Kapittel 6: funksjoner

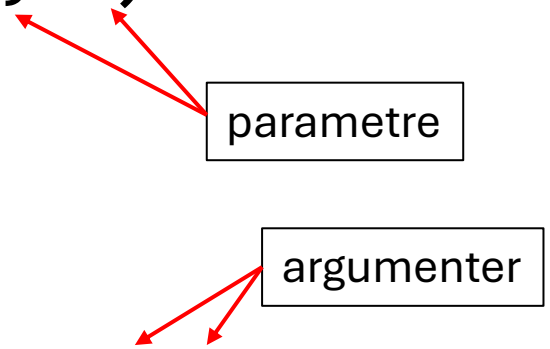
Funksjoner – en viktig byggekloss

- Funksjoner er en av de absolutt viktigste byggeklossene vi har for å kunne lage modulære og lesbare programmer. En funksjon er ikke noe annet enn et navn på en kodesnutt som vi ønsker å bruke gjentatte ganger.
- La oss begynne med det helt enkle, sånn at vi ser prinsippet: vi ønsker oss en funksjon som legger sammen to tall.

Funksjon som legger sammen to tall

```
# Først må vi definere funksjonen:
def add_two_numbers(a, b):
    return a + b


# bruk den
svar1 = add_two_numbers(1,2)
print(svar1) # 3
svar2 = add_two_numbers(3,4)
print(svar2) # 7
```



Vi lager en funksjon ved å lage en funksjonsheader som må inneholde:
def – et nøkkelord som forteller Python at vi definerer en funksjon
Funksjonsnavn – et navn vi velger, som følger navngivingsreglene i Python
Parenteser () – alltid med, selv om funksjonen ikke tar inn noen verdier
Parametre (valgfritt) – plasseres inni parentesene hvis funksjonen skal ta imot inn-data
Kolon : – avslutter headeren og markerer starten på funksjonsblokka, altså koden til funksjonen

Funksjonen *kan* ***returnere*** noe, vi bruker **return** setninga for å gjøre dette, i vårt eksempel summen av parametrene a og b.

En litt mer nyttig funksjon: gitt en score, returner karakter

```
# definer funksjonen
def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"
```

```
# bruk funksjonen

score1 = 85
grade1 = calc_grade(score1)
print(grade1)

score2 = 92
grade2 = calc_grade(score2)
print(grade2)
```

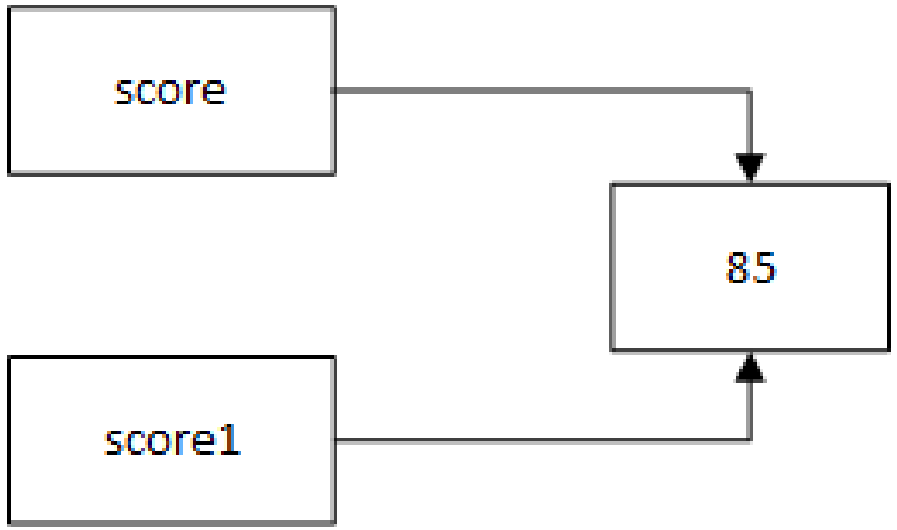
Utskrift:

```
B
A
```

Argument og parameter refererer samme objekt

```
def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"

score1 = 85
grade1 = calc_grade(score1)
```



Bruk av stack og heap ved funksjonskall

- Som programmerer bør du kjenne til hva som skjer ved et funksjonskall.
- I et program er flere ulike typer minneområder brukt. Ved et funksjonskall er minneområdene *stack* og *heap* i bruk.
- ***Stack*** er minneområdet som brukes til å administrere utveksling av data mellom funksjonskallet og funksjonen, mens ***heap*** brukes til å allokere plass til objektene som det refereres til.

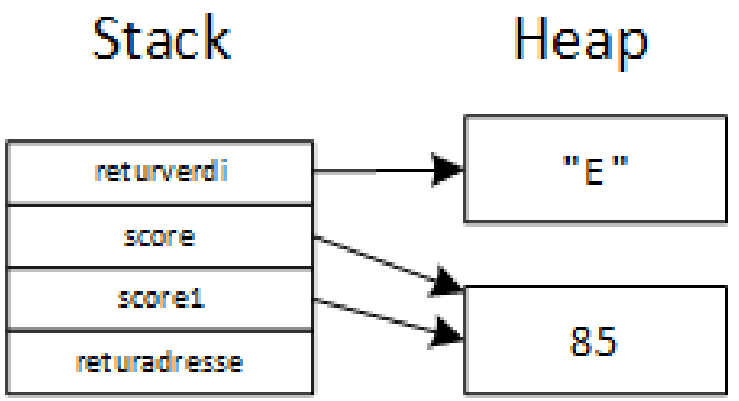
Når en funksjon kalles, legges følgende på stack (forenklet og i typisk rekkefølge):

- **Returadresse**
  - Stedet hvor programmet skal fortsette å eksekvere etter at funksjonen er ferdig, dette er grovt sett kodelinja etter funksjonskallet
- **Argument-referanser**
  - Referanser til objektene som sendes inn som argumenter
- **Parameter-referanser**
  - Disse peker til de samme objektene som argumentene (i Python kopieres referansen, ikke objektet)
- **Lokale variabler**
  - Alt som defineres inne i funksjonen (inkludert parameterne) lagres i stack-frame for funksjonen
- **Returverdi-referanse**
  - Når funksjonen returnerer, opprettes en referanse til returverdien (som ligger på heap)

Det vil se omtrent slik ut…

```
def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"

score1 = 85
grade1 = calc_grade(score1)
```



Overføring av argumenter: immutable

- Vi har tidligere sett at `int` og `str` er eksempler på *immutable* datatyper – det vil si at de ikke kan endres etter at de er opprettet. Når vi forsøker å "endre" slike verdier i en funksjon, skapes det i virkeligheten et nytt objekt, og den opprinnelige verdien forblir uendret.
- For å tydelig vise dette, lager vi to funksjoner: `modify_int(x)` og `modify_str(s)`. Disse forsøker å endre henholdsvis et heltall og en streng som sendes inn som argument.

Overføring av argumenter: immutable

- Vi har tidligere sett at `int` og `str` er eksempler på *immutable* datatyper – det vil si at de ikke kan endres etter at de er opprettet. Når vi forsøker å "endre" slike verdier i en funksjon, skapes det i virkeligheten et nytt objekt, og den opprinnelige verdien forblir uendret.
- For å tydelig vise dette, lager vi to funksjoner: `modify_int(x)` og `modify_str(s)`. Disse forsøker å endre henholdsvis et heltall og en streng som sendes inn som argument.

```
def modify_int(x):
    print("Inside function (before change):", x, "| id:", id(x))
    x = x + 1
    print("Inside function (after change):", x, "| id:", id(x))

def modify_str(s):
    print("Inside function (before change):", s, "| id:", id(s))
    s = s + "!"
    print("Inside function (after change):", s, "| id:", id(s))
```

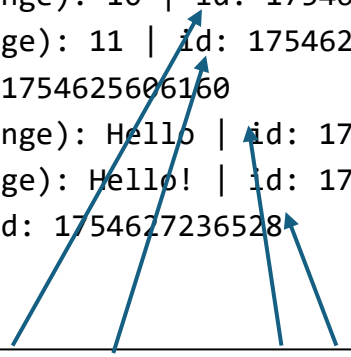
Vi kaller funksjonene slik:

```
x = 10
modify_int(x)
print("Outside function:", x, "| id:", id(x))

s = "Hello"
modify_str(s)
print("Outside function:", s, "| id:", id(s))
```

Utskrift:

```
Inside function (before change): 10 | id: 1754625606160
Inside function (after change): 11 | id: 1754625606192
Outside function: 10 | id: 1754625606160
Inside function (before change): Hello | id: 1754627236528
Inside function (after change): Hello! | id: 1754627240304
Outside function: Hello | id: 1754627236528
```



Observasjon:

Objekt-ID-en ***endres*** inne i funksjonen, noe som viser at det er opprettet et *nytt* objekt. Den opprinnelige verdien utenfor funksjonen er uendret.

Overføring av argumenter: mutable

`list` er en mutabel datatype. Et tilsvarende eksempel som for `int` og `str` viser forskjellen i oppførsel.

```
def modify_list(lst):
    print("Inside function (before change):", lst, "| id:", id(lst))
    lst.append(4)
    print("Inside function (after change):", lst, "| id:", id(lst))
```

Vi kaller `modify_list` og ser hva som skjer med lista:

```
lst = [1, 2, 3]
modify_list(lst)
print("Outside function:", lst, "| id:", id(lst))
```

Utskrift:

Inside function (before change): [1, 2, 3] | id: 1754626995968
Inside function (after change): [1, 2, 3, 4] | id: 1754626995968
Outside function: [1, 2, 3, 4] | id: 1754626995968

Observasjon:

Objekt-ID-en er ***den samme*** før og etter endringen, både inne i og utenfor funksjonen. Det betyr at listen faktisk ble *modifisert direkte*.

Navngiving argument og parametre

- En utbredt nybegynner-misforståelse er at navn på argument og parameter har betydning for parameteroverføring.
- Det er absolutt ikke tilfelle.
- Parameter navn er kun kjent innenfor funksjonen og konflikter ikke med variabelnavn definert i globalt scope.
- Derfor kan parameter navn være lik argument navn, det er ingen konflikt.
- Kodesnutten på neste side demonstrerer dette

```
# file: sc_06_03.py
01: # parameter navn kan være hva som helst
02: # parameter navn har lokalt scope og har ingenting
03: # med variabler utenfor funksjonen å gjøre
04: def add_two_numbers(x, y):
05:     return x + y
06:
07: def add_two_numbers(a, b):
08:     return a + b
09:
10:
11: x = 1
12: y = 2
13: svar1 = add_two_numbers(x, y)
14: print(svar1) # 3
```

Dette er kode hvor en funksjon defineres pånytt: det er den siste definerte funksjonen som vil bli kalt.

Retur av flere verdier

En funksjon kan returnere mer enn én verdi. Dette er nyttig når vi ønsker å hente ut flere resultater fra én beregning. For eksempel hvis vi ønsker å finne både summen og gjennomsnittet av tre tall:

```
def calculate_sum_and_average(a, b, c):
    total = a + b + c
    average = total / 3
    return total, average

sum1, avg1 = calculate_sum_and_average(3, 6, 9)
print("Sum:", sum1)
print("Gjennomsnitt:", avg1)
```

Funksjonen returnerer to verdier, og de blir «pakket ut» i to variabler. Dette gjør koden ryddig og effektiv.

Type hints

Type hints i Python er en måte å spesifisere hvilke datatyper funksjoner *forventer* som argumenter og hva de returnerer. Dette gjør koden mer lesbar og lettere å forstå, og kan hjelpe med feilsøking og automatisk verktøystøtte (f.eks. i IDE-er).
Her er funksjonen `add_two_numbers` med type hints lagt til:

```
def add_two_numbers(a: int, b: int) -> int:
    return a + b
```

Forklaring:
`a: int` og `b: int` betyr at funksjonen forventer to heltall som argumenter.
`-> int` betyr at funksjonen returnerer et heltall.

Du kan også bruke andre typer, som `float`, `str`, `list`, `dict`.

Type hints vil *ikke hindre at du kaller funksjonen med feil type argumenter*, det er kun et «tips» om hva funksjonen *forventer* og hva den returnerer.

Keyword arguments

- *Keyword arguments* gjør det mulig å sende inn verdier til funksjonen ved å bruke *navn* på parameterne. Dette gjør koden mer lesbar, spesielt når funksjonen har mange parametre.

```
def create_greeting(name, greeting):
    return f"{greeting}, {name}!"

msg1 = create_greeting(name="Ola", greeting="Hei")
msg2 = create_greeting(greeting="Hallo", name="Kari")
print(msg1)
print(msg2)
```

- Ved å bruke navn på argumentene på kallstedet, kan vi sende dem *i hvilken som helst rekkefølge.* Dersom vi ikke bruker keyword, så må vi sende argumentene i nøyaktig den rekkefølgen de er forventet å komme.

Default verdier

- Noen ganger ønsker vi at en funksjon skal ha en *standardverdi* dersom vi ikke sender inn noe. Dette gjør funksjonen mer fleksibel.

```
def greet(name, greeting="Hei"):
    print(f"{greeting}, {name}!")

greet("Anna")           # Bruker standardverdi
greet("Jon", "Hallo")   # Overstyrer standardverdi
```

- Standardverdier gjør at vi kan kalle funksjonen med færre argumenter når det passer.
- Dersom du både har standardverdi og ikke for parametrene, så må parametrene uten standardverdi komme før de med standardverdi:

Ok:

```
def funksjon(a, b=2, c=3):
    return a + b + c
```

Ikke ok:

```
def funksjon(a=1, b): # du får SyntaxError
    return a + b
```

OBS! standardverdier evalueres kun en gang; - når funksjonen defineres! (1)

- Standardverdier evalueres kun når funksjonen *defineres*, ikke hver gang funksjonen kalles. Dette kan føre til uventet oppførsel med *mutable* objekter som list og dict:

```
def legg_til(element, liste=[]):
    liste.append(element)
    return liste

print(legg_til(1))  # [1]
print(legg_til(2))  # [1, 2] ❗
```

- Det er den samme lista som brukes ved kall nummer to!

OBS! standardverdier evalueres kun en gang; - når funksjonen defineres! (2)

- Immutable objekter som **int, float, str, tuple** kan *ikke endres* etter at de er opprettet. Hvis du bruker en slik som standardverdi, er det ingen risiko for at verdien "husker" tidligere kall, fordi den ikke kan endres.

Løsning hvis mutable datatyper:

```
def legg_til(element, liste=None):
    if liste is None:
        liste = []
    liste.append(element)
    return liste

print(legg_til(1))  # [1]
print(legg_til(2))  # [2]
```

Synlighet (scope) av variabler

- Når vi programmerer i Python, er det viktig å forstå *scope* – altså hvor i programmet en variabel er synlig og kan brukes
- Python har følgende typer scope
  - **Lokal**: Gyldig bare inne i en funksjon, klasse eller annen struktur som skaper lokalt scope.
  - **Global**: Gyldig i hele modulen eller skriptet, fra det punktet den er definert.
  - **Ikke-lokal (enclosing)**: Gyldig i en omsluttende funksjon (ved nøstede funksjoner).
  - **Innebygd (built-in)**: Gyldig overalt – navn som print, len, osv.

Hva skaper lokalt scope?

Hva	Skaper lokalt scope?	Kommentar
`def` (funksjon)	Ja	Lokalt scope for parametere og lokale variabler
`class`	Ja	Eget scope for attributter og metoder, men disse er likevel tilgjengelig for klientkode
Modul (.py-fil)	Ja	Globalt scope for modulen
`lambda`	Ja	Som funksjon – har eget scope
`list` comprehension	Ja	Variabler inni er ikke tilgjengelige utenfor
`dict`/`set` comprehension	Ja	Samme som list comprehension
Generator expression	Ja	Eget scope
`if`, `for`, `while`, `try`	**Nei**	**Variabler lever videre utenfor blokken**
`with`-blokk	Nei	Ingen nytt scope
`match`-blokk (Python 3.10+)	Nei	Ingen nytt scope

if __name__ == "__main__":

- Dette uttrykket brukes for å kontrollere **om en Python-fil kjøres direkte**, eller **om den importeres som et modul** i en annen fil.

```
# File: sc_06_04_cels_to_fahr.py
def celsius_to_fahrenheit(celsius):
      fahrenheit = (celsius * 9/5) + 32
      return fahrenheit

# kjøres når filen kjøres direkte
if __name__ == "__main__":
      celsius = float(input("Skriv inn temperatur i Celsius: "))
      fahrenheit = celsius_to_fahrenheit(celsius)
      print(f"{celsius} grader Celsius er {fahrenheit} grader Fahrenheit.")
```

```
# File: sc_06_05_use_cels_to_fahr.py
import sc_06_04_cels_to_fahr as converter
# bruker funksjonen fra sc_06_04_cels_to_fahr.py
celsius = float(input("Skriv inn temperatur i Celsius: "))
fahrenheit = converter.celsius_to_fahrenheit(celsius)
print(f"{celsius} grader Celsius er {fahrenheit} grader Fahrenheit.")
```

Variabelen __name__

- Dersom Python fila kjøres direkte får variabelen `__name__` verdien **`"__main__"`**
- Dersom fila importeres, settes `__name__` til eget filnavn, altså `"sc_06_04_cels_to_fahr"`

```
# File: sc_06_04_cels_to_fahr.py
def celsius_to_fahrenheit(celsius):
      fahrenheit = (celsius * 9/5) + 32
      return fahrenheit

# kjøres når filen kjøres direkte
if __name__ == "__main__":
      celsius = float(input("Skriv inn temperatur i Celsius: "))
      fahrenheit = celsius_to_fahrenheit(celsius)
      print(f"{celsius} grader Celsius er {fahrenheit} grader Fahrenheit.")
```

```
# File: sc_06_05_use_cels_to_fahr.py
import sc_06_04_cels_to_fahr as converter
# bruker funksjonen fra sc_06_04_cels_to_fahr.py
celsius = float(input("Skriv inn temperatur i Celsius: "))
fahrenheit = converter.celsius_to_fahrenheit(celsius)
print(f"{celsius} grader Celsius er {fahrenheit} grader Fahrenheit.")
```