

4 if – else konstruksjonen

Læringsmål dette kapittel

- Bruke if «en-veis» (kun bruk av if)
- Forstå kodeblokker og betydningen av dette for kontrollflyten i betingede utsagn
- Bruke if «to-veis» altså med **if + else** (sant / ikke sant)
- Bruke nøstet **if - else** (flere valg)
- Bruke **elif** for å øke lesbarhet ved nøstet **if - else**
- **if - else** med kombinerte betingelser
- Kombinere relasjonelle og logiske operatorer
- Kunne bruke sannhetstabeller for å analysere og konstruere komplekse logiske uttrykk med and, or, og not operatorene
- Bruke sammenkjedede sammenligninger (chained comparisons)
- Bruke kortform betinget uttrykk
- Bruke **match - case** konstruksjonen
- Se på alternative måter å utføre valg (f.eks. ved hjelp av lister og ordbøker)
- Forstå hva som er sant og usant i forbindelse med verdi på ulike variabeltyper (truthiness)
- Forstå hvordan egendefinerte objekter evalueres til sant (truthy) eller usant (falsy) i en boolsk kontekst, inkludert bruken av `_bool_()` og `_len_()` metodene for egendefinerte klasser
- Kjenne til datatypen `None`
- Vite om praktiske implikasjoner av funksjoner/metoder som returnerer `None`, som f.eks. `list.sort()`-metoden

Det er med **if - else** konstruksjonen du bestemmer hva programmet ditt skal gjøre; - hvordan det skal løse problemet.

!!!

if - else og hva som følger av kode er basert på hva du mener er den riktige logikken for å løse det aktuelle problemet. Det er her de fleste programmer (som ikke leverer riktig resultat) feiler; - den som har kodet har tenkt feil. Og som jeg har sagt tidligere: dersom du ikke har forstått problemet 100% så kan du ikke skrive den riktige løsningen.

Så; - jeg gjentar regel nummer **en** før du begynner å strø kode utover editor vinduet: **forstå problemet!**

4.1 Enkle if – else med relasjonelle og logiske operatorer

if i if – else konstruksjonen evaluerer et *boolsk uttrykk* som evaluerer til True eller False

if – else konstruksjonen fungerer slik:

```
if betingelse:
    # Kode som kjøres hvis betingelsen er sann
else:
    # Kode som kjøres hvis betingelsen er usann
```

Selv betingelsen er et uttrykk som kan være en enkel sammenligning med en av de *relasjonelle* operatorene:

større enn	>
større eller lik	=
mindre enn	<
mindre eller lik	<=
likhet	==
ikke likhet	!=

... men kan også være mer komplisert; - flere relasjonelle operatorer brukt i kombinasjon med de *logiske* operatorene:

and, or og not.

Uansett; - etter evaluering av uttrykket står vi igjen med en verdi True eller False, som bestemmer om du skal følge if greina, eller else greina.

Under en oversikt som viser utfall av sammenligninger gitt at vi har tre variabler som tilordnes slik:

```
a = 10
b = 5
c = 10
```

Uttrykk	Resultat
a == b	False
a == c	True
a != b	True

a != c	False
a > b	True
a < b	False
a >= b	True
a <= b	False
a >= c	True
a <= c	True
a > b and b < c	True
a > b or b > c	True
not (a < b)	True

Kodeblokker

I forbindelse med `if - else` konstruksjonen får vi anledning til å introdusere begrepet *kodeblokk*. En **kodeblokk** er en gruppe linjer med kode som hører sammen og utføres som en enhet. I Python markeres en kodeblokk med **innrykk** (vanligvis 4 mellomrom), i motsetning til mange andre språk som bruker {}.

Her et eksempel:

```
temperature = 5
if temperature < 10:
    print("It's cold.")
    print("Wear a jacket.")
```

De to `print`-linjene er en del av `if`-blokken fordi de er innrykket.

Hvis `temperature < 10` er sant, kjøres begge linjene.

Hvis ikke, hoppes hele blokken over.

Det er ingen grense for antall kodelinjer du kan ha pr blokk, men husk på at de står på samme innrykk. Og, uansett om du har en eller flere kodelinjer som du ønsker skal utføres; - de må være rykket inn i linja under `if - (eller else-) setninga` (som må avsluttes med kolon :).

Som sagt, `if - else` konstruksjonen er ikke vanskelig, men det kan noen gang «krølle seg litt til i hodet» når en logikksekvens blir litt komplisert, f eks når en må forholde seg til flere premisser (eksempler nedenfor).

Det beste måten å forklare en språkkonstruksjon på, er med enkle eksempler som er lett å følge. Vi tar de enkle eksemplene først og bygger på med litt mer kompliserte, før vi til slutt ser på noen litt mer utfordrende eksempler.

Eksempel på «en-veis» **if**:

```
temperature = 5
if temperature < 10:
    print("It's cold outside. Wear a jacket.")
```

Forklaring: Hvis temperaturen er under 10 grader, får vi en anbefaling om å ta på jakke. Hvis ikke, skjer ingenting.

Eksempel på "to-veis» **if**:

```
temperature = 15
if temperature < 10:
    print("It's cold outside. Wear a jacket.")
else:
    print("The weather is mild. A sweater is enough.")
```

Forklaring: På grunn av **else** delen får vi alltid en beskjed – enten det er kaldt eller mildt.

4.2 Nøstet if - else

Hva hvis vi vil gi en mer presis anbefaling; - vi vil gi ulike beskjeder avhengig av flere temperaturintervall? Vi får behov for flere if – else for å løse problemet, såkalt *nøstede if – else*.

Hva hvis vi ønsker å gi anbefalinger basert på flere temperaturintervaller?

Løsningen under er korrekt, men litt uoversiktlig

```
temperature = 28
if temperature < 10:
    print("Wear a warm jacket.")
else:
    if temperature < 20:
        print("A sweater should be fine.")
    else:
        if temperature < 30:
            print("T-shirt weather!")
        else:
            print("It's hot! Stay hydrated.")
```

Forklaring: Hver **else** inneholder en ny **if**, som sjekker neste betingelse.

4.3 Bruke `elif` for å øke lesbarhet

Her har Python et nyttig keyword, `elif`, som gjør at koden over kan skrives om til noe som er *mye mer lesbart*:

```
temperature = 28
if temperature < 10:
    print("Wear a warm jacket.")
elif temperature < 20:
    print("A sweater should be fine.")
elif temperature < 30:
    print("T-shirt weather!")
else:
    print("It's hot! Stay hydrated.")
```

Ingen tvil om at `elif` gir mer **lesbar og ryddig kode**?

4.4 `if` – `else` med kombinerte betingelser

Så et eksempel som involverer *to* premisser som vi må forholde oss til, i tillegg til temperatur er det muligens regn i lufta?

```
temperature = 12
is_raining = True

if temperature < 20:
    if is_raining:
        print("Wear a raincoat.")
    else:
        print("A light jacket is enough.")
else:
    if is_raining:
        print("Take an umbrella.")
    else:
        print("Enjoy the nice weather!")
```

Legg merke til at en `else` vil tilhøre den `if` som står på samme innrykknivå, denne regelen er viktig for å kunne lese koden korrekt.

4.5 Kombinere relasjonelle og logiske operatorer

Ved å *kombinere* logiske operatorer med relasjonelle kan vi tilnærme oss løsningen på en annen måte:

```

temperature = 12
is_raining = True

if temperature < 20 and is_raining:
    print("Wear a raincoat.")
elif temperature < 20 and not is_raining:
    print("A light jacket is enough.")
elif temperature >= 20 and is_raining:
    print("Take an umbrella.")
else:
    print("Enjoy the nice weather!")

```

Var dette bedre enn foregående eksempel? Kanskje en smakssak...

Uansett, vi ser at stødig koding med `if - else` er vital for å produsere det riktige resultatet. Det er slike kodepassasjer i programmet ditt *som er selve programmet*, det som løser problemet, riktig nok banalt i dette tilfellet, men den kan ha kritisk betydning i andre situasjoner.

I eksemplene over hadde variablene en gitt verdi før inngang i `if - else` sekvensen. I de fleste tilfeller vil de variablene vi tester på ha verdier *gitt av en input fra et sted*; - data fra datastrømmer (filer, databasespøringer, data fra nettet) eller input fra konsoll. Da er det spesielt viktig at `if - else` konstruksjonen er robust mot feil i data eller grensebetingelser. Vi skal senere se på *Unit testing*, som er en teknikk for å sjekke at din kode fungerer med alle tenkelige data som input (i alle fall er det målet med unit testing).

Når vi snakker om uttrykk som er en kombinasjon av relasjonelle og logiske operatorer..., her er det på sin plass å repetere koden for å finne ut om et år er et skuddår eller ikke.

Et år er et skuddår hvis det er delelig med 4, men ikke delelig med 100, med mindre det også er delelig med 400.

Her er den kompakte løsningen i Python:

```

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print("Det er et skuddår")
else:
    print("Det er IKKE et skuddår")

```

4.5.1 Sannhetstabell for de logiske operatorene

Du vil erfare, de gangene du må mikse bruk av **and** og **or**, at det er lett å gjøre feil. Gjerne slik at en tenker *motsatt* av det det bør være 😊.

Her er en *sannhetstabell* for de logiske operatorene **and**, **or** og **not**.

A	B	A and B	A or B	not A	not B
<hr/>					
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

4.6 Sammenkjedede sammenligninger

Python har mulighet for *sammenkjedede sammenligninger*, som gjør at du kan skrive det relasjonelle utrykket slik at det ligner mer på den matematiske notasjonen.

Matematisk:

$10 \leq x < 20$

Uten sammenkjeding ville vi kodet det slik:

```
if 10 <= x and x < 20:
```

Ved bruk av sammenkjeding kan vi skrive det slik:

```
10 <= x < 20
```

Sammenligning på likhet mellom tre variabler:

```
if a == b and b == c:
    print("Alle tre er like")
```

Skriv heller:

```
if a == b == c:
    print("Alle tre er like")
```

4.7 Kortformet betinget uttrykk

Python har en konstruksjon som kalles ternær operator eller *kortformet betinget uttrykk*. Syntaksen er slik:

```
verdi_hvis_sann if betingelse else verdi_hvis_usann
```

Altså:

Det testes på betingelse. Hvis betingelsen evaluerer til **sann** vil **verdi_hvis_sann** være **resultatet av uttrykket**, hvis ikke vil **verdi_hvis_usann** være **resultatet av uttrykket**.

En nyttig anvendelse av denne konstruksjonen er å lage grammatisk korrekte utskrifter. Se på denne koden:

```
antall_epler = int(input("Hvor mange epler har du? "))
print(f"Du har {antall_epler} eple{'r' if antall_epler > 1 else ''}.")
```

Kjøring av programmet:

Hvor mange epler har du? 3
Du har 3 epler.

Kjøring av programmet:

Hvor mange epler har du? 1
Du har 1 eple.

I den siste utskriften er det noe som skurrer; - vi skulle gjerne sett utskriften
Du har ett eple.

Ikke sant?

Kanskje du kan lage den kodebiten som må inn for å få dette til?

En annen sak vi kan bemerke i dette kodeksempelet er testen på om det er mer enn ett eple, altså

`antall_epler > 1`

Hva hvis brukeren oppgir 0 eller en negativ verdi?

Kjøring av programmet:

Hvor mange epler har du? -1
Du har -1 eple.

Her kunne det kanskje vært på sin plass å introdusere en ekstra `if` test for å gi en feilmelding hvis tallet var null eller negativt.

Her en ny omskriving av koden hvor vi gir råd om bekledning, hvor vi bruker kortform betinget uttrykk:

```
temperature = 12
is_raining = True

if is_raining:
    print("Wear a raincoat." if temperature < 20 else "Take an
umbrella.")
else:
    print("A light jacket is enough." if temperature < 20 else
"Enjoy the nice weather!")
```

4.8 Bruke `match` – `case` konstruksjonen

Python 3.10 introduserte en ny og kraftig kontrollstruktur kalt **mønstermatching**, som bruker nøkkelordet `match`. Denne strukturen lar deg sammenligne en verdi mot flere mulige mønstre, på en måte som ligner på switch-setningen i andre språk – men med langt større fleksibilitet. `match`-setningen kan brukes med mange ulike datatyper, som for eksempel:

- heltall (int)
- tekststrenger (str)
- lister og tupler
- objekter fra klasser
- ordbøker (dict)

Her fokuserer vi på en datatype vi kjenner til, **heltall**, med et eksempel som er greit å forstå og godt egnet til å illustrere grunnprinsippene.

Ukedager med `match`

La oss si at vi har et heltall som representerer en ukedag, der 1 betyr mandag, 2 betyr tirsdag, og så videre. Vi kan bruke `match` til å skrive en ryddig og lettles struktur for å finne navnet på dagen:

```
dag = 3
match dag:
    case 1:
        print("Mandag")
    case 2:
        print("Tirsdag")
    case 3:
        print("Onsdag")
```

```

case 4:
    print("Torsdag")
case 5:
    print("Fredag")
case 6:
    print("Lørdag")
case 7:
    print("Søndag")
case _:
    print("Ugyldig dag")

```

- match dag: starter mønstermatchingen.
- Hver case sjekker om dag har en bestemt verdi.
- _ er et **wildcard** som matcher alt annet – nyttig som en "default"-tilfelle.

4.9 Alternatve måter å gjøre valg på

Ved hjelp av **if** – **else** og **match** konstruksjonene får vi utført valg basert på enkle eller kompliserte relasjonelle / logiske uttrykk. Du vil garantert komme over kode fra *Phytonister* (nær-nerdede programmerere 😊) som utnytter andre mekanismer i Python språket for å oppnå det samme.

Noen ganger kan det bli mer elegant og data-drevet å få utført valgene på, som vi skal se nedenfor.

Vi skal snart lære om **collection** klassene **list**, **tuple** og **dictionary**. I disse datastrukturene kan vi *indeksere* oss inn for å finne verdier, eller bruke en **nøkkel** for å slå opp en verdi (dictionary).

Slik kunne vi løst «finne dag» oppgaven ved hjelp av en liste:

```

dag = 3

ukedager = ["Mandag", "Tirsdag", "Onsdag", "Torsdag",
            "Fredag", "Lørdag", "Søndag"]

if 1 <= dag <= 7:
    print(ukedager[dag - 1])
else:
    print("Ugyldig dag")

```

Forklaring: Vi bruker variabelen **dag** som en *indeks* for å få tak i rett tekststreng i lista som heter **ukedager**. Indekseringen skjer ved at vi oppgir et tall innenfor [] parantesene i setninga

```
print(ukedager[dag - 1])
```

Vi *trekker fra 1* på indeksen, fordi første element i en liste *alltid er 0*.

Slik kunne vi løst «finne dag» oppgaven ved hjelp av en dictionary:

```
dag = 3

ukedager = {
    1: "Mandag",
    2: "Tirsdag",
    3: "Onsdag",
    4: "Torsdag",
    5: "Fredag",
    6: "Lørdag",
    7: "Søndag"
}

print(ukedager.get(dag, "Ugyldig dag"))
```

Forklaring: En *dictionary* (her *ukedager*) fungerer på den måten at du bruker en *nøkkel* for å slå opp en verdi, her ved hjelp av **get** metoden, som returnerer en *default verdi* (standardverdi) hvis nøkkelen ikke finnes.

list, tuple, set og **dict** er sentrale *collection* klasser I Python, og vi kommer utførlig tilbake til disse, selvsagt.

4.10 sant og usant i forbindelse med verdi på ulike variabeltyper: *truthiness*

Allt uttrykk i Python kan evalueres til en boolsk verdi når det trengs — for eksempel i en **if**-setning eller ved bruk av **bool()**-funksjonen. Dette kalles ofte for "truthiness".

bool() er konstruktøren til **bool** klassen. Den returnerer et **bool objekt** som er enten **True** eller **False**.

La oss se hva som skjer når vi gir **bool()** (eller en **if** test) ulik input:

```
bool(0)      # False
bool(1)      # True
bool(None)   # False
bool("")     # False
bool("Hei")  # True
bool([])     # False
bool([1, 2]) # True
```

De to siste linjene berører klassen `list`, som vi ennå ikke har sett på, men viser at en *tom* liste er `False`, mens en liste *med innhold* er `True`.

Forskjell på **uttrykk** (expression) og **statement**

Et **uttrykk** er noe som **kan evalueres til en verdi**. Eksempler:

```
2 + 2
"hei" + "verden"
len([1, 2, 3])
x > 5
```

(Hvis skrevet inn i REPL ville alle disse kvittere med en output)

Dette skiller seg fra et **statement** (setning), som gjør noe, men ikke nødvendigvis returnerer en verdi, som:

```
if, for, while, def, class, return, etc
```

Her er en oversikt over ulike datatyper og hva som vil evaluere til usant / `False` (falsy) og sant / `True` (truthy). Noen av datatypene er «ukjent» for oss nå, men her er en samlet oversikt.

Type	Eksempel på falsy	Eksempel på truthy
<code>int</code>	<code>0</code>	<code>1, -5</code>
<code>float</code>	<code>0.0</code>	<code>3.14</code>
<code>str</code>	<code>""</code>	<code>"hei"</code>
<code>list</code>	<code>[]</code>	<code>[1, 2]</code>
<code>tuple</code>	<code>()</code>	<code>(1,)</code>
<code>dict</code>	<code>{}</code>	<code>{"a": 1}</code>
<code>set</code>	<code>set()</code>	<code>{1, 2}</code>
<code>NoneType</code>	<code>None</code>	<code>(ikke mulig)</code>
<code>bool</code>	<code>False</code>	<code>True</code>
Objekt	<code>__bool__() → False</code>	<code>__bool__() → True</code>

`NoneType` kan aldri bli truthy, se neste delkapittel om verdien `None` og klassen `NoneType`.

Den siste linjen for `Objekt` krever en forklaring: Når du lager dine egne klasser i Python, *vil instanser av disse klassene være "truthy"* — altså evaluere til `True` i et `if`- uttrykk.

Men du kan *styre* hvordan objektet ditt oppfører seg i en boolsk kontekst ved å definere én av disse metodene i klassen:

`__bool__(self)` → skal returnere True eller False
`__len__(self)` → hvis `__bool__(self)` ikke finnes, brukes
`__len__`, og 0 regnes som False

4.11 Verdien None

None i Python er en *spesiell konstant* som representerer *fraværet av en verdi* eller en "null-verdi". Det er *ikke* det samme som 0, `False`, eller en tom streng - *det er en egen type*.

```
>>> type(None)
<class 'NoneType'>
>>>
```

None brukes ofte for å indikere at:

En funksjon ikke returnerer noe eksplisitt.
 En variabel ikke har fått en verdi ennå.
 En verdi mangler eller er ukjent.

Eksempel, en ofte brukte funksjon som returnerer None:

```
liste = [3, 1, 2]
resultat = liste.sort()

print(resultat) # Skriver ut: None
```

Koden over er ofte en overraskelse for nybegynnere. Igjen en liten «sneek preview» av noe vi skal se på senere, en *liste* som vi sorterer. Lista over defineres til å ha tre heltall som vi ønsker å sortere ved hjelp av lista sin innebygde funksjon `sort()`. Mange tror at `sort()` returnerer ei ny, sortert liste, men `sort()` sorterer på stedet, altså den endrer den originale lista. Derfor returneres **None**. De videre programlinjene vil garantert føre til at programmet krasjer, når en begynner å bruke variabelen `resultat`, som er **None** 😢.

© 2025 [Frode Næsje]

Alle rettigheter reservert. Ingen deler av denne publikasjonen kan reproduseres, distribueres, eller overføres i noen form eller på noen måte, elektronisk, mekanisk, fotokopiering, opptak, eller på annen måte, uten forhåndstillatelse fra forfatteren.