.0
```

Funksjoner vs hovedprogram

Hovedprogrammet er den delen av programmet som ikke ligger inni noen funksjon. Generelt:

- Programeksekveringen starter med første programsetning i hovedprogrammet og fortsetter linje for linje, fra topp til bunn.
- Funksjoner utføres bare når man kaller på dem.

Merk: funksjoner kan kalles fra hovedprogrammet *eller* fra en funksjon. Dette kan noen ganger føre til lange "kjeder" av funksjonskall.

Plenumsoppgaver

Exercise 2.15: Index a nested list

We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

a) Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`.

b) We can visit all elements of `q` using this nested `for` loop:

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`?

Filename: `index_nested_list`.

Plenumsoppgaver

Exercise 3.20: Write functions

Three functions, `hw1`, `hw2`, and `hw3`, work as follows:

```
>>> print hw1()
Hello, World!
>>> hw2()
Hello, World!
>>> print hw3('Hello, ', 'World!')
Hello, World!
>>> print hw3('Python ', 'function')
Python function
```

Write the three functions.

Filename: `hw_func`.

Exercise 3.23: Wrap a formula in a function

Implement the formula (1.9) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters ρ, K, c, and T_w can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T_o = 25$ C).

Filename: `egg_func`.

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_o - T_w}{T_y - T_w} \right] . \tag{1.9}$$

Plenumsoppgaver

Exercise 3.28: Find the max and min elements in a list
Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. Write your own `max` and `min` functions.

Hint Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, set `max_elem` equal to that element. Use a similar technique to compute the minimum element.
Filename: `maxmin_list`.