REALFAGSLØYPER

Programmering

NOEN SENTRALE BEGREP

NATURFAGSENTERET
NASJONALT SENTER FOR NATURFAG I OPPLÆRINGA

MATEMATIKKSENTERET
Nasjonalt senter for matematikk i opplæringen

Harald Sande
MATEMATIKKSENTERET

REALFAGSLØYPER

Innholdsfortegnelse

Programmering i matematikk

Fra høsten 2020 er programmering en del av skolefaget matematikk. Det medfører at mange matematikklærere må sette seg inn i et nytt fagfelt. Programmering innebærer å designe og planlegge, skrive, teste, feilsøke og optimalisere kildekoden til et dataprogram. Arbeidet er ofte tidkrevende, og det kan sette utholdenheten på prøve. Samtidig kan det være givende, skape mange aha-opplevelser og gi stor mestringsfølelse. Konkrete matematiske begrep kan utforskes og beskrives ved hjelp av programmeringsverktøy, og tankeprosesser i programmeringsarbeidet krever blant annet logikk, algoritmer, abstraksjon og «mønstersniffing». Samtidig krever arbeidet en systematisk tilnærming som ofte også innebærer oppdeling av problemer. Mange av de samme tankeprosessene og arbeidsmetodene kan vi kjenne igjen fra matematikkfaget.

Denne artikkelen trekker frem noen sentrale begrep knyttet til programmeringsfaget – og ikke alle er ukjente for matematikklærere. *Algoritmer*, *variabler* og *funksjoner* er kjente begrep som brukes på samme måte både i programmering og matematikk, mens *løkker* blant annet kan brukes slik at de handler om å oppdage og beskrive mønstre og symmetrier i matematikk. Når programmering blir en del av matematikkfaget, kan det blant annet bidra til å styrke norske elevers kompetanse i anvendelse, forståelse og utvikling av algoritmer.

Scratch og Python er programmeringsspråk som kan være aktuelle å ta i bruk i skolen. De er høy-nivå språk, som vil si at de er nærmere menneskelignende språk enn maskinspråk (som på laveste nivå består av 0 og 1) og dermed også enklere å lære enn språk på lavere nivå. I Scratch bygges programmer ved å ta i bruk blokker av ferdig eller delvis ferdig kode. Man kan også lage egne blokker med kode, som vi skal se i avsnittet om funksjoner. I Python lages programmer ved å skrive kodelinjer på et språk som er laget slik at mennesker skal kunne forstå det. Det gjør det mye enklere å forstå og modifisere andres kode, sammenlignet med å arbeide med et språk på lavere nivå. Eksemplene i artikkelen er laget i Scratch og Python, men begrepene som er i fokus er generelle for programmering uavhengig av språk.

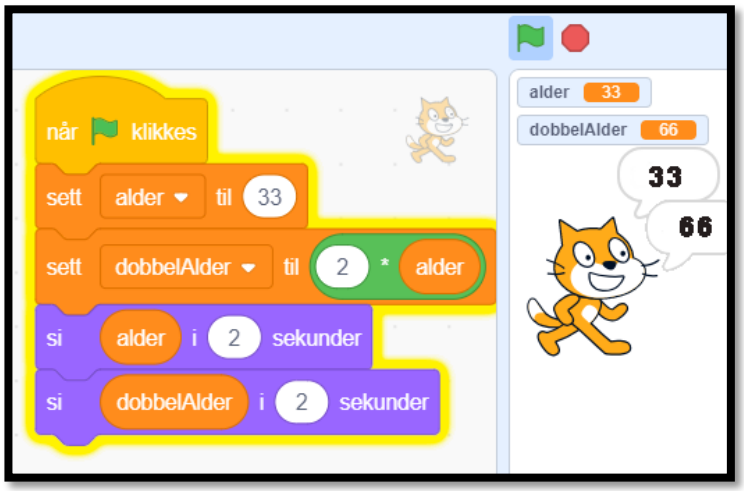
Algoritmer

Algoritmer er ikke ukjent for matematikklærere, og selv om begrepet kanskje ikke nevnes så ofte i klasserommet brukes de daglig i matematikkfaget. De er regelrette oppskrifter på hvordan man effektivt og presist kan utføre beregninger med de fire regneartene, hvordan vi kan speile eller rotere en figur om et punkt, hvordan vi kan anvende Pytagoras setning for å finne ukjente sider i rettvinklete trekanter etc.

Et dataprogram består av én eller flere algoritmer som er laget for å utføre handlinger i bestemte rekkefølger. Dersom en algoritme virker som den skal, vil den behandle all data som kommer inn på samme måte, hver gang den kjøres. Et enkelt eksempel kan være en algoritme som legger sammen to tall. Brukeren skriver inn *tall A* og *tall B,* som kalles *input.* Algoritmen gjør det samme med tallene hver gang: (*tall A*) + (*tall B*), men det som kommer ut som resultat (*output*) er avhengig av input. Dataalgoritmer utføres nøyaktig i den rekkefølgen de er skrevet, linje for linje, og krever derfor presisjon av programmereren. Dette fører til at en viktig del av en programmerers arbeid innebærer feilsøking og retting («debugging») og optimalisering av algoritmer og syntaks (grammatikk).

Variabler

Variabler er heller ikke et ukjent begrep for matematikklærere. De fungerer på samme måte når man programmerer som når de brukes i matematikk. Variabler er grunnleggende komponenter i dataprogrammer og vi gjør to ting med dem:
1) gir dem en verdi,
2) sjekker hvilken verdi de har.
I *Figur 1* og *Figur 2* ser du eksempler på bruk av variabler i programmeringsspråkene Scratch (blokkbasert) og Python (tekstbasert). I eksemplene *tildeles* variablene verdier.



FIGUR 1: VARIABLER I SCRATCH

Vi har to variabler: *alder* og *dobbelAlder*. I første linje tildeles *alder* verdien 33, i andre linje tildeles *dobbelAlder* verdien som er dobbelt så stor som verdien til *alder*. Det vil si at den ene variabelen er uavhengig, mens den andre er avhengig av den første. Eksemplene er like i struktur og behandler variablene på samme måte, men visuelt uttrykker de variablene litt forskjellig til brukeren. I Scratch-eksemplet vil en ferdigprogrammert snakkeboble få det til å se ut som at katten sier alderen i to sekunder, før den sier det dobbelte av alderen i to sekunder etterpå, og programmet er ferdig. I Python-eksemplet vil *alder* skrives på første linje på skjermen, mens det dobbelte skrives på linje to, før programmet stopper. Kommandoen «print» i Python sørger for at det som plasseres i

```
alder = 33

dobbelAlder = 2 * alder

print(alder)
print(dobbelAlder)
```

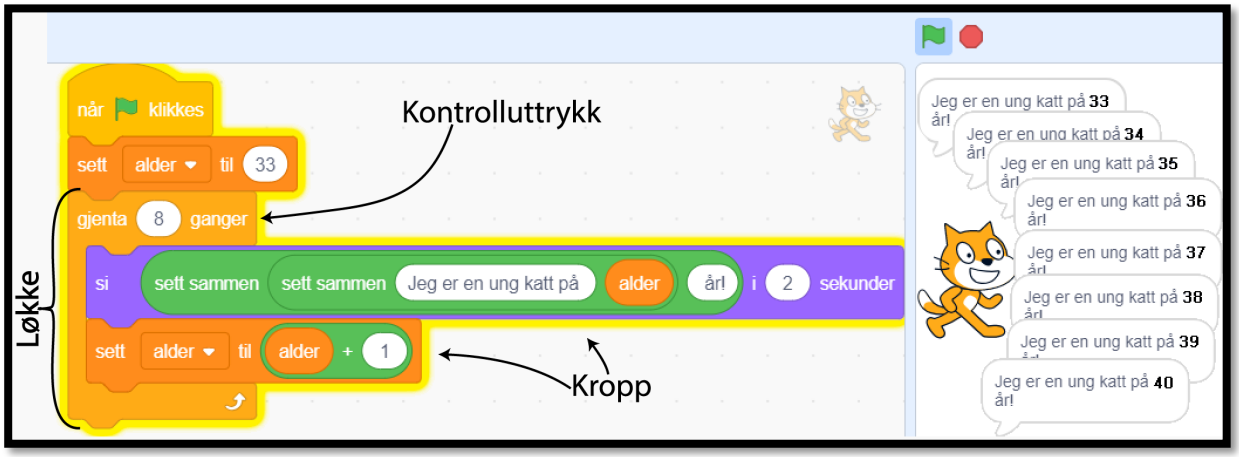
FIGUR 2: VARIABLER I PYTHON

parentesen bak blir skrevet på skjermen når programmet kjøres. Husk at kodelinjene i algoritmene utføres én om gangen, i den rekkefølgen de er skrevet. Det vil si at *alder* alltid blir skrevet ut før *dobbelAlder.*

I mange programmeringsspråk må man definere hvilken datatype en variabel skal være plassholder for. Datatyper kan for eksempel være heltall, desimaltall eller tekst, og har noe å si for hvor stor plass i minnet som reserveres for den enkelte variabel. Dette er ikke nødvendig når vi definerer variabler i Scratch og Python. I Python kan vi for eksempel skrive *a = 23* og *b = 0.5,* og *a* vil automatisk holde av plass til heltall, mens *b* holder av plass til desimaltall. Om en variabel skal endre type i løpet av programmet, må det likevel defineres hva den skal omgjøres til.

Løkker

Vanligvis kjøres en sekvens med kode en gang. Ved å ta i bruk løkker kan vi få hele eller deler av programmet til å gjentas flere ganger. Enten et gitt antall ganger, uendelig (helt til programmet avsluttes), eller til gitte betingelser blir oppfylt. Løkker er veldig nyttig både for å spare tid for forfatteren og for mengden arbeidsminne som programmet legger beslag på. Det er derfor et poeng at programmet tar så liten plass som mulig - desto færre linjer med kode jo bedre. I *Figur 3* ser du et eksempel fra Scratch, hvor en løkke bestemmer at en kodesekvens skal gjentas 8 ganger. Legg merke til hvordan variabelen *alder* øker med 1 for hver gang løkken kjøres.



FIGUR 3: LØKKER I SCRATCH

I eksemplet fra Python (se *Figur 4*) må vi definere en teller som holder styr på hvor mange ganger løkken kjøres. I dette tilfellet starter telleren med verdien 0, og løkken skal kjøres så lenge telleren har verdi mindre enn 8. Uttrykket *while(teller<8),* kalles et kontrolluttrykk og

holder styr på hvor mange ganger løkken skal kjøres. Denne telleren kan også erstattes med variabelen *alder: while(alder<=40),* men da kan vi ikke kontrollere hvor mange ganger løkken kjøres uavhengig av variabelen *alder*. I eksemplet fra Scratch (Figur 3) er en slik teller ferdigprogrammert og innebygd i blokken som tas i bruk.

```
alder = 33
teller = 0

while(teller<8):
    print("Jeg er en ung katt på", alder, "år")
    alder = alder + 1
    teller = teller + 1
```

```
Jeg er en ung katt på 33 år
Jeg er en ung katt på 34 år
Jeg er en ung katt på 35 år
Jeg er en ung katt på 36 år
Jeg er en ung katt på 37 år
Jeg er en ung katt på 38 år
Jeg er en ung katt på 39 år
Jeg er en ung katt på 40 år
```

FIGUR 4: LØKKER I PYTHON

De tre linjene under kontrolluttrykket kalles kroppen til løkken og beskriver hva som faktisk skal skje hver gang løkken kjøres. Til høyre i *Figur 4* er resultatet av at de tre linjene med kode kjøres 8 ganger.

Når løkken har kjørt 8 ganger, fortsetter programmet med linjene som kommer etter løkken. Om det ikke kommer flere linjer, avsluttes programmet. Legg merke til at variablene *alder* og *teller* øker med 1 for hver gang løkken kjøres. I eksemplet fra Python (*Figur 4)* ser vi at svært få linjer med kode trengs for å få mye data ut. For å få 1000 linjer med samme tekst trenger

```
alder = 33

print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
alder = alder + 1
print("Jeg er en ung katt på", alder, "år")
```

```
Jeg er en ung katt på 33 år
Jeg er en ung katt på 34 år
Jeg er en ung katt på 35 år
Jeg er en ung katt på 36 år
Jeg er en ung katt på 37 år
Jeg er en ung katt på 38 år
Jeg er en ung katt på 39 år
Jeg er en ung katt på 40 år
>>>
```

FIGUR 5: LITE EFFEKTIVT PROGRAM

vi kun å endre kontrolluttrykket til *while(teller<1000),* så kjøres løkken 1000 ganger. Det finnes forskjellige typer løkker som kan brukes avhengig av hvordan vi vil sette opp

kontrollutrykket. Det vil avhenge av om vi vil at løkken skal gjentas et gitt antall ganger, uendelig, eller til gitte betingelser oppfylles.

I *Figur 5* ser vi at vi kan få samme resultat ved å skrive ut 8 enkeltkommandoer, men at dette er svært lite effektivt sammenlignet med å ta i bruk løkker.

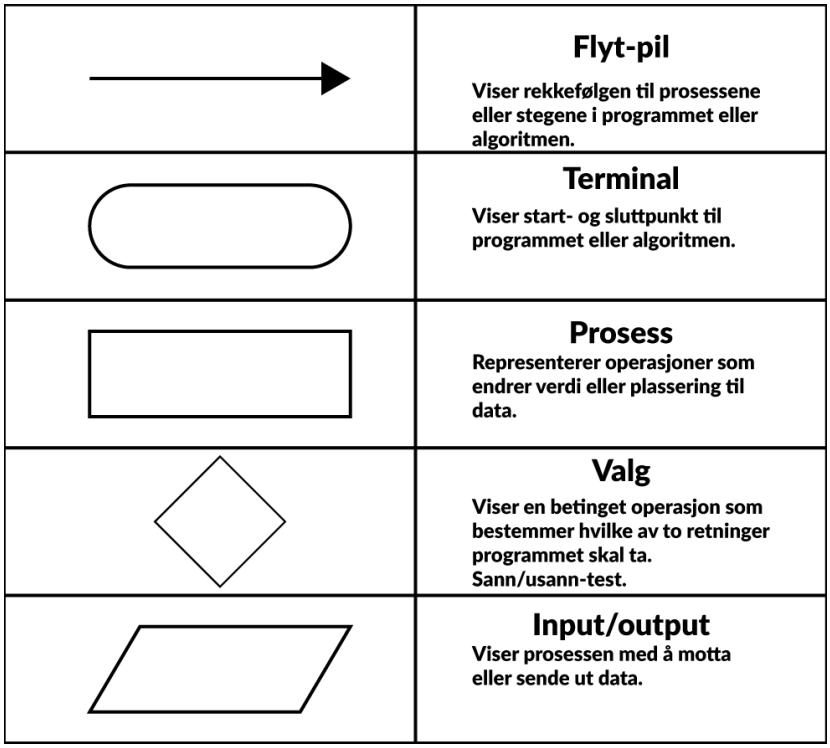
Pseudokode og flytskjema

Pseudokode er en beskrivelse av hendelser og strukturer i et dataprogram. For mennesker er det ofte enklere å lese pseudokode enn tradisjonell programkode. Pseudokoden er bygd etter strukturen til et vanlig programmeringsspråk, men skrives for å leses av mennesker, ikke maskiner. Det er derfor ingen standard konvensjon for hvordan pseudokode skal skrives, og man står fritt til å utforme den slik man selv ønsker. Pseudokode er en del av planleggingen i programmeringsarbeidet, og bidrar med oversikt til forfatteren, samtidig som det er lesbart på tvers av hvilket programmeringsspråk som skal anvendes. La oss si at løkken fra forrige eksempel skal kjøres helt til *alder = 45,* og når katten har rundet 40 år, skal det ikke lenger skrives «Jeg er en ung katt på *alder* år», men heller «Jeg er en middelaldrende katt på *alder* år». Nedenfor er et eksempel på hvordan pseudokoden til denne algoritmen kan se ut:

- Program KatteAlder start:
    - Variabel *alder = 33*
- Mens (*alder* < 40): skriv «Jeg er en ung katt på *alder* år.»
    - Adder 1 til *alder*
- Mens (40 <= *alder* <= 45): skriv: «Jeg er en middelaldrende katt på *alder* år.»
    - Adder 1 til *alder*
- Program KatteAlder slutt.

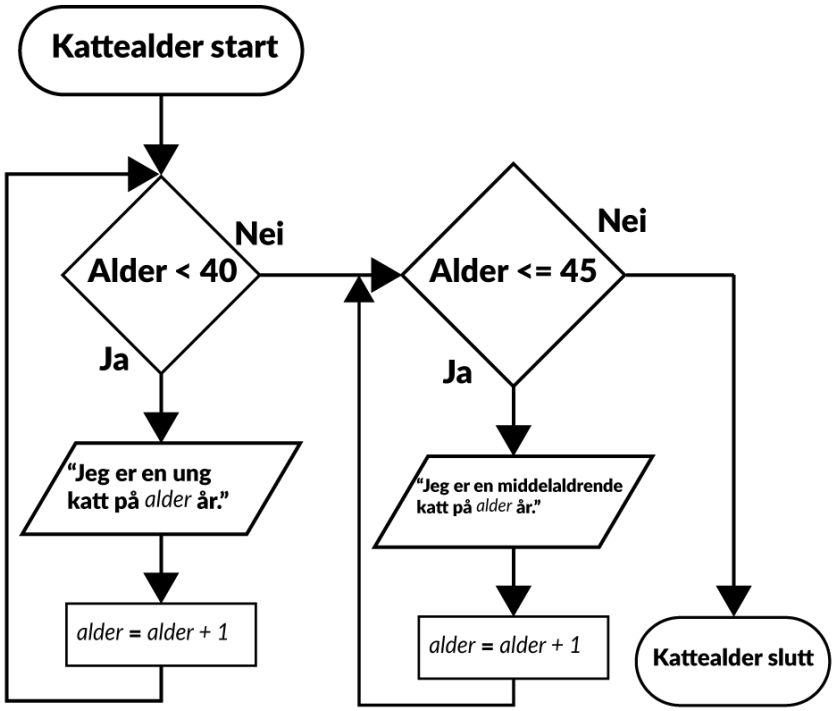
Flytskjema er en type diagram som beskriver hendelser og strukturen i algoritmer. Det fremstiller stegene i algoritmene som bokser med forskjellige former, avhengig av hva de representerer. Mellom hvert steg (boks) er det piler som viser retningen til arbeidsflyten.

Flytskjema og pseudokode beskriver begge hendelser og strukturer i dataprogrammer. Flytskjema tilbyr en mer grafisk tilnærming og passer godt til delproblemer eller mindre prosjekter, da de er oversiktlig, men de krever en del plass. Når flytskjema anvendes, bør standardfigurer tas i bruk. I *Figur 6* ser du en tabell med noen vanlige symboler som brukes i flytskjema.



FIGUR 6: SYMBOLER I FLYTSKJEMA

Programmet Kattealder kan for eksempel se slik ut:



FIGUR 7:

FLYTSKJEMA KATTEALDER

Betingelser

Vi har allerede sett på betingelser i flere eksempler ovenfor. I eksemplet som visualiseres med flytskjemaet i *Figur 7* blir handlinger utført på bakgrunn av to betingelser. Den første betingelsen er «*alder* mindre enn 40», den andre er «*alder* mindre enn eller lik 45». Med utgangspunkt i om disse betingelsene er sanne eller ikke, blir gitte operasjoner utført. Betingelser er viktig i programmering, og styrer mye av flyten i et program. Som i eksemplet ovenfor ser vi at betingelsene bare kan ha én av to verdier, de kan være *sann* eller *usann*, *true* or *false*, *ja* eller *nei*, *1* eller *0*. En vanlig test i et program består av at én betingelse testes. Dersom den er sann utføres *x,* om den er usann utføres *y.* Det kan for eksempel se slik ut:

- Hvis (*alder* < 40): skriv «Du er jo bare ei ungkatt!»
    - Ellers: skriv «Livserfaring er fint.»

Dette er en kombinasjon av *hvis-* og *ellers-utsagnet* og fører til at koden som kommer etter *ellers* blir utført dersom betingelsen (*alder < 40*) ikke er sann. Dette fungerer fint i mange tilfeller, men hva om vi i tillegg vil at det skal skrives noe annet dersom *alder* er under 20? Det vil si at vi har tre betingelser som skal føre til forskjellig output. Se på følgende eksempel:

- Hvis (*alder* <= *20*): Skriv «Du er jo bare ei ungkatt!»
    - Ellers Hvis (20 < *alder* < 40): skriv «Du er vel egentlig midt imellom du.»
    - Ellers: skriv «Livserfaring er fint.»

Dette kalles et *ellers hvis-utsagn* og lar oss teste flere betingelser dersom den foregående ikke er sann. Om ingen er sann vil den siste linjen hvor det står *Ellers* utføres, som vist i eksemplet i *Figur 8.* Uttrykket *elif* er en sammentrekking av *else* og *if.* Vi kan bruke så mange *elif (Ellers Hvis)* som vi måtte ønske.

```
bruker_input = input("Hvor gammel er du? ")

alder = int(bruker_input)

if (alder <= 20):
    print("Du er jo bare ei ungkatt.")
elif (20 < alder < 40):
    print("Du er vel egentlig midt imellom du.")
else:
    print("Livserfaring er fint.")
```

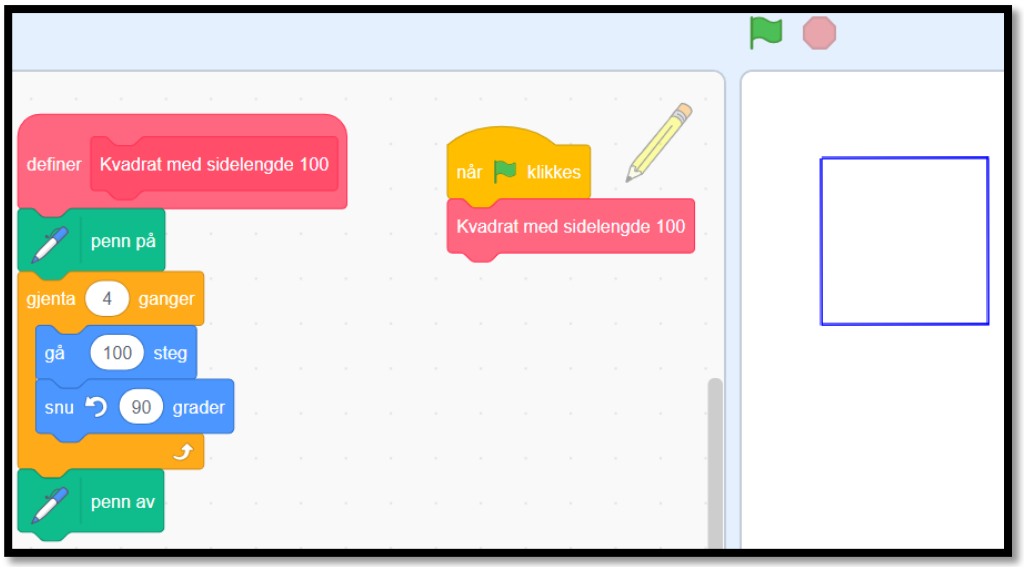
```
Hvor gammel er du? 18
Du er jo bare ei ungkatt.


Hvor gammel er du? 33
Du er vel egentlig midt imellom du.


Hvor gammel er du? 45
Livserfaring er fint.
```

FIGUR 8: ELSE IF I PYTHON

Uttrykk som evaluerer om betingelser er sanne eller usanne kalles *boolske uttrykk*, etter den engelske matematikeren George Boole. Uttrykkene returnerer boolske variabler, som kun kan inneholde verdien sann eller usann.

Funksjoner, parametere og subrutiner

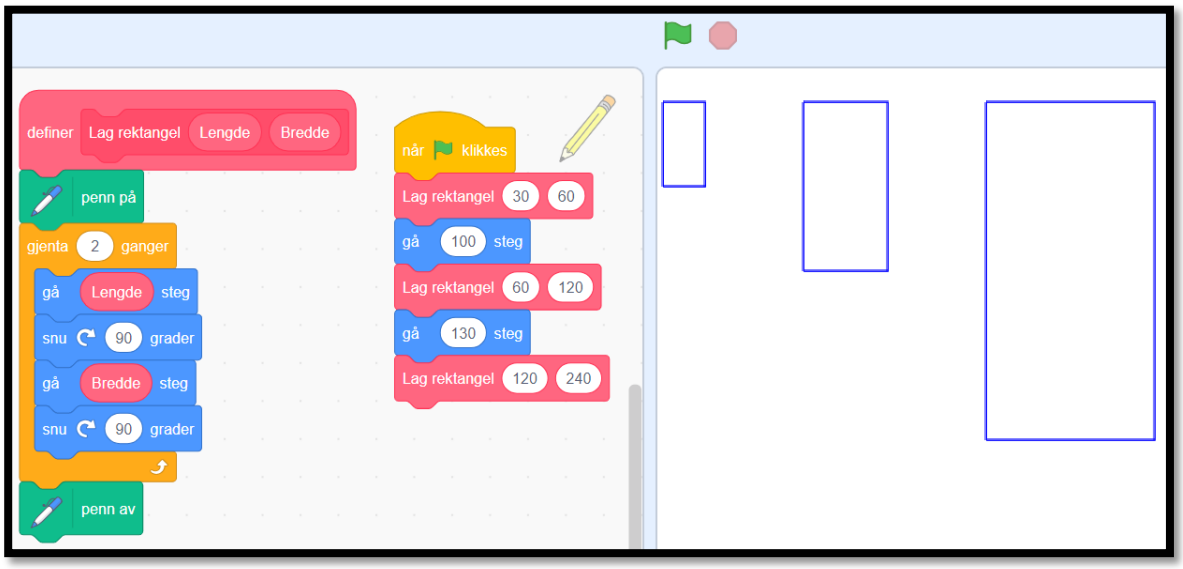
På samme måte som variabler kan lagre verdier som kan brukes igjen og igjen i programkoden, kan funksjoner lagre sekvenser med kode som kan brukes igjen og igjen. Det handler om gjenbruk av kode for å spare tid og for å holde programmet så ryddig og oversiktlig som mulig. Følgende eksempel viser en funksjon som tegner et kvadrat med sidelengde 100:



FIGUR 9: FUNKSJON SOM TEGNER KVADRAT MED SIDELENGDE 100

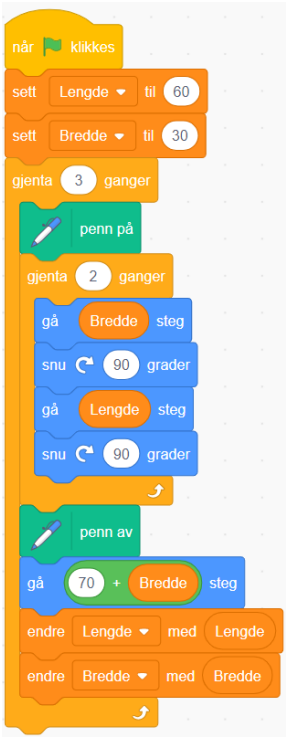
Til venstre i *Figur 9* ser vi at funksjonen defineres med navnet «Kvadrat med sidelengde 100». Under navnet kommer kodesekvensen som skal utføres hver gang funksjonen kalles på (det vil si hver gang vi ønsker at den skal tas i bruk). Til høyre under det grønne flagget ser vi hvordan funksjonen kalles på når det grønne flagget trykkes på. Kanskje ikke veldig imponerende med tanke på at vi kun tegner kvadratet én gang. Funksjoner er spesielt nyttige når vi vil gjøre det samme eller *nesten* det samme flere ganger. Gå ut fra at vi ønsker å tegne tre rektangler med forskjellig lengde og bredde, altså *nesten* det samme tre ganger. Om vi må lage tre funksjoner for å få dette til vil det være like effektivt å legge koden for de tre rektanglene direkte inn i hoveddelen av programmet. Heldigvis kan vi lage én funksjon som lar oss tegne rektangler med forskjellige lengder og bredder, og det er her *parametere* kommer inn i bildet. Når vi har definert hva denne funksjonen skal gjøre, kan vi sende parametere (verdier) til den. Funksjonen utfører det vi ønsker, basert på hvilke parametere vi

sender til den når den kalles på. Sagt med et eksempel kan vi lage en funksjon som heter «Lag rektangel», hvor vi bestemmer at man må sende parametere til funksjonen hver gang den kalles på. Disse parameterne kan vi kalle «Lengde» og «Bredde» og de blir lokale variabler i funksjonen – det vil si at de kun kan brukes innenfor funksjonen. I funksjonens definisjon bruker vi variablene *Lengde* og *Bredde* i stedet for å angi hvor langt og bredt vi vil at rektanglet skal være. Vi vil jo ha muligheten til å lage forskjellige rektangler med den samme funksjonen. Når vi så *kaller* på funksjonen, skriver vi inn parametere som sendes inn i funksjonens variabler, og vi får tegnet rektangler med forskjellige lengde og bredde.



FIGUR 10: TRE REKTANGLER MED FUNKSJON

For ordens skyld må det nevnes at eksemplet fra *Figur 10* rimelig enkelt kan løses ved bruk av løkker, hvor variablene *Lengde* og *Bredde* dobles for hver gang (se *Figur 11*). I programmer hvor en slik sekvens kun skal brukes én gang, spiller det kanskje ingen rolle om man lager en funksjon eller en løkkestruktur, men så fort samme sekvens skal gjentas andre steder i programmet, vil en funksjon være å foretrekke.



FIGUR 11: TRE REKTANGLER UTEN FUNKSJON

Funksjoner kan også brukes i andre funksjoner. Denne kaller vi da en subrutine.

FIGUR 12: FUNKSJON MED SUBRUTINE

I Python må vi definere funksjoner på samme måte, vi velger et navn og bestemmer hva som skal skje når funksjonen kalles på:

```
def min_funksjon():
    print("En hilsen fra funksjonen din")

min_funksjon()
```

FIGUR 13: FUNKSJON I PYTHON

```
En hilsen fra funksjonen din
```

FIGUR 14: OUTPUT FRA FUNKSJONEN

I *Figur 13* ser vi hvordan funksjonen *min_funksjon* blir definert. Når den kalles på vil alle linjene med kode kjøres. I dette tilfelles er det kun én linje som definerer funksjonen, og når den kalles på skrives teksten som vises i *Figur 14.* Funksjonen kan brukes igjen og igjen i programmet, og hver gang den kalles på vil det samme skrives ut, uten at vi trenger å skrive teksten hver gang. På samme måte som i eksemplet fra Scratch, kan vi sende parametere til funksjoner i Python, som blir behandlet ut fra definisjonen til funksjonen. La oss si at vi vil

lage en funksjon som regner ut arealet til et hvilket som helst trapes. Den kan se slik ut:

```
def areal_trapes(a, b, h):
    areal = (a + b) * h/2
    print("For trapes med a = {0}, b = {1} og h = {2}, vil arealet være lik {3}.".format(a,b,h,areal))

areal_trapes(6, 4, 3)
areal_trapes(2, 3, 4)
areal_trapes(13, 16, 5)
areal_trapes(1, 2, 3)
areal_trapes(2, 3, 4)
areal_trapes(17, 5, 6)
areal_trapes(2.3, 5.1, 7)
```

FIGUR 15: FUNKSJON MED PARAMETERE I PYTHON

De første tre linjene i programmet definerer funksjonen *areal_trapes.* Tallene 0-3 som står i krøllparentes refererer til variablene *a, b, h* og *areal* som skal settes inn i teksten som skal skrives ut. Til forskjell fra eksemplet i *Figur 13* er ikke parentesen bak navnet til funksjonen tomt. Det vil si at funksjonen forventer å få tilsendt tre parametere hver gang den kalles på. Disse parameterne kan vi kalle *a, b* og *h,* siden vi skal bruke de til å regne ut arealet til trapeser. Hver gang funksjonen kalles på må vi altså sende med tre parametere som brukes til å regne ut arealet av trapeset. I eksemplet kalles funksjonen på 7 ganger med forskjellige parametere hver gang. Resultatet blir da at funksjonen beregner arealet til 7 forskjellige trapeser og skriver ut den forhåndsbestemte teksten:

```
For trapes med a = 6, b = 4 og h = 3, vil arealet være lik 15.0.
For trapes med a = 2, b = 3 og h = 4, vil arealet være lik 10.0.
For trapes med a = 13, b = 16 og h = 5, vil arealet være lik 72.5.
For trapes med a = 1, b = 2 og h = 3, vil arealet være lik 4.5.
For trapes med a = 2, b = 3 og h = 4, vil arealet være lik 10.0.
For trapes med a = 17, b = 5 og h = 6, vil arealet være lik 66.0.
For trapes med a = 2.3, b = 5.1 og h = 7, vil arealet være lik 25.9.
```

FIGUR 16: OUTPUT FRA FUNKSJON SOM KALLES PÅ 7 GANGER

Oppsummering

Som nevnt innledningsvis krever programmeringsarbeid både tid og utholdenhet. Det krever at den som skal lære har evne til holde ut i problemløsningssituasjoner, til å mislykkes og til å oppsøke kilder med mer kompetanse. Det kan innebære å diskutere med en lærer, medelev, foreldre eller å ta i bruk internett. Internett er ressurs nummer én for mange som lærer programmering, og det flyter over av nyttige artikler og hjelpsomme mennesker som besvarer spørsmål i forum. Programmering er en kompetanse som flere og flere må inneha i

en stadig mer automatisert verden, og for mange kan det virke som en uoverkommelig stor og sammensatt jungel å sette seg inn i. Vi anbefaler å lære ett språk godt i begynnelsen før nye utforskes. Strukturen mellom forskjellige språk er såpass lik, og det går fortere å lære nye språk når en har et godt grunnlag.

Denne artikkelen belyser noen viktige begrep knyttet til programmering, men det er gjennom anvendelse, utforskning, prøving og feiling at man får muligheten til å virkelig forstå og til å være kreativ.