Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Tom Sunde og Charlotte Hoem

Programmering i STEM-fag: Anvendelse av programmering i naturfag og fysikk.

Innføring i programmering og algoritmisk tenkning for lærere i ungdomsskolen og videregående skole.

Bacheloroppgave i Informasjonsbehandling
Veileder: Helge Hafting
Mai 2022

NTNU
Kunnskap for en bedre verden

Tom Sunde og Charlotte Hoem

Programmering i STEM-fag: Anvendelse av programmering i naturfag og fysikk.

Innføring i programmering og algoritmisk tenkning for lærere i ungdomsskolen og videregående skole.

Bacheloroppgave i Informasjonsbehandling
Veileder: Helge Hafting
Mai 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

Sammendrag

Med de endrede kravene etter fornyelse av læreplaner for grunnskole og videregående skole i 2020, er uttrykk som pseudo-kode, metoder, algoritmisk tenkning og programmering blitt høyaktuelle begrep. Dette er nok nye begreper for de fleste av elevene som skal få gleden av å delta i undervisning hvor dette skal inkorporeres i allerede eksisterende emner, men også for lærerne, der nok en stor andel kanskje møter disse nye kravene med en noe ambivalent holdning.

I Norge har en valgt en stegvis tilnærming til det å introdusere programmering og algoritmisk tenkning, som allerede starter på barneskolen gjennom å snakke om modeller og det å følge konkrete oppskrifter, til at en allerede i 5-klasse på barnetrinnet snakker om programmering som et problemløsings-begrep i matematikkfaget. I stedet for å legge det en i rammeplanen kaller "digitale ferdigheter" for seg selv i et eget fag i skolen, har en i Norge valgt å inkorporere programmering og algoritmisk tankegang, i flere ulike fag. Således dukker programmering og algoritmisk tenkning også opp i tradisjonelle fag som matematikk og fysikk, men også som deler av Naturfag på ungdomstrinnet.

Målet med denne oppgaven er å levere et godt hjelpemiddel for lærere i ungdomsskole-trinnet og i videregående skoler. Sluttproduktet tilbyr innføringshefter hvor lærerne kan sette seg inn i både de mest elementære konseptene, og en rekke videregående konsepter innen programmering i Scratch, og i Python. I tillegg tilbyr sluttproduktet ulike oppgavehefter rettet mot hvert enkelt av de ulike års-trinnene på ungdomsskolen, og videregående skole. De enkelte oppgavene starter med en tankeprosess fra å definere en problemløsnings-strategi, via algoritmisk tenking og steg-for-steg beskrivelse av fremgangsmåte for å omforme et praktisk problem i form av en oppgave-tekst, til kode, til siste steg i prosessen som er skriving av fungerende programkode som svarer på de definerte spørsmålene som skal løses. Vi har utformet følgende hefter:

For 8. til 10. klasse: "Innføringshefte i Scratch og Python for ungdomsskolen", inkludert utfyllende eksempler.

For 8. til 10. klasse, naturfag: "Oppgavehefte i Scratch og Python for ungdomsskolen" med åtte oppgaver.

For 1. til 3. klasse VGS: "Innføringshefte i Python for videregående skole", inkludert utfyllende eksempler. Temaene i heftet for ungdomsskolen forutsettes kjent.

For 1. klasse VGS, naturfag: "Oppgavehefte i Python for VG1" med seks oppgaver.

For 2. klasse VGS, fysikk: "Oppgavehefte i Python for VG2" med ni oppgaver.

For 3. klasse VGS, fysikk: "Oppgavehefte i Python for VG3" med seks oppgaver.

Oppgavene er av ulik størrelse, vanskelighetsgrad, og kompleksitet. De er også utformet på en slik måte at de er selvstendige oppgaver, og lærerne kan selv bedømme hvilke oppgaver som er aktuelle å bruke i sin undervisning.

Abstract

With the changed requirements after the renewal of curricula for primary and secondary school in 2020, expressions such as pseudo-code, methods, computational thinking, and programming have become highly relevant concepts. These are most likely new concepts for most of the students who will have the pleasure of participating in teaching where this will be incorporated into already existing subjects, but also for the teachers, where another large share of the teachers may meet these new requirements with a somewhat ambivalent attitude.

In Norway, a step-by-step approach has been chosen to introducing programming and computational thinking, which already starts in primary school by talking about models and following specific recipes, to already in 5th grade in primary school talking about programming as a problem-solving concept in mathematics. Instead of putting what one in the curricula plan calls "digital skills" for itself in a separate subject in school, Norway has chosen to incorporate programming and computational thinking in several different subjects. Thus, programming, and algorithmic thinking also appear in traditional subjects such as mathematics and physics, but also as parts of science at the lower secondary level.

The aim of this thesis is to provide a good aid for teachers in the upper primary school level and in secondary schools. The end product offers introductory booklets where teachers can familiarize themselves with both the most elementary concepts, and a number of advanced concepts in programming in Scratch, and in Python. In addition, the end product offers various exercise booklets aimed at each of the different year-stages in upper primary school and secondary school. The individual tasks start with a thought process from defining a problem-solving strategy, via computational thinking and step-by-step description of procedures for transforming a practical problem in the form of a task text, into code, to the last step in the process which is writing working program code that answers the defined questions to be solved. We have designed the following booklets:

For 8th to 10th grade: An introduction booklet in Scratch and Python for upper primary school, including supplementary examples.

For 8th to 10th grade, science: An exercise booklet in Scratch and Python for upper primary school, with eight assignments.

For 1st to 3rd grade secondary school: An introduction booklet in Python for secondary school, including supplementary examples. The topics in the booklet for upper primary school are assumed to be familiar.

For 1st grade secondary school, science: An exercise booklet in Python for 1st grade secondary school, with six assignments.

For 2nd grade secondary school, physics: An exercise booklet in Python for 2nd grade secondary school, with nine assignments.

For 3rd grade secondary school, physics: An exercise booklet in Python for 3rd grade secondary school, with six assignments.

The tasks are of different sizes, degree of difficulty, and complexity. They are also designed in such a way that they are independent tasks, and the teachers can judge for themselves which tasks are relevant to use in their teaching.

Innhold

Figurer

Tabeller

1 Introduksjon

1.1 Skolen og samfunnet

Skolen er en viktig del av samfunnet. Skolen er en vesentlig del av livet, som er med å former oss til hvordan vi blir som voksne mennesker. Både det sosiale og det faglige er en viktig del av pakka. I denne bacheloren skal vi se på den faglige biten, mer spesifikt innføringen av programmering i naturfag og fysikk på ungdomstrinnet og videregående nivå. I den forbindelse må det defineres en rekke uttrykk – programmering, algoritmisk tenkning, koding – og ikke minst må det ses på hva som er definert i læreplanverket.

Digitalisering hjemme og i skolen er noe som ikke en gang var i folks tanker for 30-40 år siden, i alle fall ikke den jevne nordmann i gata. Digitalisering er et område som har vokst med eksponentiell fart, og det er tatt i bruk både private datamaskiner, utlånsutstyr fra skolene i form av laptoper og nettbrett, smartskjerm istedenfor eller i tillegg til tradisjonell tavle, kommunikasjon med foreldre via app heller enn muntlig og brev. Det digitale omslutter oss på alle kanter. Dette er en utvikling som tidlig ble lagt merke til, og allerede for 18 år siden kom en stortingsmelding om at digitale ferdigheter må være en del av opplæringen i skolen:

1.1.1 Kunnskapsløftet

Digitale ferdigheter integrert i alle fag gir muligheter for nye og varierte arbeidsformer og til å knytte ungdomsskolen nærmere videregående skoler, arbeidslivet og andre instanser, blant annet gjennom digital kommunikasjon.

Dette har jo blitt gradvis innført i form av bruk av PCer, nettbrett, smartskjermer og annet. Fra og med høsten 2020 er også ordet programmering brukt som et kompetansemål for elevene. Heller enn å innføre ett fag som heter programmering, er det bestemt at opplæringen av dette skal foregå i flere allerede eksisterende fag. Dette vil si at lærere i matematikk, naturfag, kunst og håndverk, fysikk, kjemi (Vogt, 2021).

På det tidspunktet ble det ikke spesifisert i læreverket hva som skulle gjøres eller hvordan det skulle oppnås. De nye læreplanene i «Kunnskapsløftet 2006» inneholdt ikke noe spesifikt om digitale ferdigheter. Først nå i disse dager, i læreplanene i «Kunnskapsløftet 2020», er digitale ferdigheter og programmering nevnt i flere av læreplanene. Disse innføres gradvis fra skoleåret 2021/2022 og 2022/2023.

1.2 Fagfornyelsen

I arbeid med denne rapporten har vi diskutert mye rundt hvorvidt skolene faktisk har startet med undervisning i programmering, og hvorvidt de kommer til å implementere det etter hvert. Vi har diskutert om programmering burde vært et eget fag, eller om det er fornuftig at flere lærere underviser programmering på tvers.

I fagfornyelsen brukes ordet «programmering» i flere fag, blant annet naturfag, fysikk, matematikk, kunst og håndverk med flere. Likevel har vi ikke på UDIR sine sider funnet at de har definert ordet programmering. Dette er jo en hel diskusjon i seg selv.

Et begrep de har definert, er algoritmisk tenkning. Den første setningen i artikkelen lyder som følger: "Å tenke algoritmisk er å vurdere hvilke steg som skal til for å løse et problem, og å kunne bruke sin teknologiske kompetanse for å få en datamaskin til å løse (deler av) problemet." (Utdanningsdirektoratet, 2019)

Vi har tatt utgangspunkt i at algoritmisk tenkning er en grunnleggende ferdighet som er viktig å ha når man senere skal begynne å kode, og derfor sett på det som en del av vår oppgave og også komme med en innføring til nettopp algoritmisk tenkning når vi skal utforme vårt oppgavehefte for undervisere i fysikk og naturfag.

1.3 Læreplanene

I læreplanen for naturfag er det ikke definert egne kompetansemål for hvert klassetrinn slik det er i enkelte andre fag. I læreplanen for barneskolen nevnes heller ikke programmering, men ordet *reflektering* blir brukt – «*Reflektere over hvordan teknologi kan løse utfordringer, skape muligheter og føre til nye dilemmaer*» (Utdanningsdirektoratet, u.d.c.) – som vi mener kan tolkes som en start på innføring av algoritmisk tenkning.

Heller ikke for ungdomstrinnet er det delt opp i klassenivå, men heller satt opp et felles kompetansemål for hva man skal ha lært i løpet av de tre årene på ungdomsskolen. Her brukes den vide beskrivelsen *«Bruke programmering til å utforske naturfaglige fenomener»* (Utdanningsdirektoratet, u.d.a). Det er altså opp til hver skole, lærer eller lærebokforfatter å velge hva som legges i dette begrepet.

Det er per dags dato ikke alle lærebøker som har programmering inkludert, men det er kommet en del nye bøker på markedet. Vi har sett på:

- Naturfag 8-10 fra Cappelen Damm (Cappelen Damm, u.d.b) ,
- Element 8-10 fra Gyldendal (Gyldendal, u.d.),
- Naturfag SF fra Aschehoug (Brandt et al., 2020),
- Naturfag Påbygging VG3 fra Aschehoug (Brandt et al., Naturfag Påbygging)
- Ergo Fysikk 1 fra Aschehoug (Callin et al., ERGO Fysikk 1, 2021)
- Ergo Fysikk 2 fra Aschehoug (Callin et al., Ergo Fysikk 2, 2022)
- Kraft 1 fra Cappelen Damm (Dellnes et al., 2021)

Gjennomgående i alle, er at programmering ikke er brukt overhode på 8. trinn og 9. trinn, kun i 10. trinn. Når vi har utformet oppgavene for ungdomsskolen, har vi derfor tatt utgangspunkt i fagstoffet mer enn den koden som finnes der. På videregående trinn er det flere programmeringseksempler vi har tatt utgangspunkt i.

1.4 Oppgavens tittel

Arbeidstittel til oppgaveteksten vi først fikk utlevert var «Programmering i STEM-fag: Anvendelse av programmering i fysikk på videregående nivå», med anslått arbeidsmengde estimert til å passe for en student.

Etter samtaler med oppdragsgiver ble oppgaven utvidet til å også dekke naturfag og fysikk på ungdomstrinnet (8-10 klasse). Vår gruppe leverte et forslag til utvidet problemstilling som også inkluderer utarbeidelse av grundige innføringshefter i programmering for lærere som skal undervise på de ulike trinnene, slik at forventet arbeidsmengde er tilpasset to studenter. Vår endelige tittel på oppgaven er

«Programmering i STEM-fag: Anvendelse av programmering i naturfag og fysikk»

- Innføring i programmering og algoritmisk tenkning for lærere i ungdomsskolen og videregående skole.

1.5 Problemstillingen

I den opprinnelige oppgaveteksten vi fikk presentert fra oppdragsgiver, ble det gitt en kort beskrivelse av hvilke behov som fantes, og hva oppgaven hadde som mål å besvare. Denne oppgaven har som mål å:

1. Utvikle grundige innføringshefter i programmering for lærere som skal undervise i naturfag og/eller fysikk på ungdomstrinnet, eller i den videregående skolen.

2. Utvikle gode oppgavehefter i programmering for lærere som de kan benytte i sin undervisning på de ulike alderstrinnene. U8-10, VG1, VG2 og VG3

Etter at programmering nå inngår som pensum i en del av de ulike fagene på skolen, har det umiddelbart oppstått et stort behov for opplæring og videreutdanning av eksisterende lærere i tjenesten. Resultatet er et sterkt behov for gode hjelpemidler til lærerne som skal undervise i fysikk og naturfag i ungdomsskolen og den videregående skole. Lærere som har svært varierende forkunnskaper og kompetanse innen programmering. Vårt håp er at innføringsheftene for ungdomstrinnet og videregående skole skal komme til god nytte, og hjelpe til med å gjøre overgangen til en skolehverdag der digitale ferdigheter er en del av pensum, enklere for lærerne og føre til bedre undervisning for elevene.

Der innføringsheftene er ment til å motivere og hjelpe lærerne til å komme hurtig og godt i gang med prosessen med å benytte programmering i skolen, skal oppgaveheftene forhåpentlig fungere som et alternativ og supplement til den klassiske ordinære undervisningen. Oppgaver som tidligere ble regnet som for tidkrevende eller vanskelig å kalkulere – som numeriske beregninger, kan nå med relativt enkle grep behandles ved hjelp av programvare. Oppgaveheftene starter likevel forsiktig med enkle innføringsoppgaver, men krever etter hvert en del innsats, og gir kanskje mulighet for dybdelæring på en litt ny måte.

2 Teori

2.1 Algoritmisk tenkning

Algoritmisk tenkning er en problemløsningsmetode. En stor del av det å programmere, er å vite hvordan man kan løse et problem ved hjelp av en datamaskin. I den forbindelse er det viktig å vite hvordan man skal gå frem for å løse problemet, og det er her algoritmisk tenkning kommer inn.

I dette delkapittelet vil det retts fokus mot hva algoritmisk tenkning er, hvordan algoritmisk tenkning undervises i skole og hvordan man konkret kan gå frem for å forstå tekstoppgaver og løse de ved hjelp av kode.

Utdanningsdirektoratet (UDIR) starter sin artikkel om algoritmisk tenkning med følgende avsnitt:

> Å tenke algoritmisk er å vurdere hvilke steg som skal til for å løse et problem, og å kunne bruke sin teknologiske kompetanse for å få en datamaskin til å løse (deler av) problemet. I dette ligger også en forståelse av hva slags problemer/oppgaver som kan løses med teknologi og hva som bør overlates til mennesker. Algoritmisk tenkning er den norske oversettelsen av det engelske computational thinking (Utdanningsdirektoratet, 2019).

Videre skriver UDIR om hvordan man kan bruke algoritmisk tenkning til «å bryte ned komplekse problem til mindre, mer håndterlige delproblemer som lar seg løse.» Dette vil være en viktig del av vår rapport – å lære lærere hvordan kan bryte ned oppgaver til mindre deloppgaver, slik at de kan skrive kode for hvert enkelt av deloppgavene, for så å sette disse sammen igjen. På den måten vil det større problemet kunne løses på en steg-for-steg-måte.

Figur 2-1: Utdanningsdirektoratet har på sine sider en plakat man kan laste ned om algoritmisk tenkning (Utdanningsdirektoratet, 2019).

Det blir også presisert at det er viktig å få tak å hva som faktisk er problemet, og hva som er unødvendige detaljer. Til dette finnes det mange problemløsningsmetoder. Et eksempel på dette er Polya sin metode for å løse matematiske problemer (Polya, 1973), og som på mange måter er en metode for å løse problemer ved hjelp av algoritmisk tenkning. Den er som følger:

1. Forstå problemet
2. Lag en plan
3. Gjennomfør planen
4. Evaluer planen

Paul Vickers har laget en oppskrift for programmerere, hvor han har tatt Polyas metode to steg videre:

5. Beskriv hva du har lært
6. Dokumenter løsningen.

Denne 6-trinns-metoden kalles «How to think like a programmer (HTTLAP)», eller «Hvordan tenke som en programmerer» på norsk. Følgende er en oversettelse av Vickers sin sekstrinnsmodell (Vickers, u.d.):

Tabell 2-1: How to think Like a Programmer (HTTLAP)

1. Forstå problemet	
Du må forstå problemet. Gå ikke videre til neste nivå før du har gjort dette.	Hva blir du bedt om å gjøre? Hva er nødvendig? Prøv å start problemet om igjen? Kan problemet bli bedre uttrykt med en tegning eller et diagram eller et bilde? Ved å bruke matematiske notasjoner? Ved å bygge en modell av tre, papir eller papp? Hva er det ***uvisse***? Er å finne den ukjente en del av problemstillingen? Skriv ned det du **vet** om problemet. Har du gjort noen antagelser? Hvis ja, hva kan/burde du kjøre med det?
Sov på problemet og kom tilbake til det med nye øyne.	Hva er hoveddelene av problemet? Er det et problem med flere deler?
2. Utarbeide en plan for å løse problemet	
Begynn å tenke på informasjonen du har, og hva løsningen skal gjøre.	Har du løst dette problemet før (kanskje med andre verdier/mengder? Er problemet ***lignende*** på et du har møtt før? Hvis ja, kan du bruke noe av kunnskapen fra det problemet her? Gjelder løsningen på det problemet for dette på noen måte? Er noen deler av problemet enklere å løse enn andre? Hvis problemet er for vanskelig, kan du løse en enklere versjon av det, eller et relatert problem? Vil det å gjenta problemet (kanskje å fortelle det med dine egne ord til noen andre) hjelpe deg å få tak i det? Prøv å beskrive problemet i et annet ***språk*** (f.eks. skjematisk, billedlig, ved å bruke matematikk, bygge en fysisk modell eller en representasjon).

Slutt å bry hjernen din og sov på problemet.	Brukte du all informasjonen fra problemformuleringen? Kan du tilfredsstille alle forventningene til problemet? Er det noe du har utelatt?
3. Gjennomføre planen	
Skriv ned løsningen din. Legg merke til ting gjort i orden (rekkefølge), ting gjort betinget (seleksjon) og ting gjort gjentatte ganger (iterasjon).	Skriv ned den grunnleggende rekkefølgen av handlinger nødvendig for å løse det generelle problemet. Dette kan inkludere å gjemme noen av detaljene for å få den overordnede rekkefølgen riktig. ER rekkefølgen av handlinger riktig? Avhenger rekkefølgen av at noen ting må være sanne? Hvis ja, vet du disse tingene fra problemformuleringen? Eller har du gjort noen antagelser? Hvis du har gjort antagelser, hvordan vil du verifisere at de er riktige? Hvis du ikke kan se noen løsning på hele problemet, kan du se noen deler av problemet du kan løse? Hvis problemet er for komplisert, prøv å ta vekk noen av vilkårene/begrensningene og se om det gir deg en vei inn. Gå tilbake til rekkefølgen av handlinger. Burde alle handlinger utføres i ***alle*** omstendigheter? Burde noen av handlingene (eller grupper/blokker av handlinger) bare bli utført dersom visse vilkår er møtt? Gå tilbake til rekkefølgen av handlinger. Er det nok å utføre hver handling bare én gang for å gi ønsket utfall? Hvis nei, har du handlinger (eller grupper/blokker av handlinger) som dermed må ***repeteres***?
Sov på det igjen.	Gå tilbake enda en gang. Hører noen av handlingene/blokkene til ***inni*** andre? For eksempel, har du en blokk av handlinger som må repeteres, men bare hvis noen vilkår er møtt?
4. Evaluere resultatet	
Undersøk resultatene du får når du bruker løsningen din.	Bruk løsningen din for å møte kravene til problemet. Fikk du det riktige svaret eller det riktige utfallet? Hvis ikke, hvorfor ikke? Hvor gikk løsningen din feil? Hvis du tror du fikk det riktige svaret, hvordan kan du være sikker? Virket noen deler av løsningen tungvint eller lite fornuftig? Kan du gjøre de delene enklere, raskere og tydeligere?
Få noen andre til å bruke eller følge løsningen din.	Gi løsningen din til noen andre og be dem om å bruke den for å gjennomføre oppgaven. Var instruksjonene tydelige nok for dem? Trengte de å be om hjelp eller avklaring? Misforstod de noen av instruksjonene? Hvis ja, hvorfor? Fikk de det samme svaret som deg? Hvis ikke, hvem sitt svar var riktig?
5. Beskrive hva du har lært	
Lag en oversikt over prestasjonene dine og vanskelighetene du har møtt.	Hva har du lært dra denne øvelsen? Hva vet du nå som du ikke visste før du startet?

	Hvilke spesifikke vanskeligheter møtte du på? Var det noen aspekter av problemet som skapte særlige vanskeligheter? Hvis ja, tror du at du vet hvordan du skulle taklet det om du møtte på noe lignende i fremtiden? Sammenlign din ferdige løsning med ditt første forsøk. Hva lærer forskjellene deg?
6. Dokumentere løsningen	
Forklar løsningen din. Forsikre deg om at den blir forstått.	Er det noen aspekter ved løsningen som kan være vanskelig å forstå? Er det fordi de er dårlig skrevet, eller ganske enkelt fordi løsningen er så komplisert? Hvis du skulle finne frem igjen løsningen din om fem år, er det noen deler som vil være vanskelige å forstå?

Vi har tatt utgangspunkt i denne tenkemåten når vi har utformet oppgaveheftene, men ikke fulgt Vickers oppskrift fullstendig. Vi har utarbeidet en litt enklere gjennomgang basert på denne oppskriften.

2.1.1 Algoritmisk tenkning i naturfag og fysikk

Kjerneelementet «Teknologi» i læreplanen til Naturfag er følgende (Utdanningsdirektoratet, u.d.c):

> Elevene skal forstå, skape og bruke teknologi, inkludert programmering og modellering, i arbeid med naturfag. Gjennom å bruke og skape teknologi kan elevene kombinere erfaring og faglig kunnskap med å tenke kreativt og nyskapende. Elevene skal forstå teknologiske prinsipper og virkemåter. De skal vurdere hvordan teknologi kan bidra til løsninger, men også skape nye utfordringer. Kunnskap om og kompetanse innenfor teknologi er derfor viktig i et bærekraftsperspektiv. Arbeid med kjerneelementet teknologi skal kombineres med arbeid knyttet til de andre kjerneelementene.

Her kommer det tydelig frem at programmering er et av kjerneelementene man skal lære seg i løpet av 11 år med naturfag, fra 1. klasse på barneskole til 1. klasse videregående nivå. Ordene «algoritmisk tenkning» er ikke skrevet eksplisitt, men både å forstå, skape og bruke teknologi, samt at dette skal kombineres med å tenke kreativt og nyskapende, er elementer som inngår i det å tenke algoritmisk. Som UDIR selv skriver, er det å tenke algoritmisk «å vurdere hvilke steg som skal til for å løse et problem». Man må forså et problem for å kunne skape en løsning til problemet ved å bruke teknologi. På samme side skriver UDIR at «Den algoritmiske tenkeren må være systematisk og analytisk i sitt arbeid, men det er minst like viktig å være skapende, eksperimenterende og åpen for alternative løsninger.»

Kjerneelementet «Praksiser og tenkemåter i fysikk» i læreplanen for Fysikk er følgende (UDIR):

> Kjerneelementet praksiser og tenkemåter i fysikk handler om hvordan naturvitenskapelige metoder, eksperimenter, teorier og modeller blir utviklet og brukt. Kjerneelementet handler også om å bruke programmering, eksperimenter, teorier og modeller for å forstå fysiske sammenhenger og fenomener. Videre

> handler det om å bruke og veksle mellom ulike representasjonsformer for å belyse og forstå teorier og modeller (Utdanningsdirektoratet, u.d.a).

På samme måte nevnes programmering, uten at algoritmisk tenkning er nevnt. Poenget er dog det samme: Man kommer ikke unna algoritmisk tenkning om man ønsker å programmere. Derimot er det ikke slik at dersom man kan å tenke algoritmisk, så kan man å programmere. Programmering er så mye mer enn det, og som diskuteres nærmere i et senere delkapittel. Først mer om hvordan man bruker algoritmisk tenkning i dagliglivet.

2.1.2 Algoritmisk tenkning i praksis

På samme måte som man ikke kan å programmere kun fordi man kan å tenke algoritmisk, brukes ikke algoritmisk tenkning kun i programmering. Også i dagliglivet bruker man algoritmisk tenkning til ulike oppgaver. Statped forklarer algoritme på følgende måte:

> En algoritme er helt enkelt forklart, en oppskrift. Det er en steg-for-steg-plan for å få noe gjort. Hvis algoritmen skal utføres av en datamaskin vil den at du skal skrive stegene med kommandoer som datamaskinen forstår. Dette kaller vi et programmeringsspråk. Når du er i utlandet og snakker engelsk kan det hende at du ikke er helt nøyaktige med grammatikk eller ordvalg, men likevel forstår andre hva du mener. Slik er det ikke for en datamaskin. Den må få beskrevet alle stegene i riktig rekkefølge. Oppskriften, eller algoritmen, må være helt nøyaktig (Statped, 2021).

Videre kommer de med dagligdagse oppgaver som kan løses ved hjelp av algoritmisk tenkning, nemlig strikking og baking. Når man strikker, kan man si at man følger Polya's oppskrift for algoritmisk tenkning:

1. Forstå problemet: Forstå oppskriften på det du ønsker å strikke. Er det noe som er uklart? Følger det med et bilde? Forstår du bildet? Hva kan du allerede, og hva må du lære deg underveis? Hvilke teknikker brukes? Antar oppskriften at du kan noe fra før, eller er alt grundig forklart?
2. Lag en plan: Har du strikket før? Var det en annen vanskelighetsgrad enn plagget du nå har bestemt deg for å strikke? Er det noen deler av strikkingen som er lettere å gjennomføre enn andre? Er det for vanskelig, eller greier du å gjennomføre den? Trenger du hjelp fra noen andre underveis i prosjektet? Har du tatt hensyn til all informasjonen oppskriften oppgir?
3. Gjennomfør planen: Start strikkingen slik du har planlagt den. Er det noe som viser seg å være vanskeligere enn det du opprinnelig trodde? Og i tilfelle man strikker feil; Gå tilbake til punktet før feilen oppstod og gjør det på nytt.
4. Evaluer planen: Er ditt endelige produkt det samme som oppskriften tilsier? Ble det riktig i forhold til den, eller gjorde du endringer underveis? Er du mer fornøyd med ditt produkt enn det produsenten av oppskriften hadde? Er det noen deler av strikkingen som kunne blitt gjort lettere? Kunne oppskriften vært tydeligere noe sted?

Den samme typen oppskrift kan brukes når man baker eller gjøre andre hverdagslige oppgaver. De fleste har i en eller annen situasjon brukt algoritmisk tenkning, uten at

man stopper opp og tenker «Hei, nå tenker jeg på en algoritmisk måte». Likevel er det denne tenkemåten man kan overføre til programmering og skolearbeid.

2.2 Programmering

I dette avsnittet vil vi forsøke å belyse hva programmering faktisk er. "Senter for IKT i utdanningen", som nå er fusjonert med Utdanningsdirektoratet, utarbeidet i 2016 et notat kalt "Programmering i skolen". Dette var i forbindelse med arbeidet med det nye kunnskapsløftet og de nye læreplanene, hvor de tar for seg ulike tilnærminger til å inkludere programmering i skolen. I den forbindelse har de også et avsnitt om hva de mener programmering er:

> Programmering, slik det er brukt i [notatet "Programmering i skolen"], omfatter mer enn å bare skrive programkode som kan kjøres på en datamaskin, det inkluderer også prosessen med å komme fram til denne koden. Det vil si prosessen fra å identifisere et problem og tenke ut mulige løsninger på problemet, til å skrive kode som kan forstås av en datamaskin, og å feilsøke og kontinuerlig forbedre denne koden (Sevik m.fl., 2016).

Og det er ikke kun i dette notatet at ordet programmering brukes på denne måten. Det er viden brukt i hele verden. Folkeuniversitetet har på sine sider en artikkel som heter "Hva er egentlig programmering?". Denne er myntet på mennesker som vurderer å studere programmering. På artikkelsiden har de følgende formuleringer:

> Å *programmere* vil si å instruere en datamaskin til å utføre bestemte oppgaver for oss. Det lar oss automatisere arbeid, analysere store mengder informasjon eller å utvikle ny teknologi. Når vi programmerer utvikler vi *dataprogrammer*, som består av en rekke instrukser som datamaskinen forstår og utfører. En som jobber med programmering kalles derfor gjerne for en *utvikler*. […] For å utvikle programmer er det aller viktigste steget å kunne bryte opp større oppgaver i mindre biter som kan løses hver for seg, steg for steg. Det handler altså om problemløsning, analytisk tenkning og ikke minst kreativitet, viktige ferdigheter i dagens samfunn.

Også her tas det opp at programmering er så mye mer enn bare det å skrive kode. Det er også alt forarbeidet og alt etterarbeidet, det er tankegangen og det er analyseringen, problemløsningen, feilsøkingen, rettingen, testingen, kreativiteten og så mye mer enn bare den faktiske kodingen. "Kodingen" er også et vidt begrep, som består av flere av de nevnte egenskapene.

3 Valg av metode

3.1 Design Science Methodology

Som vår forskningsmetode har vi valgt Design Science.

> Design science is the design and investigation of artifacts in context. The artifacts we study are designed to interact with a problem context in order to improve something in that context (Wieringa, 2014, s. 3).

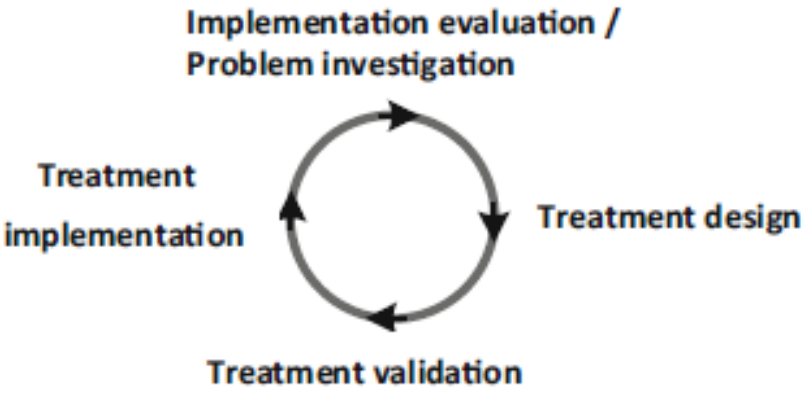
Wieringa skriver også at et designproblem er et problem hvor man (re)designer en artefakt slik at den på en bedre måte bidrar til å oppnå et mål (Wieringa, 2014, s. 15). Denne metoden sammenfaller godt med vår oppgave, som består i å undersøke hva som finnes av eksisterende programmeringsoppgaver i fysikk og naturfag, for deretter å på en bedre måte lære læreren hvordan den kan undervise i programmering i sitt fag. Vi skal redesigne eksisterende oppgaver. For vårt tilfelle vil det være både å forklare eksisterende programmeringsoppgaver, samt og lage programmeringsoppgaver basert på andre oppgaver i undervisningsmateriellet som kan være relevante å løse med programmering.

Design Science-metoden består av flere oppgaver som gjentas iterativt. I hovedsak er det snakk om tre oppgaver:

- Problemundersøkelse (eng: problem investigation)
- Behandlingsdesign (eng: treatment design)
- Behandlingsvalidering (eng: treatment validation).

Denne syklusen med tre oppgaver er del av en større syklus, kjent som designsyklusen. I den er målet å validere behandlingen (eng: validated treatment), det vil si å bruke artefaktet (eng: artifact) i den virkelige verden og evaluere resultatene (Wieringa, 2014, s. 27).

Med artefakt menes det noe som er laget av mennesker av et praktisk behov (Wieringa, 2014, s.29). Med behandling menes det at artefaktet samhandler med et problem for å løse problemer (Wieringa, 2014, s. 28).



Figur 3-1: Designsyklusen. Forenklet versjon basert på Wieringas ingeniørsyklus (2014, s. 28).

Vårt prosjekt er bestilt av NTNU AIT med den opprinnelige tittelen «Programmering i STEM-fag: Anvendelse av programmering i fysikk på ungdomstrinnet og videregående nivå». Utgangspunktet vårt, eller problemundersøkelsen vår, starter derfor her. At lærere på grunnskoletrinnet og videregående nivå skal undervise i programmering i tillegg til det faget de opprinnelige underviste i. Dette gjelder ikke alle fag, men for flere av de eksisterende fagene, som matematikk, kjemi, fysikk, naturfag med flere.

Det innledende arbeidet blir derfor å finne ut mer om hvilken opplæring lærerne har fått, eller mer hvilke materiell de har tilgang til, både fysiske og digitale. Vår oppdragsgiver har satt som utgangspunkt at lærerne ikke får noen spesifikk opplæring i hvordan de skal lære elevene programmering.

Videre i behandlingsdesignfasen har vår oppgave vært å finne eksisterende programmeringsoppgaver i læremateriellet for de ulike trinnene, både i naturfag for ungdomstrinnet og 1. klasse videregående, og i fysikk for 2. og 3. klasse videregående skole. Der det ikke har eksistert tilstrekkelig antall kodeoppgaver, har vi valgt å utvikle egne oppgaver basert på det pensumet som eksisterer.

I behandlingsvalideringsfasen har vi sørget for at det naturlig progresjon i vanskelighetsnivået for oppgavene, og også at algoritmisk tenkning er en del av prosessen fra start til slutt. En viktig del av valideringen er også å teste det vi har utviklet mot brukere, altså lærere i tjenesten. Det lar seg dessverre ikke gjennomføre i løpet av den tiden vi skriver på denne rapporten.

Det siste nivået i figuren, behandlingsimplementering (eng: treatment implementation), er da heller ikke gjennomførbart. Selve implementeringen vil skje etter at vår oppgave er fullført, men vi håper det vil være mulig å enten bruke direkte, eller å videre forbedre slik at vårt arbeid når sitt mål: Å være et opplæringshefte for lærere i tjenesten skal lære elevene programmering i naturfag og fysikk.

Selv om flere av oppgavene i designsyklusen må utelates fra vårt arbeid, er den likevel et godt verktøy for oss i vårt arbeide. Den har også blitt brukt iterativt i form av at vi selv har testet vårt produkt, og gått tilbake til tidligere sykluser for å forbedre vårt arbeide.

4 Valg av teknologi

Under analysefasen så vi på en rekke ulike verktøy og teknologier, med mål om å best hjelpe lærerne å komme i gang med å lære seg programmering. Ulike programvare-løsninger ble vurdert, fra pakker som Anaconda, til enkle stand-alone programmer en laster ned lokalt på egen datamaskin. Micro:bit, Raspberry Pi og Arduino ble vurdert, for å lage programmer og utføre målinger, som kobler praktiske eksperimenter sammen med programmering og tyding av resultater. 3d-visualisering i Unity og Unreal Engine, ble vurdert, for å lage spennende illustrasjoner og fysiske modeller, men det var likevel hovedsakelig 2 hovedmoment som gjorde at vi landet på Scratch og Python

1) teknologien må tilføre noe som hjelper en i utførelsen av programmering i undervisningssituasjonen

2) teknologien må være allment tilgjengelig uten at det medfører krav til innkjøp av utstyr, inngåelse av abonnementer, eller nedlastning av programvare som krever oppdateringer, og vedlikehold.

Både Scratch som første innføring i programmering, og Python som er populært blant alle fra den pure nybegynner til den riktig erfarne utvikleren, treffer begge disse kriteriene. Begge løsningene er meget lette å komme i gang med. Det er en lav inngangsterskel, ettersom en kun trenger å åpne en nettside og så er man i gang. I tillegg finnes det store mengder av dokumentasjon, bøker, videoer, hele kurs en kan følge på plattformer som YouTube, egne kurs, og ikke minst et stort fellesskap av store og mindre hjelpe-grupper med felles interesse for Python, eller Sctratch.

Vi vurderte lenge også å lage en egen liten videoserie, men kom frem til at det egentlig ikke ville tilføre noe mer innhold til læringen i denne start-fasen som lærerne selv nå står i. Da er nok enkle informative steg-for-steg innføringshefter det optimale, og vi håper våre kommer til god nytte.

5 Resultater

I problemstillingen kom vi frem til at … (sjekk kapittel 1.5, når det er limt inn).

Vi har derfor utarbeidet et innføringshefte/opplæringshefte for lærerne både i Scratch og Python for ungdomsskoletrinnet, og en fortsettelse av Python for den videregående skole. Det er også utarbeidet et oppgavehefte for ungdomsskolen og tre oppgavehefter for videregående skole, ett for hvert av trinnene VG1, VG2 og VG3.

I heftet for ungdomsskolen er oppgavene av en ganske generell karakter, mer tenkt for å gi en innføring i programmering og koding. Noen oppgaver har tema fra naturfag.

I heftene for videregående skole er temaet naturfag/fysikk for VG1, og fysikk for VG2 og VG3.

I de neste punktene kommer følgende:

5.1 Innføringshefte i Scratch og Python for ungdomsskolen

5.2 Oppgavehefte i Scratch og Python for ungdomsskolen

5.3 Innføringshefte i Python for videregående skole

5.4 Oppgavehefte i Python for VG1

5.5 Oppgavehefte i Python for VG2

5.6 Oppgavehefte i Python for VG3

5.1 Innføringshefte i Scratch og Python for ungdomsskolen

Scratch

Lær Kidsa Koding! er en frivillig bevegelse som "vil hjelpe de unge til ikke bare å bli brukere, men også skapere med teknologien som verktøy" (Lær Kidsa Koding, u.d.). En viktig del av deres formålsparagraf, er at de ønsker å bidra til rekruttering til IT-yrkene og realfagene. På deres nettsider finner man både veiledninger, oppgaver, kodeklubber og mye annet som er relevant. Det kan brukes både av undervisere og av elevene selv, og er et godt sted å starte.

På nettsidene til Lær Kidsa Koding! har de lagt ut oppgaver for mange ulike programmeringsspråk eller verktøy, som for eksempel Scratch, Python, micro:bit, LEGO Mindstorms. Scratch og micro:bit blir også brukt i lærebøkene Element 10 fra Gyldendal (Arntzen et al., 2022) og Naturfag 10 fra Cappelen Damm (Steineger et al., 2021) . Vi ser det derfor som nyttig at vi tar for oss Scratch i denne gjennomgangen. Dette fordi det er et lavterskeltilbud, hvor man også kan bruke gode online varianter for å skrive kode. Micro:bit krever innkjøp av utstyr, og vi ønsker at hvem som helst kan kunne starte med programmering, uten å måtte laste ned programmer eller kjøpe ekstra utstyr.

Online-varianten av Scratch vi har valgt å bruke ligger på domenet til Massachusetts Institute of Technology (MIT): https://scratch.mit.edu/. Nederst på siden kan man velge å stille inn språket til norsk. Det er også online-varianten Lær Kidsa Koding! bruker for sine eksempler. De sier at "Scratch er et grafisk programmeringsspråk utviklet spesielt for at barn og unge enkelt skal lære seg programmering" (Lær Kidsa Koding, u.d.) – nøyaktig det samme vi ønsker å bidra til med denne bacheloren.

5.1.1 Å bruke Scratch

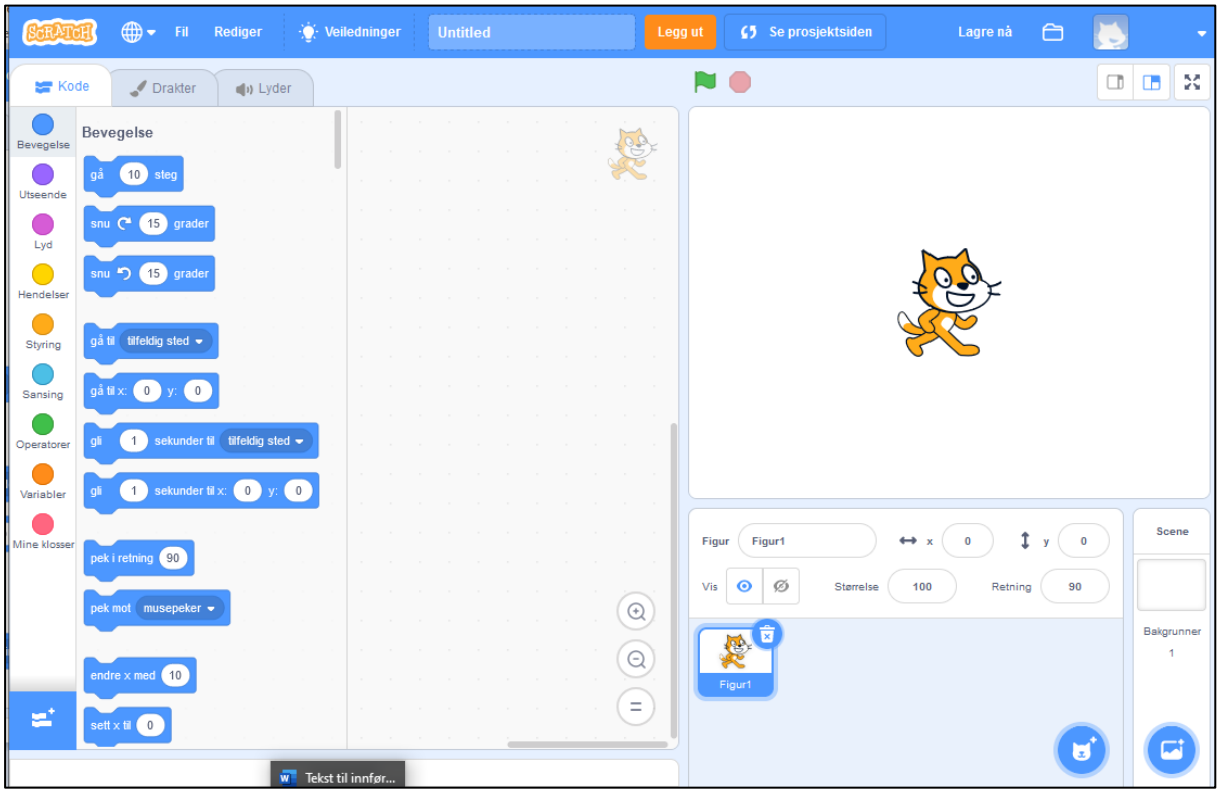
For å komme i gang med Scratch vil vi her gå gjennom en enkel oppgave man kan gjøre sammen med elevene. Det er en enkel figur som skal spørre hva de heter, og hva deres favorittfag er.

Gå til https://scratch.mit.edu.

Nederst på siden kan man velge om man vil bruke Scratch på bokmål, nynorsk, engelsk eller 70 andre språk. Det kan være lurt å tenke over om man ønsker å lære elevene å skrive kode på norsk eller engelsk. Om/Når de senere skal lære seg tekstbasert programmering, er sannsynligheten stor for at de må kode på engelsk. Vi kommer ikke med noe anbefaling til hva dere skal gjøre, men anbefaler at dere selv tenker over hva dere vil.

Om valget faller på norsk, vil vi tipse om at man også kan endre språk etter at koden er ferdigskrevet. På den måten kan elevene veksle mellom norsk og engelsk, og lære seg de engelske begrepene til senere bruk. Vi har valgt å skrive denne gjennomgangen med Scratch innstilt på norsk bokmål.

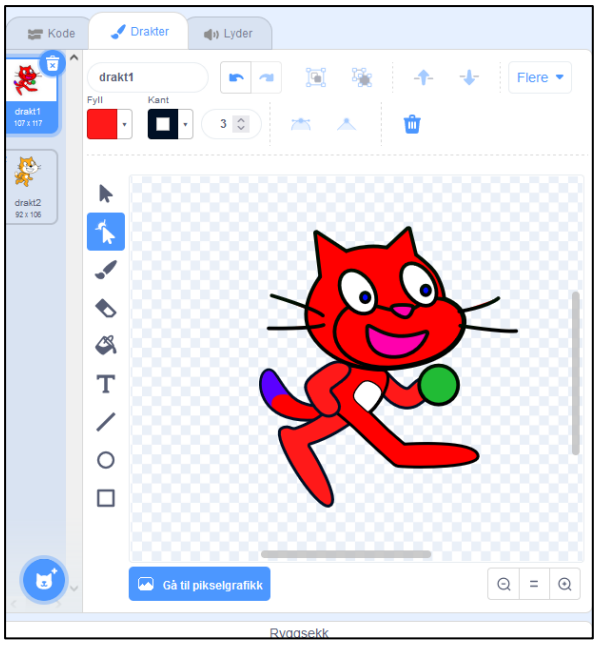
Når du er inne på sidene til Scratch, trykk på "Programmering". Da vil siden se slik ut:



Figur 5-1: Landingssiden for Scratch Programmering

Her er det mange fane-valg, og det kan være lurt å trykke litt rundt på siden for å se hvilke funksjoner som er tilgjengelige.

I "Drakter"-fanen kan man for eksempel endre på den eksisterende figuren, både endre farger, størrelse og annet, eller man kan legge til elementer (som den grønne ballen i hånden hans):

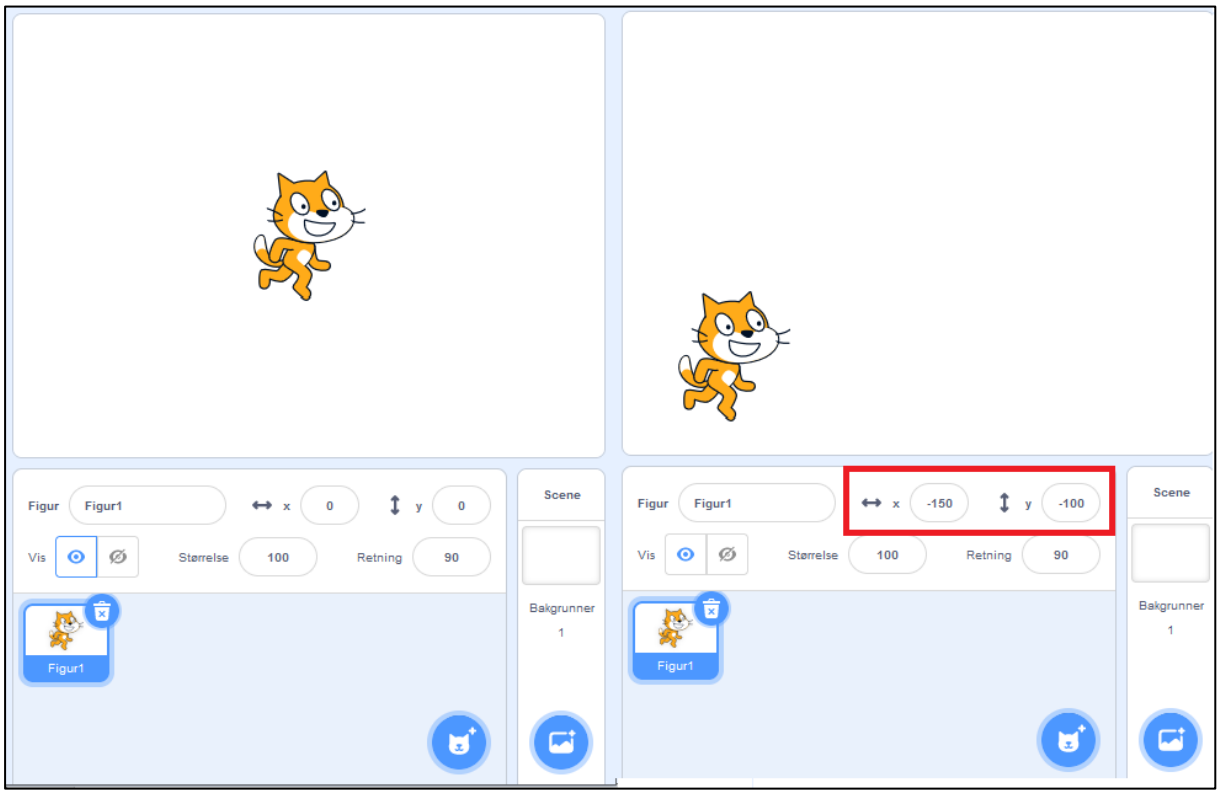


Figur 5-2: Lage egen figur i Scratch

Eller man kan starte helt på nytt og tegne en egen figur på frihånd.

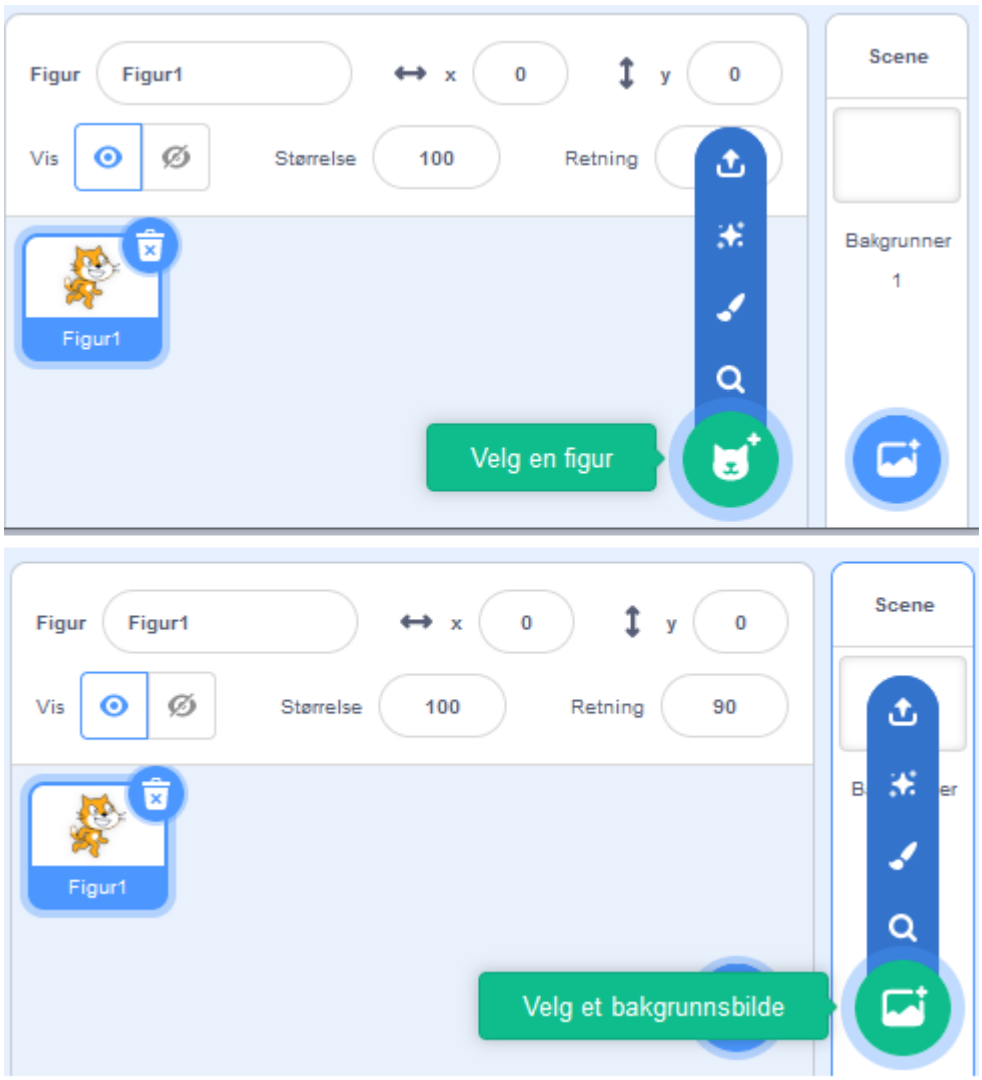
I "Lyder"-fanen kan man endre på lyder som allerede finnes i biblioteket, eller man kan bruke mikrofon og spille inn helt egne lyder.

I vinduet til høyre kan man endre på hvor stor figuren skal være, hvilken retning den skal kunne bevege seg, hva den heter og andre innstillinger:



Figur 5-3: Innstillinger for startposisjonen til figuren i Scratch

Nederst til høyre kan også velge en annen figur, velge et eksisterende bakgrunnsbilde, eller laste opp, tegne eller bli tildelt en tilfeldig figur eller bakgrunnsbilde:



Figur 5-4: Velge andre figurer og bakgrunnsbilder i Scratch

Som skrevet innledningsvis, anbefaler vi at dere trykker litt rundt og blir kjent med siden før dere setter i gang.

5.1.2 Scratch kodeprosjekt

Etter en kikk rundt på sidene, er det på tide å programmere det første programmet. Vi har valgt å vise veldig enkle programkoder her i Scratch, for så i neste omgang å vise stort sett de samme programkodene i Python.

5.1.2.1 Scratch eksempel 1 - En enkel innføring: Favorittfag

I første program skal vi ha en enkel dialog med katten:

Gå til kode-menyen "Hendelser". Her kan man velge hvordan programmet skal starte. Vi velger "når denne figuren trykkes". Under fanen "Sansing" kan man få figuren til å stille et spørsmål og vente. Når man stiller et spørsmål, betyr det at brukeren skal skrive inn et svar. Derfor vil figuren også vente med å gå videre til neste kommando til brukeren har trykket på Enter.

Vi ønsker at figuren skal spør brukeren hva den heter, og hva den har som favorittfag. Kodeblokken vil kanskje se noe slik ut:

Om man kjører denne koden, vil man ikke oppfatte at katten sier "Hyggelig å møte deg" og "Det liker jeg også". Det er fordi det ikke er lagt inn noe venting, og programmet kjører direkte videre til neste linje. Det er derfor viktig å bruke programkoder som angir hvor lenge teksten skal stå der, når man har flere handlinger like etter i koden.

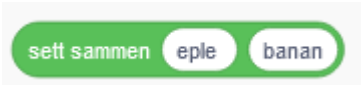
Figur 5-5: Kode uten vente-signal

Bytt ut de to blokkene:

Nå vil man kunne se all dialogen, men likevel er man kanskje ikke helt fornøyd. For hvor ble det av de svarene man ga til katten?

Figur 5-6: Kode med ventesignal

Ved å bruke en operator (grønn), kan man sette sammen egen tekst, og det svaret man ga til katten. Vi ønsker å bruke operatoren som ser slik ut:

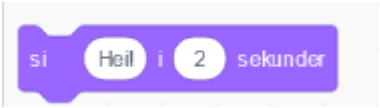


Figur 5-7: Sett sammen-operator

Da kan man legge inn både egen tekst, og blokker som allerede finnes i Scratch. Rett under kommandoen "Spør … og vent", ligger det en liten boks som heter svar. Denne skal vi nå bruke sammen med "sett sammen"-blokken ovenfor.

Figur 5-8: Spør- og svar-elementer

Disse skal begge legges inn i den lilla "Si … i … sekunder" fra tidligere:



Figur 5-9: Si ... i ... sekunder

Til sammen blir det da seende slik ut:

Figur 5-10: Sette sammen ulike elementer

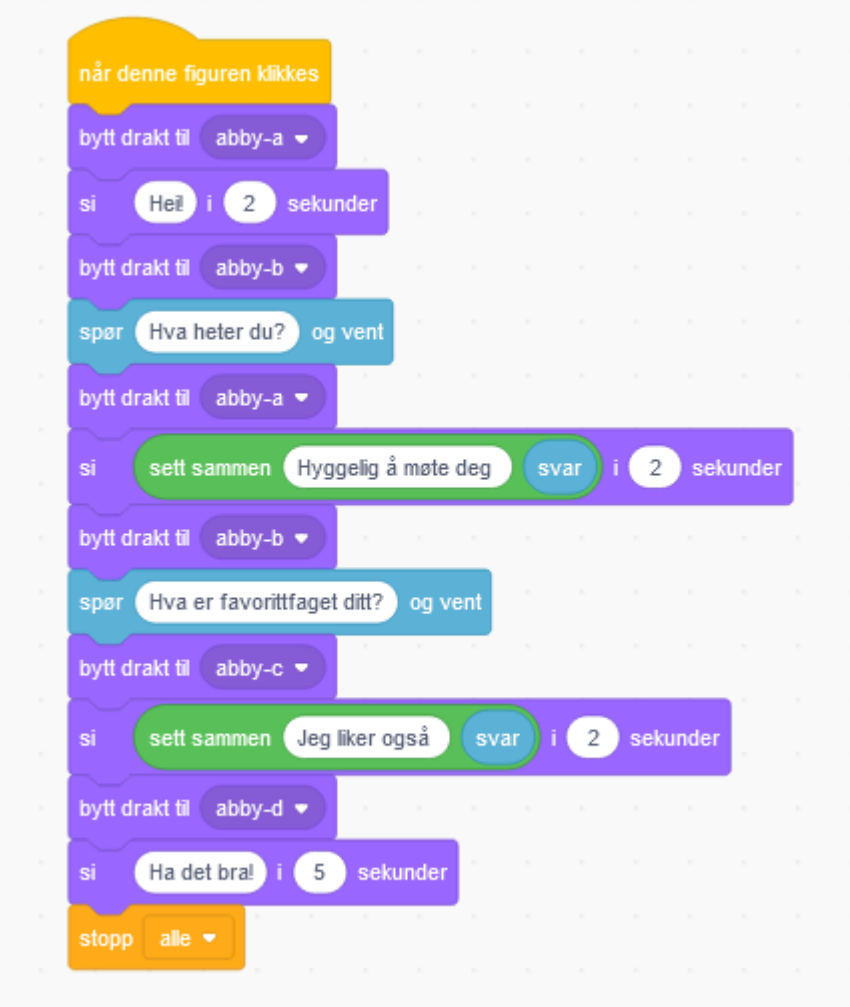
Vi gjør det samme under spørsmålet "Hva er favorittfaget ditt?"

Til sammen ser koden nå slik ut:

Figur 5-11: Scratch program nummer 1

Figur 5-12: Skjermdump fra Scratch program nummer 1

Denne koden fungerer som planlagt, og man kan si seg ferdig. Man kan også bygge videre på den. Ved å velge en annen figur, en bakgrunn og noen bevegelser figuren skal gjøre, kan man få en litt morsommere visualisering.



Figur 5-13: Scratch program nummer 1, med styling

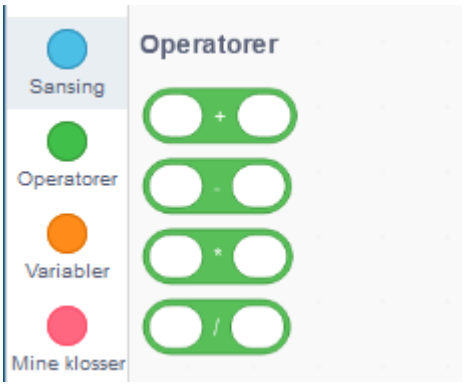
Med denne koden, og bakgrunnen "Chalkboard", vil man få opp dialogen som er gjengitt i bildene her. Der kan dere se at figuren endrer positur mellom hver gang hun sier noe. På lignende måte kan man endre bakgrunn underveis, og på den måten for eksempel gå fra klasserommet til et annet sted, som utenfor skolen, til stranda, i skogen eller mange andre situasjoner.

Figur 5-14: Skjermdump fra Scratch program nummer 1, med styling

5.1.2.2 Scratch eksempel 2 - Bruk av operatorer: Konvertering fra Celsius til Kelvin

Man kan også bruke Scratch til å konvertere mellom verdier som Celsius, Kelvin og Fahrenheit, eller meter per sekund til kilometer per time.

I denne gjennomgangen viser vi litt raskere hvilken kode som er brukt, som dere kan bygge videre på i undervisningen.



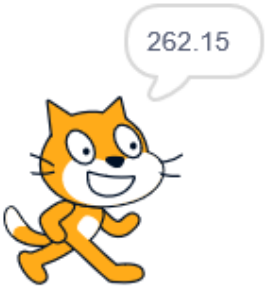
Figur 5-15: Operatorer i Scratch

Man kan gjøre det helt enkelt ved å spørre hvor mange grader Celsius man ønsker å konvertere, for så å gi svaret direkte uten noe mer informasjon. Man gjør dette ved hjelp av operatorer som finnes i Scratch, i dette tilfellet addisjon.

Å konvertere fra Celsius (C) til Kelvin (K) kan man gjøre med formelen K = C + 273,15. I Scratch skriver man desimaltall med punktum istedenfor komma, altså K = C + 237.15.

Denne koden kan gjøres veldig enkel ved å angi hvordan programmet skal startes, spør hvor mange grader Celsius man vil konvertere, og deretter gi svaret direkte. Eksempel for 25 grader Celsius:

Figur 5-17: Bruk av operatorer



Figur 5-16: Skjermdump fra bruk av operatorer

Man kan også bruke en "Sett sammen"-blokk slik som tidligere for å legge til litt forklarende tekst. Dialogen blir da:

- Hvor mange grader Celsius?
- 25
- I Kelvin er det 262.15

Figur 5-18: Bruk av operatorer og tekst sammen

Figur 5-19: Skjermdump fra bruk av operatorer og tekst sammen

5.1.2.3 Scratch eksempel 3 - Å sette sammen mange elementer: Konvertering fra Celsius til Fahrenheit

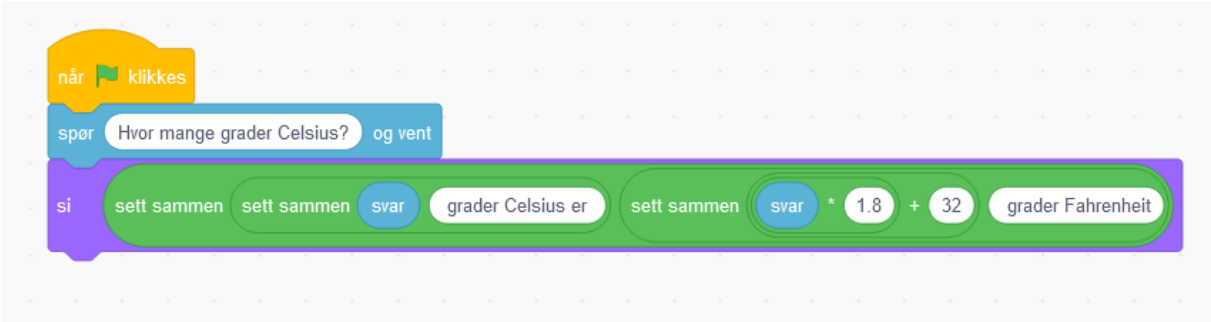
Her skal vi vise hvordan man kan bygge sammen flere operatorer for å regne ut litt mer kompliserte regnestykker. Her får vi bruk for både addisjon og multiplikasjon i samme regnestykket for å konvertere fra Celsius (C) til Fahrenheit (F). Formelen for dette er F = (C *1.8) + 32.

Den enkle koden kan da se slik ut, og med input 25 grader Celsius vil katten ganske enkelt svare 77:

Figur 5-21: Konvertere Celsius til Fahrenheit

Figur 5-20: Skjermdump fra konvertere Celsius til Fahrenheit

Også her kan man gjøre det om til å ligne mer på en naturlig samtale. Den koden vil se litt mer komplisert ut i Scratch, men det er enkelt å bygge videre med klosser inni hverandre:



Figur 5-22: Konvertere Celsius til Fahrenheit med samtale

Figur 5-23: Skjermdump fra konvertere Celsius til Fahrenheit med samtale

En mer forklarende figur over hvilke blokker som må settes inn, og hvor, for å få til dette:

Figur 5-24: Visualisering av hvilke elementer som må settes sammen

5.1.2.4 Scratch eksempel 4 - Bruk av variabler: Konvertering fra Fahrenheit til Celsius

En litt mer oversiktlig måte å gjøre det samme på, er å bruke variabler. I Scratch kan man lage egne variabler, og bruke de på ulike måter. Her vil vi vise noen av dem.

VI starter med å lage to variabler: En som heter Celsius, og en som heter Fahrenheit. Variablene vil da dukke opp i samme vindu som figuren. Man kan velge om disse skal være synlige eller ikke. Vi velger her å la de vises. Du kan se at det er krysset av ved siden av "Celsius" og "Fahrenheit", men ikke "min variabel":

Figur 5-26: Opprette egen variabel

Figur 5-25: Variabelen vises i dialogvinduet.

Vi kan nå bruke disse i beregningene vi skal gjøre. Formelen for å gjøre om Fahrenheit (F) til Celsius (C) er som følger: C = (F - 32) / 1,8. Som vi husker må vi bruke punktum istedenfor komma i desimaltall i Scratch.

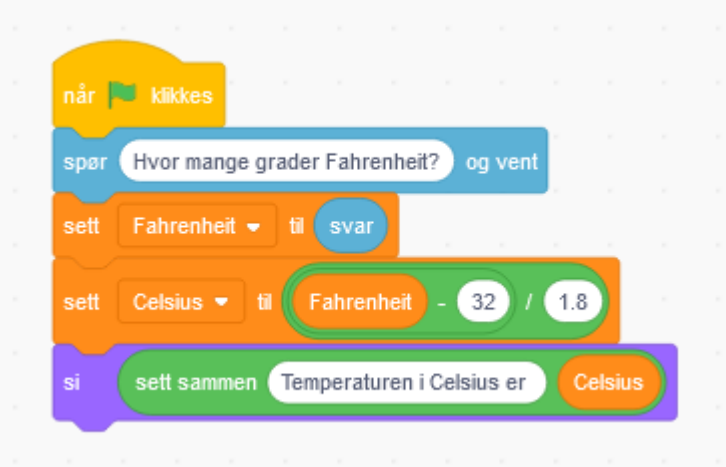
Det første vi må gjøre, er å stille et spørsmål "Hvor mange grader Fahrenheit?" og vente, slik vi gjorde i forrige oppgave. Men istedenfor å bruke svaret direkte i beregningene slik som i sted, så skal vi heller sette variabelen "Fahrenheit" til svaret som blir gitt:

Figur 5-27: Opprette variabel Fahrenheit

Deretter skal vi bruke variabelen Celsius til å gjøre beregningene. Også denne gangen blir det nødvendig å bygge sammen noen komponenter, men ikke like mye som i forrige konvertering.

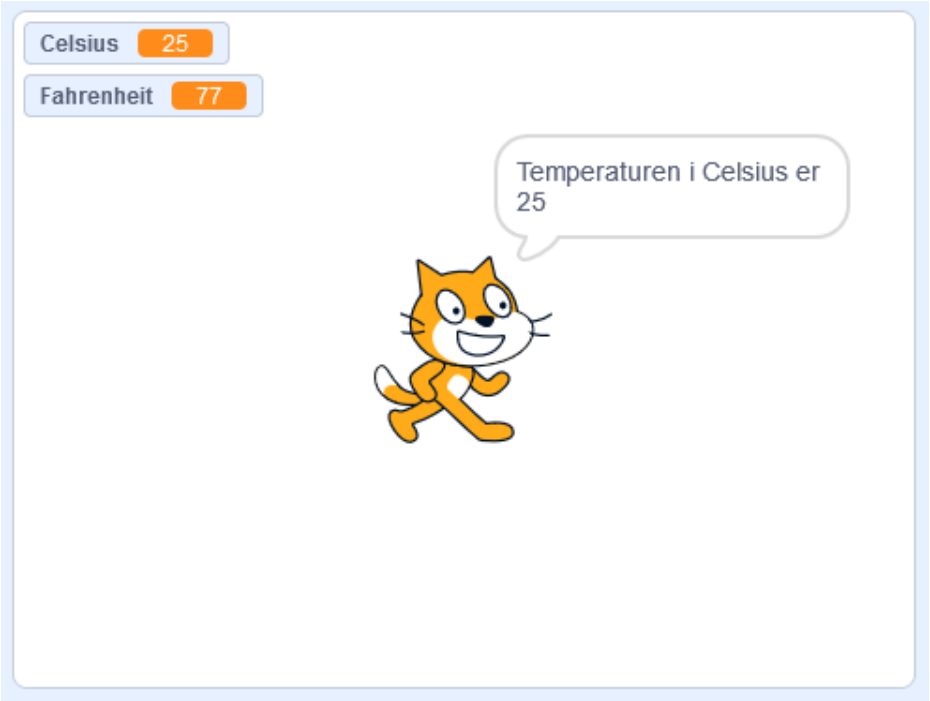
Figur 5-28: Gjøre beregniner i en variabel

Det neste man kan gjøre, er å få katten til å si hva temperaturen er med en "sett sammen"-blokk, skrive inn tekst, og bruke variabelen Celsius. Hele koden blir da slik:



Figur 5-29: Konvertere Fahrenheit til Celsius

Siden vi nå har brukt variabler, og valgt å la disse være visende på skjermen, vil man også kunne se temperaturen i både grader Fahrenheit og grader Celsius, så det er ikke strengt tatt nødvendig med den siste tekstbiten.



Figur 5-30: Skjermdump fra konvertere Fahrenheit til Celsius

5.1.2.5 Scratch eksempel 5 - Bruk av løkker: Å sjekke om PIN-koden er riktig.

Vi kan bruke løkker for å kjøre en kode dersom en hendelse har inntruffet, og en annen kode dersom den ikke er det – såkalte betingelser. På engelsk kaller vi én av disse løkkene for en if/else-løkke, på norsk hvis/ellers-løkke. **Hvis** noe har inntruffet, kjør denne koden som kommer her. **Ellers** kjør denne koden istedenfor.

Et eksempel for å vise dette er når man taster inn PIN-koden på telefonen. **Hvis** koden er riktig, lås opp telefonen. **Ellers** gi beskjed om at PIN-koden er feil:

Vi starter med å spørre "Hva er PIN-koden din?", med en vanlig spørre-blokk fra tidligere.

Figur 5-31: Scratch spørre-blokk

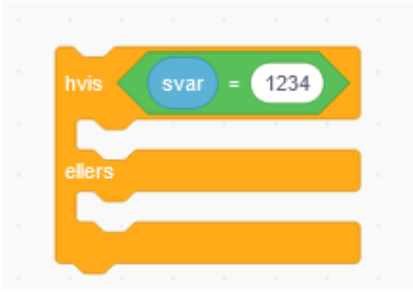
Deretter finner vi frem denne hvis/ellers-blokken fra styring-menyen:

Figur 5-32: Scratch hvis/ellers-blokk

Det vi nå ønsker er å sjekke om det brukeren taster inn faktisk er den riktige PIN-koden, altså om svar = korrekt PIN-kode. I operator-menyen har vi flere sammenlignings-blokker. Vi ønsker den tredje av disse:

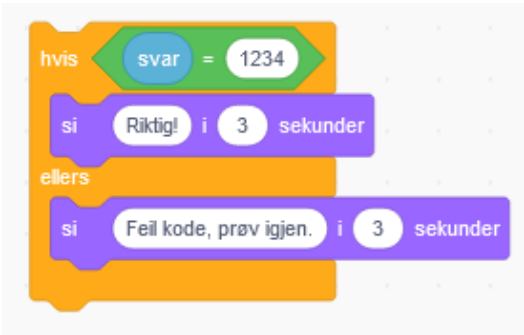
Figur 5-33: Scratch sammenligningsoperatorer

Så setter vi disse sammen, og angir at riktig PIN-kode er 1234:



Figur 5-34: Sammenligningsoperator i en hvis/ellers-blokk

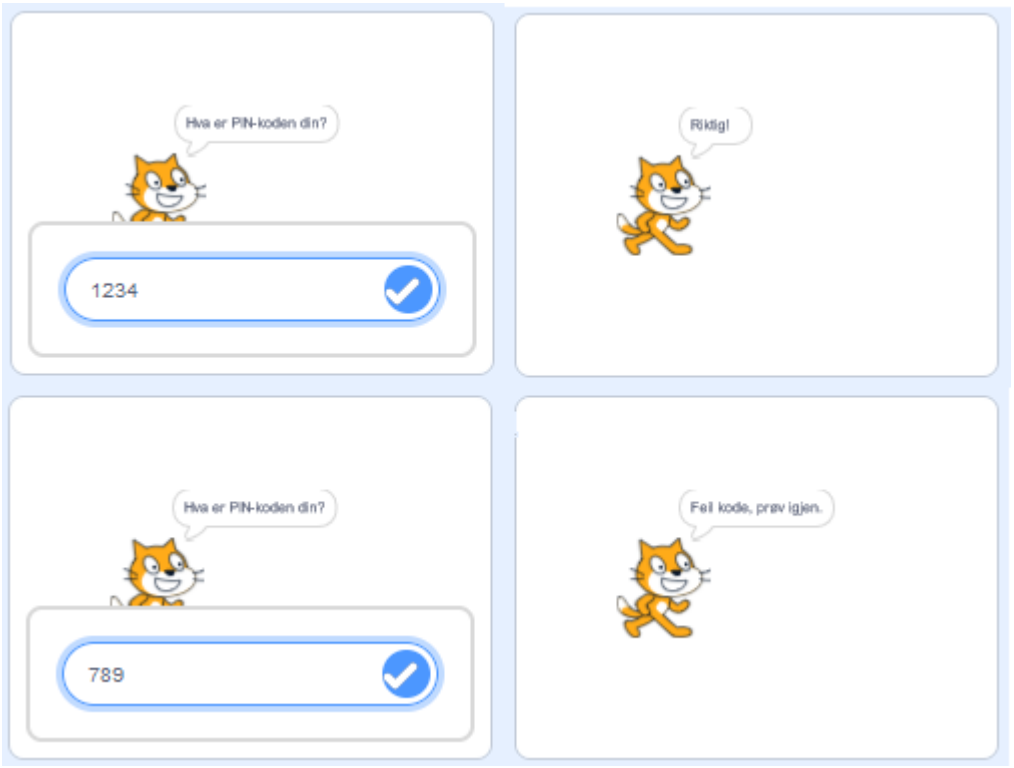
Hvis PIN-koden er korrekt, ønsker vi å gi beskjed til brukeren at den er det. **Ellers** ønsker vi å gi beskjed om at den er feil:



Figur 5-35: Hvis/ellers-blokk med sammenligningsoperator og tekst som skal skrives ut

Til slutt ønsker vi å stoppe all kode, slik at man eventuelt kan prøve med en ny PIN-kode dersom man tok feil. Så setter vi alle disse elementene sammen, og den totale koden ser slik ut:

Figur 5-36: Program som sjekker korrekt PIN-kode



Figur 5-37: Skjermdump fra program som sjekker PIN-kode

Python

Nå som elevene er introdusert for Scratch og å kode med blokker, ønsker vi å vise noen av de samme programmeringsoppgavene i Python. Før vi tar fatt på koden, vil vi gi en innføring i hva Python er, hvilke programmer man kan bruke, og hvordan man kommer i gang med å skrive kode.

5.1.3 Introduksjon Python

5.1.3.1 Nettbasert kompilator: Programiz.com

I denne gjennomgangen har vi valgt å bruke en variant som ligger på en nettside, sånn at terskelen for å komme i gang skal være lav. Vi har valgt å bruke programiz.com sin, men det finnes mange ulike online kompilatorer. Et annet godt alternativ er trinket.io. Begge steder kan man begynne å kode direkte i nettleseren, uten behov for å logge inn. Hovedforskjellen er at med trinket sin kan man importere biblioteker, noe man ikke kan med programiz. Vi skal ikke importere biblioteker i denne gjennomgangen, og har derfor ikke behov for å bruke den nettsiden.

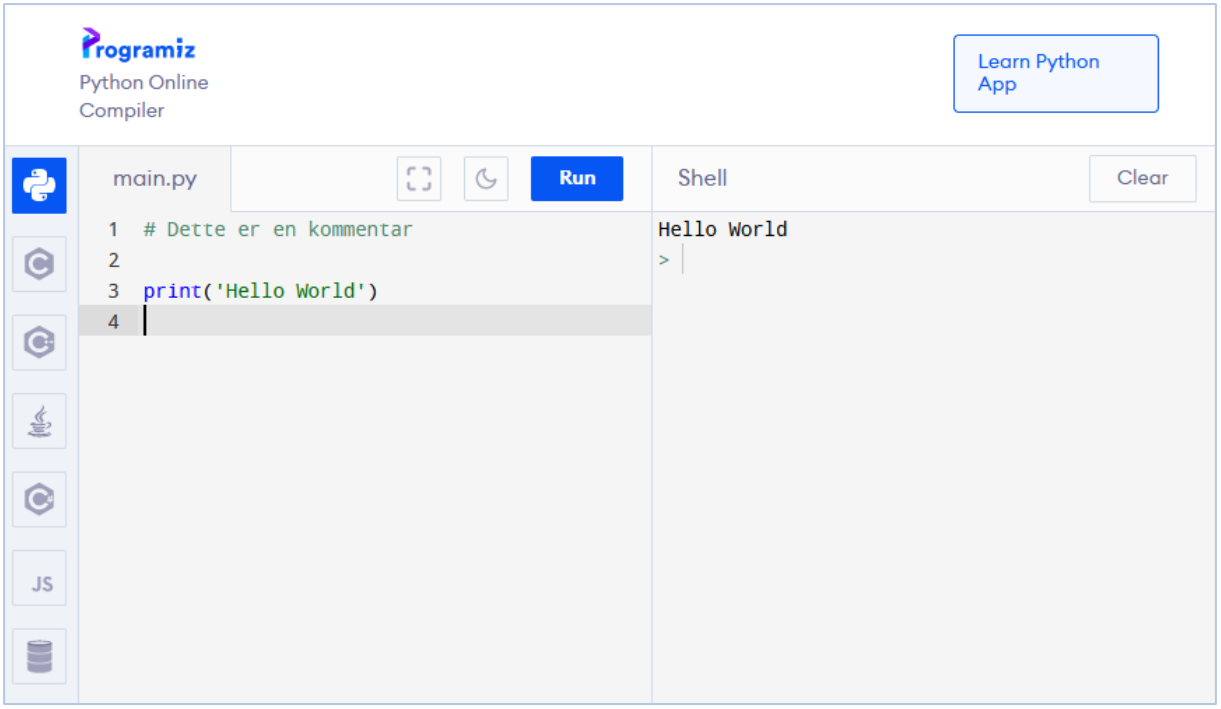
Man kan også velge å laste ned programvare, for eksempel Spyder, Anaconda, Visual Studio Code, Thonny, Jupyter, Atom eller andre. Da vil det være veldig enkelt å laste inn data fra egne filer (for eksempel tekstfiler som Microsoft Word eller Notepad, regnearkfiler som Microsoft Excel eller lignende). Det er ikke noe vi skal gå gjennom i dette innføringsheftet, og det er derfor ikke nødvendig med et eget program.

På både programiz.com og trinket.io kan man velge å opprette en egen gratis bruker, men det er ikke nødvendig for å bruke sidene deres til å kode. Det kan derimot være en fordel dersom man ønsker å lagre arbeidet sitt til senere bruk. Selv om man ikke logger inn, kan man laste ned den koden man har skrevet til sin datamaskin, og ta vare på det på den måten.

5.1.3.2 Tekstfil og konsoll

Når man skriver kode, skriver man i en tekstfil. Programmet (for eksempel Spyder) eller den nettbaserte kompilatoren (for eksempel programiz.com) vil så lese denne tekstfilen, tolke det som står der og utføre de kommandoene den får beskjed om.

Kodeprogrammer er ganske "dumme". De trenger klare instrukser for å tolke det du har skrevet til noe fornuftig. Om noe er skrevet på feil måte, vil ikke programmet forstå hva det skal gjøre, og man vil få en feilkode. I løpet av denne gjennomgangen skal vi vise noen av de vanligste feilene som kan oppstå.



Figur 5-38: Skjermdump fra https://www.programiz.com/python-programming/online-compiler/

Til venstre, i main.py, er tekstfilen – stedet der programmereren skriver inn kode. Til høyre er konsollen. Her vises det hva programmet har tolket basert på koden.

All kode i dette heftet er skrevet på programiz.com, og bildene med koder er fra denne nettsiden.

5.1.3.3 Print()-funksjon og kommentarer

Når man bruker nummertegn (#) i Python, så forteller man programmet at "dette skal du ikke bry deg om". Ved å bruke nummertegn, kan man skrive kommentarer til koden som kan leses av mennesker, men som programmet hopper over.

I bildet ovenfor er det skrevet et lite program på én linje (linje 3).

For å starte programmet, trykker man på den blå knappen hvor det står "Run", en forkortelse for "Run code" eller "Kjør koden". Denne koden skriver ut teksten "Hello World", som dere kan se i konsollen til høyre. Dette kalles å skrive til skjermen, eller output.

5.1.4 Datatyper

I tekstfilen main.py har man skrevet koden print('Hello World'). Apostrofene rundt teksten 'Hello World' forteller Python at "her kommer det en tekststreng". En tekststreng er en av Pythons innebygde datatyper. Det finnes mange datatyper, og her går vi gjennom de vi trenger i dette heftet:

Tabell 5-1: Datatyper i Python

Forkortelse	Datatype	Eksempel
str	Tekststreng (string)	'Hello World' "Hello World"
int	Heltall (integer)	25
float	Desimaltall (decimal number), flyttall (floating number)	3.14
bool	Boolsk verdi	True, False

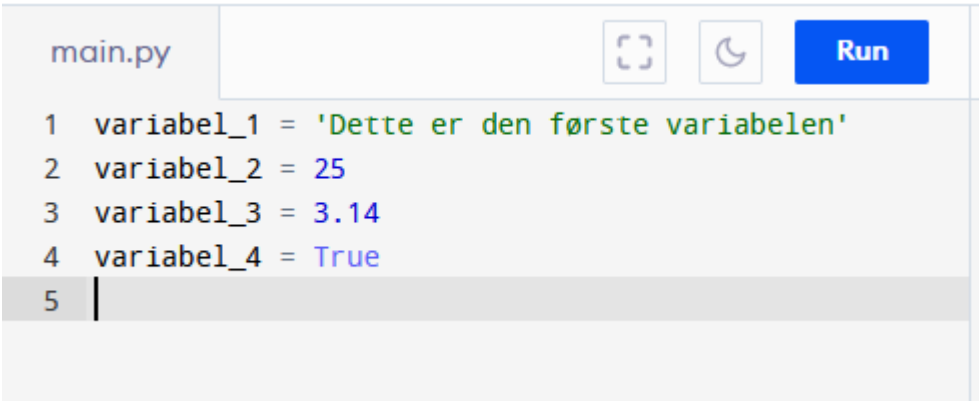
Legg merke til at man kan bruke enten enkle eller doble anførselstegn i en string, men man må velge én type anførselstegn rundt teksten i en streng. Merk også at desimaltall må skrives med punktum, ikke komma som er vanlig på norsk.

Andre datatyper er blant annet komplekse tall, lister, tupler og set.

5.1.4.1 Variabler

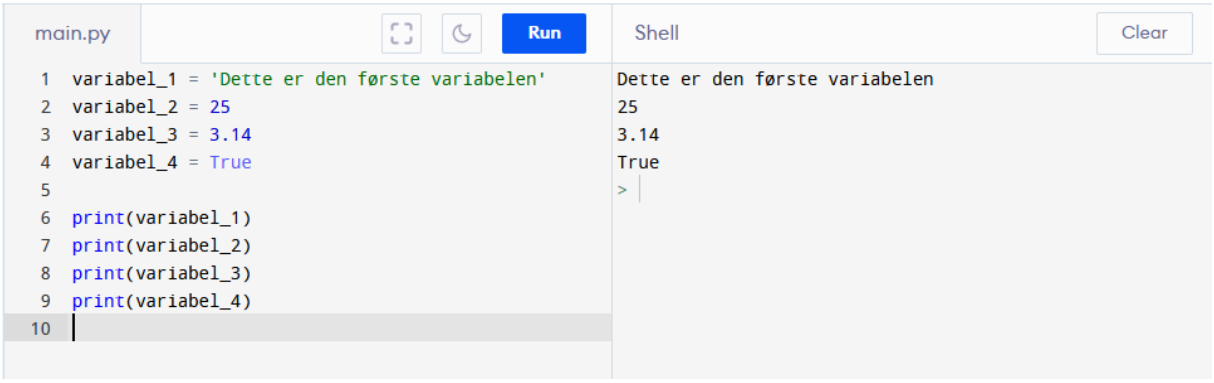
Variabler er noe vi får mye bruk for når vi skal skrive kode i Python. En variabel er et sted hvor man kan lagre informasjon, et alias man kan bruke senere i koden. Man kan også oppdatere og endre variabler underveis.

Når man bestemmer hva en variabel skal inneholde, sier man at man *tilordner* variabelen en verdi. Man tilordner variabler ved hjelp av et enkelt likhetstegn. En variabel kan inneholde hvilken som helst datatype:



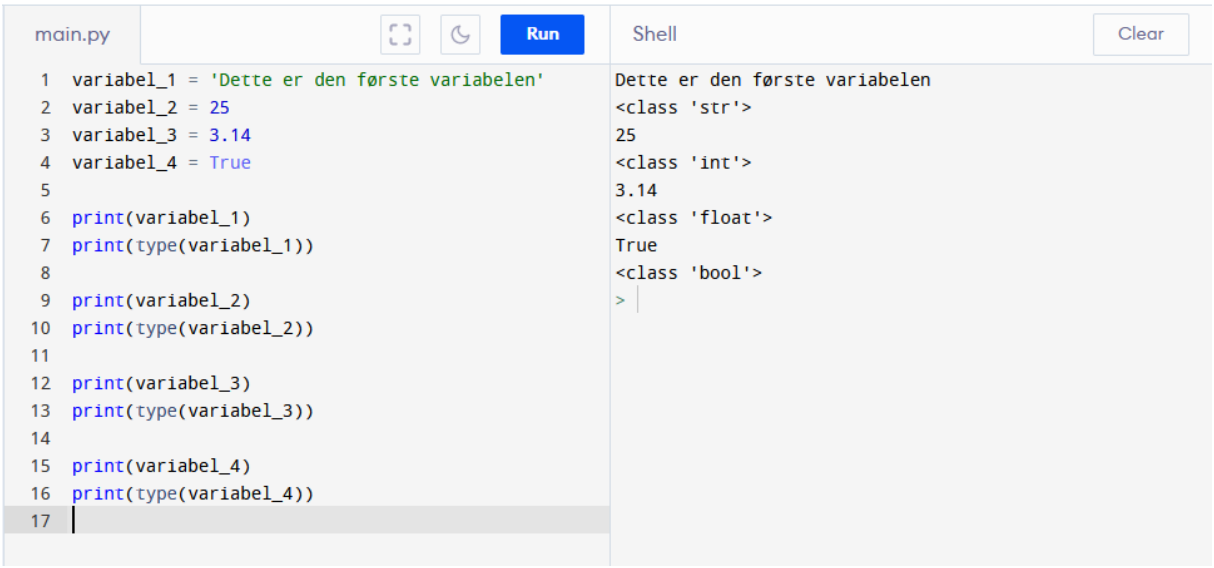
Figur 5-39: Variabler med ulike datatyper

For å hente ut informasjonen fra variabelen, kan man for eksempel bruke en **print()**-setning. I vårt tilfelle vil da informasjonen fra variabelen bli skrevet til konsollen:



Figur 5-40: Utskrift av variabler med print()

Hvis man vil vite hvilken datatype en variabel inneholder, kan man bruke kommandoen **type()**:



Figur 5-41: Utksrift av variabler med print() og print(type())

Da kan man se at variabel 1 hører til klassen **str** (tekststreng), variabel 2 er en **int** (heltall) og så videre. Legg også merke til at Python ikke bryr seg om linjeskiftene mellom de ulike **print**-setningene. En tom linje har ikke noe kode, og uten klare instrukser gjør ikke Python noen ting.

5.1.4.2 Variabelnavn

Det finnes noen regler for hva en variabel kan kalles. De er som følger:

- Variabelnavnet kan bare inneholde bokstaver (A-z), tall (0-9) og understreker (_)

- Variabelnavnet kan kun starte med bokstaver eller understreker, ikke tall
- Variabelnavnet skiller mellom store og små bokstaver

Eksempler på variabelnavn som er gyldige:

- x
- var1
- _varx
- Dette_er_en_variabel_for_farten_til_en_bil

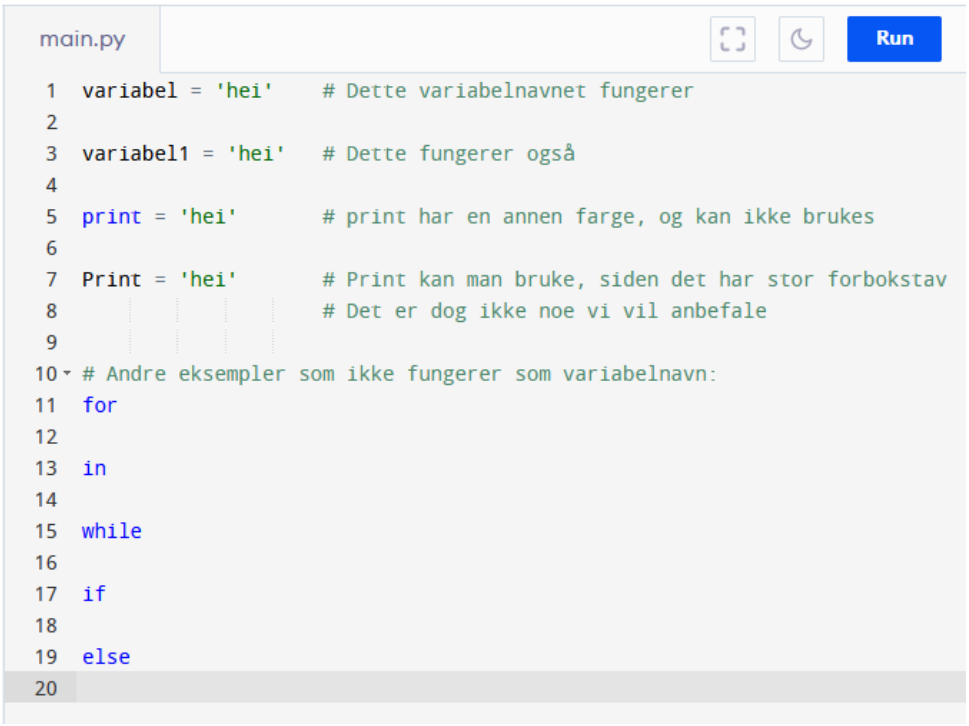
Vi anbefaler dog å holde variabelnavnene korte og forklarende, for eksempel "fart" eller "bil_fart" i siste eksempel.

Eksempler på variabelnavn som ikke er gyldige:

• 1_variabel	Man kan ikke starte en variabel med et tall
• Variabel-1	Man kan ikke bruke minustegn
• fart(bil)	Man kan ikke bruke parenteser
• 25	Man kan ikke starte med et tall
• var*	Man kan ikke bruke stjernetegn
• høst	Man kan ikke bruke andre bokstaver enn A til z

Og som nevnt: variabelnavn skiller mellom store bokstaver. Var, var, vAr, VaR og VAR er fem forskjellige variabler, som kan inneholde hver sine verdier.

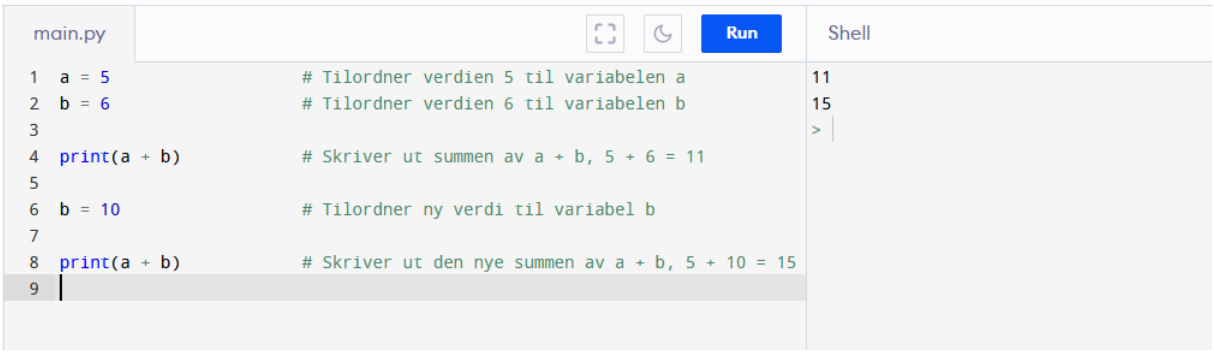
I tillegg er det noen ord man ikke kan bruke, som allerede er innebygde funksjoner eller spesielle ord i Python. Vi har allerede nevnt **print**, men det finnes en rekke andre. Om man skriver en av disse funksjonene inn i tekstfilen, vil det endre farge, som indikerer at dette er en innebygget funksjon. Vi skal ikke gå gjennom alle disse her, men et lite utvalg som eksempler:



Figur 5-42: Eksempler på variabelnavn som fungerer og ikke fungerer

5.1.4.3 Tilordne ny verdi til en variabel

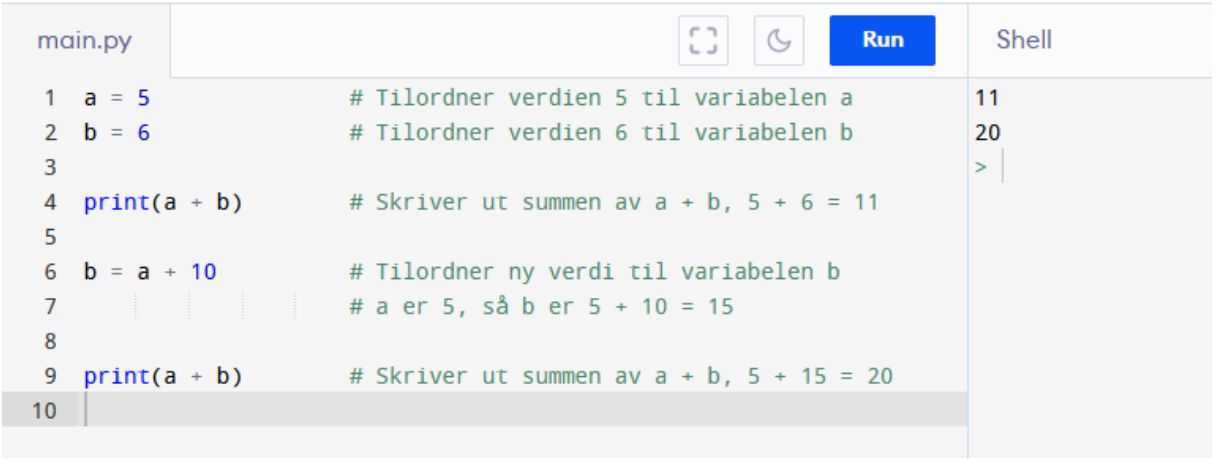
Når man har tilordnet verdi til en variabel, kan man senere tilordne denne en ny verdi:



Figur 5-43: Tilordne ny verdi til en variabel med tall

Variabelen **a** har verdien 5 hele tiden, mens variabelen **b** har verdien 6 før den første **print()**-setningen. 5 + 6 = 11, som vi kan se i konsollen til venstre. Så tilordner vi variabelen **b** en ny verdi, b = 10. Samme **print()**-setning i linje 4 og linje 8 skriver ut to ulike resultater.

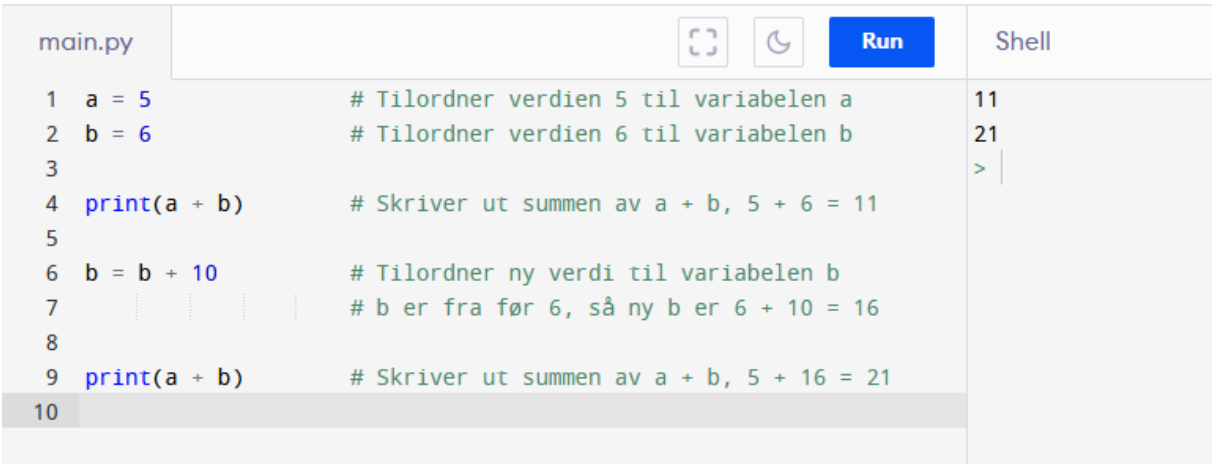
Man kan også tilordne ny verdi ved å bruke en annen variabel:



Figur 5-44: Tilordne ny verdi til en variabel med en annen variabel

Verdien for **a** er også her uendret gjennom hele koden – det er variabelen **b** vi tilordner ny verdi i linje 6.

Man kan til og med tilordne en ny verdi til en variabel ved hjelp av den gamle verdien:



Figur 5-45: Tilordne ny verdi til en variabel med samme variabel

I linje 2 tilordner vi verdien 6 til variabel **b**. I linje 6 sier vi at variabelen **b** skal oppdateres, den skal nå være lik den gamle verdien til variabelen **b** pluss 10.

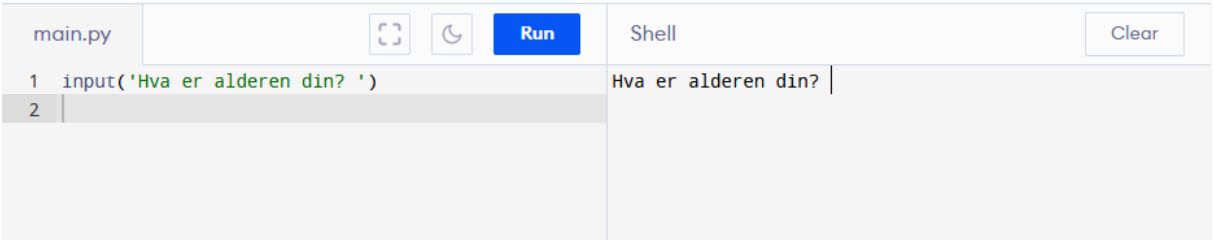
Derfor er det viktig at man gir variabler navn som er beskrivende, og som minsker sannsynligheten for at man bruker samme variabelnavnet om igjen, uten å være klar over det.

5.1.5 Input

Input() er en innebygget funksjon i Python. Den lar brukeren av programmet komme med input til koden. Det vil si at et program kan oppføre seg forskjellig for forskjellige brukere, avhengig av hva som blir tastet inn.

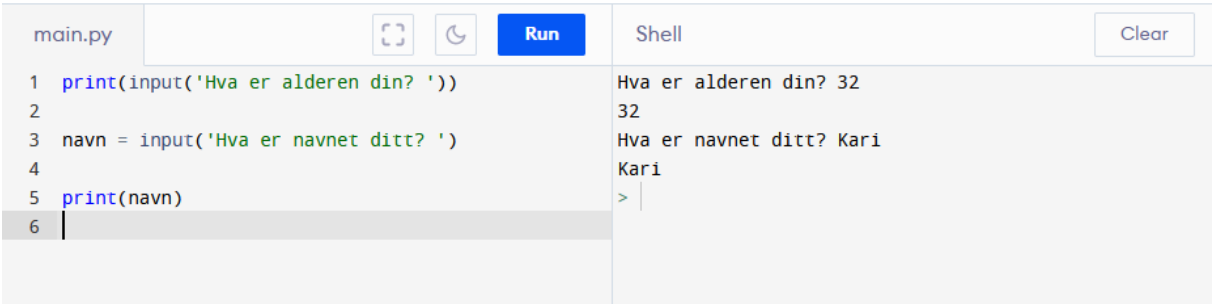
For å bruke input-funksjonen, skriver man input(), og så en tekst mellom parentesene som man vil at skal vises på skjermen. Om man for eksempel ønsker å spørre om noens alder, skriver man input('Hva er alderen din? '). Systemet vil deretter vente på at

brukeren skal skrive inn noe, før den kjører resten av koden. I vårt tilfelle skriver vi direkte i konsollen:



Figur 5-46: Python input()

Koden slik den er skrevet nå, gjør ikke noe annet enn å spørre hva alderen er. Den bruker ikke den informasjonen til noe, og tar ikke vare på den til senere bruk. Man kan enten skrive ut alderen direkte, eller så kan man tilordne input()-setningen til en variabel, og bruke variabelnavnet for å kalle på input'et senere:



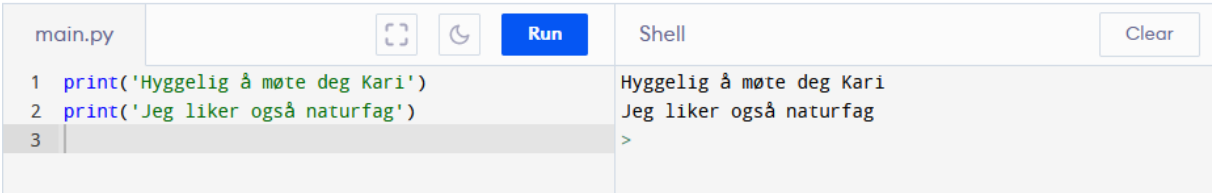
Figur 5-47: Python print(input())

Det som er verdt å merke seg, er at det som blir skrevet inn i input alltid blir tolket som en tekst-streng. Så selv om 32 kan se ut som det er tall av typen int (heltall), så er det faktisk en string (tekststreng).

Vi skal nå se på det vi har gått gjennom så langt i et eksempel. Til det bruker vi det samme som første eksempel i Scratch, en enkel samtale mellom brukeren og programmet.

5.1.5.1 Python eksempel 1 - En enkel innføring: Utskrift til konsollen ved hjelp av variabler og input

Med Python kan man som nevnt bruke **print()** for å skrive direkte til konsollen. Vi kan derfor skrive koden fra første eksempel i Scratch på følgende måte:

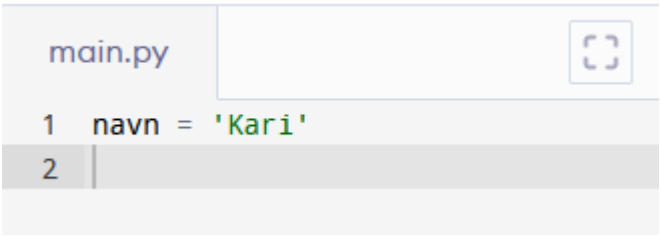


Figur 5-48: Utskrift til konsollen

Vi skal nå vise hvordan man kan gjøre dette ved hjelp av variabler.

I motsetning til koden i Scratch, hvor vi spurte hva brukeren heter og så skrev man det inn, skal vi her kode et navn direkte. Etterpå skal vi vise hvordan man også kan "kommunisere" med Python ved å la brukeren selv skrive inn navn og favorittfag ved hjelp av **input()**.

Vi starter programmet med å skrive vår første variabel på denne måten:



Figur 5-49: Opprette en variabel

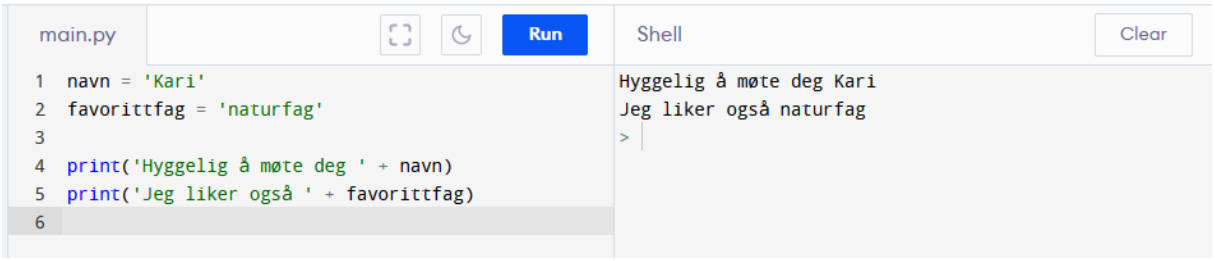
Så oppretter vi vår neste variabel. Det må vi gjøre på neste linje, eller så tror Python at det bare er en fortsettelse av variabelens navn.

Figur 5-50: Opprette to variabler

Så skal vi få programmet til å skrive det samme på skjermen som i sted:

```
Hyggelig å møte deg Kari
Jeg liker også naturfag
```

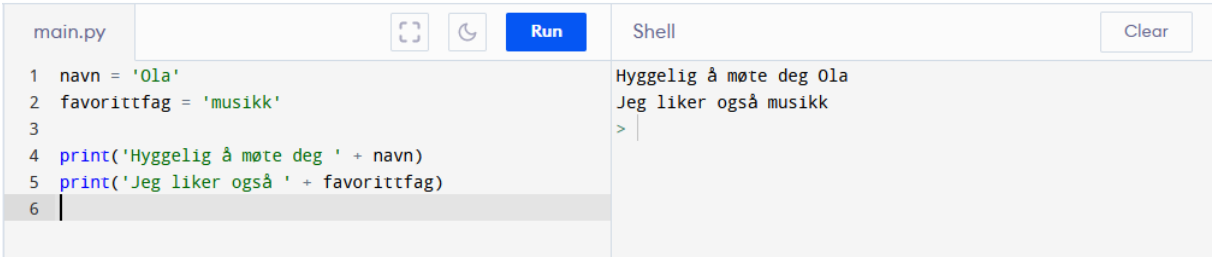
Da må vi få Python til å skjønne at den skal hente variabelen når vi skal skrive ut teksten. Det gjør vi ved å bruke operatoren + mellom tekststrengen og variabelnavnet direkte i **print()**-funksjonen:



Figur 5-51: Skrive ut variabler sammen med tekst

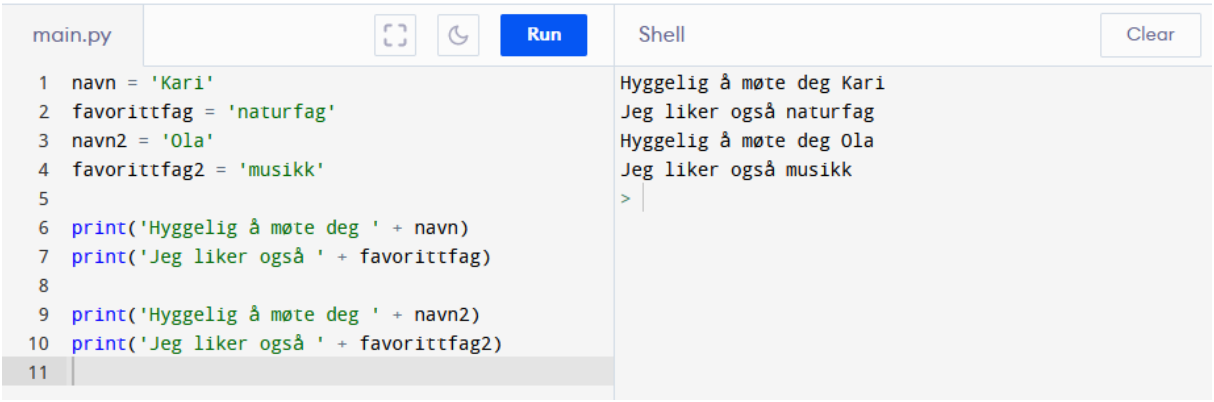
Legg merke til at til sist i tekststrengene 'Hyggelig å møte deg ' og 'Jeg liker også ' så er det et mellomrom. Dette er fordi vi legger sammen to tekststrenger. 'Hei' + 'Kari' tolker Python til å bety 'HeiKari'. Derfor må vi legge til et mellomrom: 'Hei ' + 'Kari' blir til 'Hei Kari'.

Dersom man vil bytte til en annen person, kan man endre i variabelen for å få et annet navn og fag skrevet ut til konsollen:



Figur 5-52: Endre innhold i variabel med hardkoding

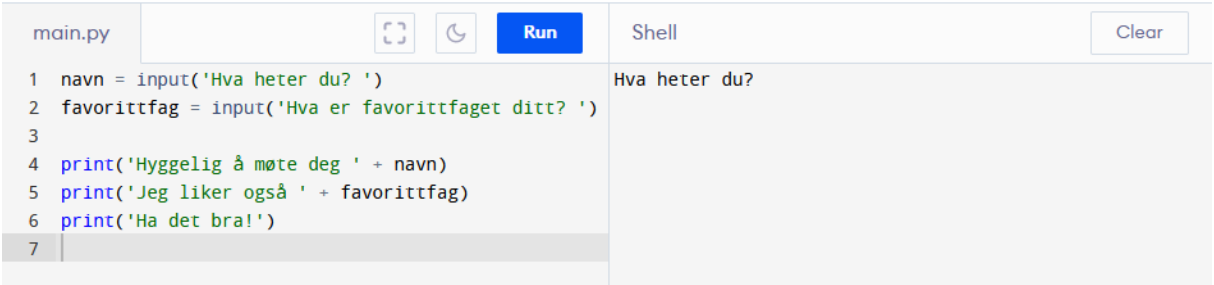
Eller man kan legge inn flere variabler og flere print-setninger:



Figur 5-53: Sette inn flere variabler for å skrive ut flere setninger

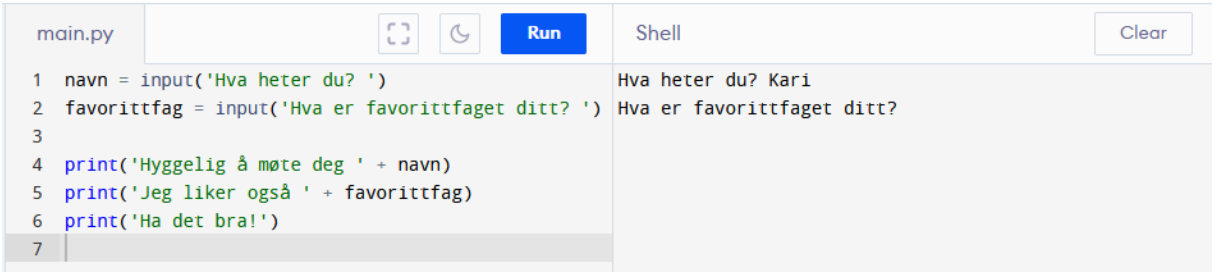
Dess flere personer man ønsker å legge til, dess lengre blir koden. Det kan derfor være lurere å simulere en samtale-setting, slik vi gjorde i Scratch, med at konsollen spør hva navnet og favorittfaget til brukeren er.

Input-funksjonen ber brukeren om å taste inn noe, som man så kan bruke til det man vil senere i koden. Input-funksjonen skjønner selv at den må vente på svar, så man trenger ikke å legge til et vente-signal, slik man måtte i Scratch:



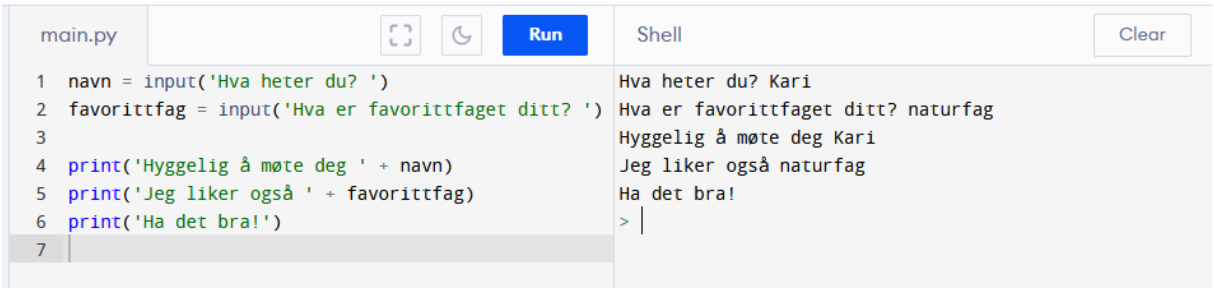
Figur 5-54: Bruke input() i en variabel

Som dere ser, starter man med å skrive variabelnavnet = input(). Så skriver man den teksten som man vil at skal vises til brukeren i anførselstegn, inne i parentesene. Systemet venter nå på at brukeren skal skrive inn noe.



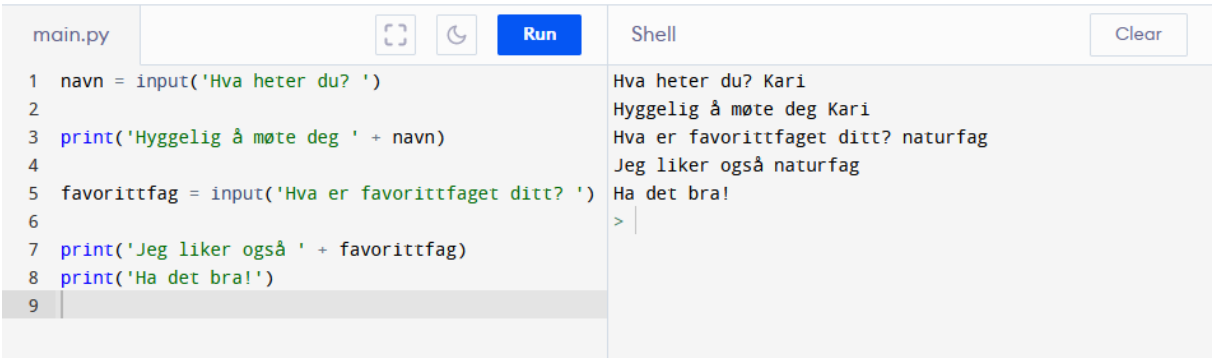
Figur 5-55: Å bruke input() gjør at prorgammet venter på svar

Når koden er satt opp på denne måten, så går programmet direkte til neste spørsmål. Etter å ha tastet inn favorittfaget og trykket <Enter>, blir teksten som kommer ut seende slik ut:



Figur 5-56: Utskrift til konsollen ved hjelp av variabler og input

For å få det til å se mer ut som en dialog, kan man bytte plass på variabelen favorittfag og print-setningen som sier Hyggelig å møte deg:



Figur 5-57: Utskrift til konsollen ved hjelp av variabler og input med logisk rekkefølge

Da ser det mer ut som en dialog, akkurat på samme måte som det første eksempelet i Scratch:

Figur 5-58: Samme kode i Scratch, som Python i figur 5-57

5.1.6 Matematiske operatorer

Vi har sett at plusstegnet kan brukes både til å legge sammen tall som ved vanlig addisjon, og for å sette sammen et tekststreng og en variabel i en print()-setning. Vi skal nå også se nærmere på de andre ulike matematiske operatorene.

Tabell 5-2: Matematiske operatorer

Operator	Navn	Eksempel	Output
+	Addisjon	10 + 4	14
-	Subtraksjon	10 – 4	6
*	Multiplikasjon	10 * 4	40
/	Divisjon	10 / 4	2.5
//	Heltallsdivisjon	10 // 3	3
%	Modulus	10 % 3	1
****	Eksponent	10 ** 3	1000

Modulus er en regneoperasjon som viser resten av en divisjon. $\frac{10}{3} = 3 + \frac{1}{3}$. 10 % 3 er derfor 1.

Heltallsdivisjon viser hvor mange heltall som er i regnestykket. $\frac{10}{3} = 3 + \frac{1}{3}$. 10 // 3 er derfor 3.

5.1.7 Tilordningsoperatorer

Vi har sett at Python kan bruke likhetstegn for å tilordne operatorer, men det finnes også en rekke andre. En rask kikk avslører at det dreier som om de matematiske operatorene sammen med et likhetstegn. Husk dog at her er det snakk om å tilordne variabler, altså at uttrykket på venstre side skal inneholde verdiene på høyre side.

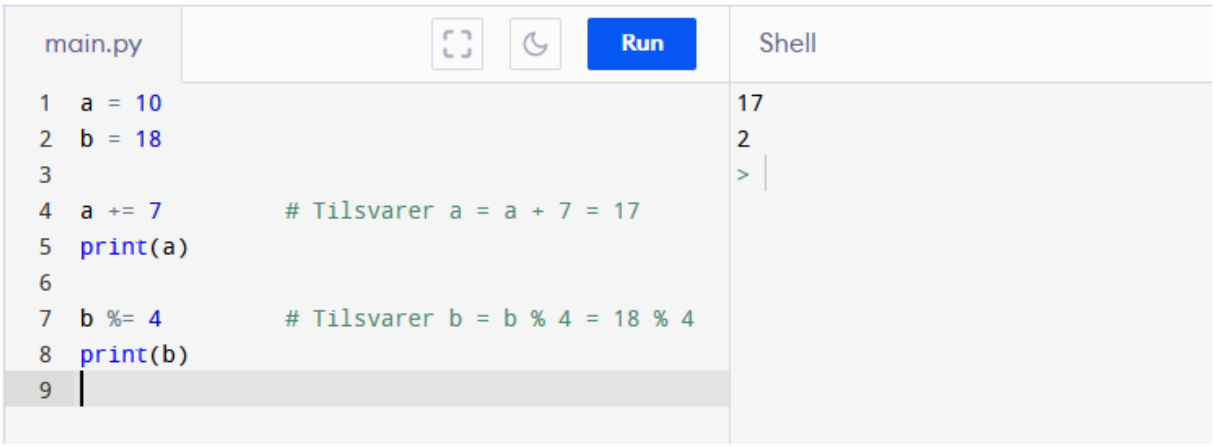
La oss i dette eksempelet si at a = 10, b = 4 og c = 3

Tabell 5-3: Tilordningsoperatorer

Operator	Beskrivelse	Eksempel	Tilsvarer	Output variabel a
=	Tilordning	a = a + b	a = a + b	14
+=	Addisjon og tilordning	a += b	a = a + b	14
-=	Subtraksjon og tilordning	a -= b	a = a – b	6
*=	Multiplikasjon og tilordning	a *= b	a = a * b	40
/=	Divisjon og tilordning	a /= b	a = a / b	2.5
//=	Heltallsdivisjon og tilordning	a //= c	a = a // c	3
%=	Modulus og tilordning	a %= c	a = a % c	1
****=**	Eksponent og tilordning	a **= c	a = a ** c	1000

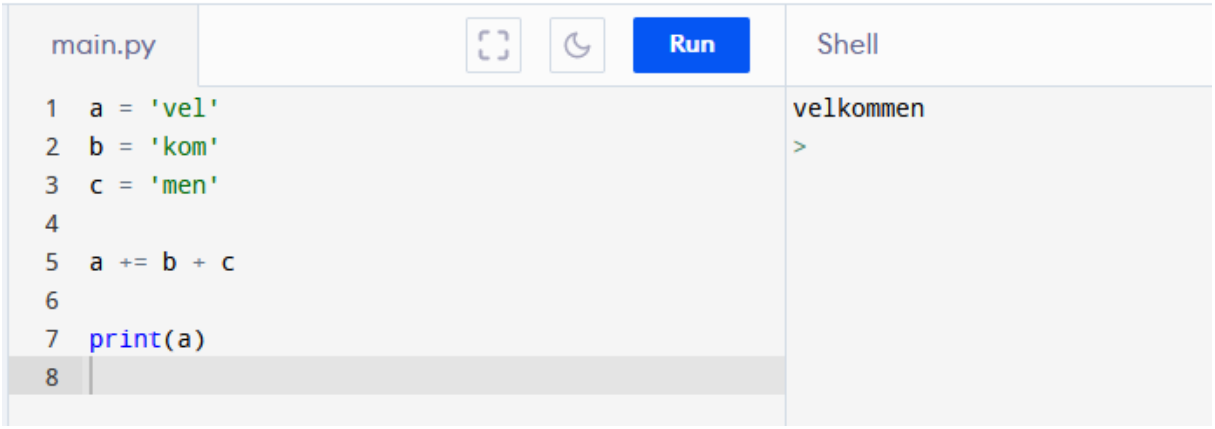
I alle tilfeller er startverdien til variabel a lik 10, mens sluttverdien er som i kolonnen til høyre. Verdien til a er endret fordi vi har tilordnet den en ny verdi med ulike tilordningsoperatorer.

Man trenger ikke kun skrive variabler på høyre side av tilordningsoperatoren. Man kan også tilordne ved hjelp av tall:



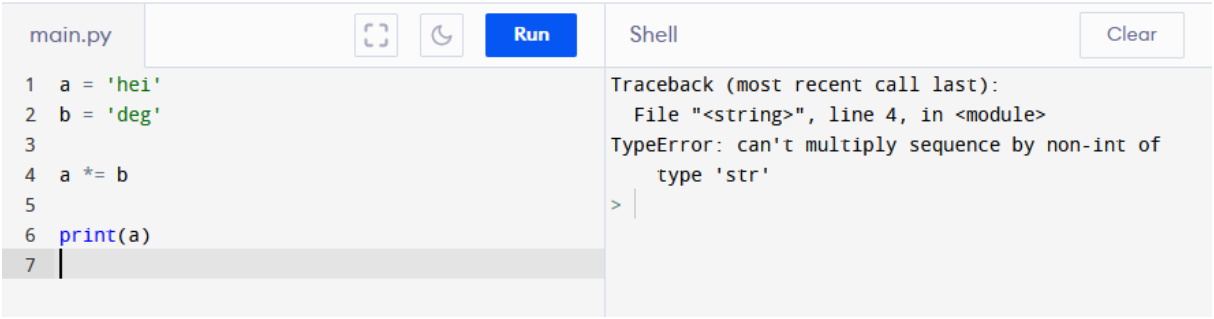
Figur 5-59: Eksempel på bruk av tilordningsoperatorer med tall

Med plusstegnet kan man også tilordne variabler av typen tekststreng (string) med operatoren +=



Figur 5-60: Eksempel på bruk av tilordningsoperatorer med tekst

Dette fordi man kan sette sammen tekststrenger ved hjelp av plusstegnet, som vi går grundigere gjennom i neste delkapittel. Det fungerer dog ikke med noen av de andre tilordningsoperatorene, i og med at de oppfører seg som matematiske operatorer. Det gir ikke mening å multiplisere 'hei' med 'deg':

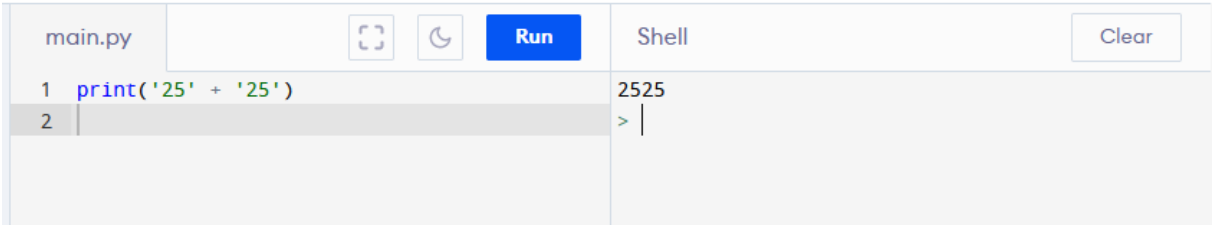


Figur 5-61: Eksempel på feil bruk av tilordningsoperatorer med tekst

Som dere ser, sier feilkoden at man ikke kan multiplisere sekvenser som er ikke-heltall av typen tekststreng.

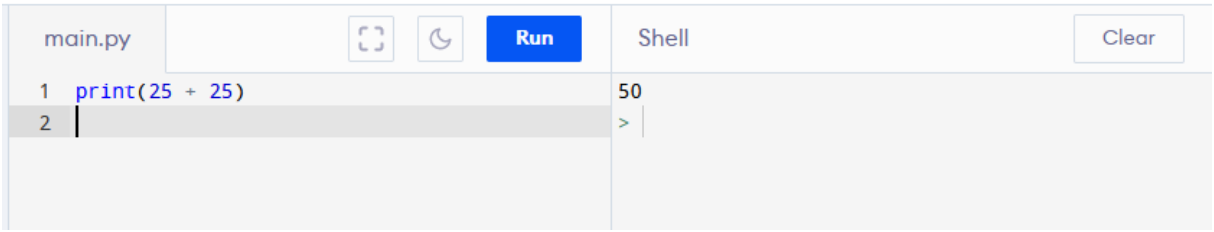
5.1.8 Konkatenering og å sette sammen variabler av ulike typer

Vi har allerede sett at likhetstegn brukes for å tilordne variabler, og at plusstegnet kan brukes til å sette sammen tekst og variabelnavn. Vi skal nå se på hvordan Python skjønner at vi skriver inn tall. I motsetning til når vi skriver tekst, som må skrives mellom anførselstegn, så kan man skrive tall direkte. Dersom man skriver tall mellom anførselstegn, for eksempel '25', så tolker Python dette som tekst. Hvis man da forsøker å legge sammen '25' + '25', så skriver Python 2525:



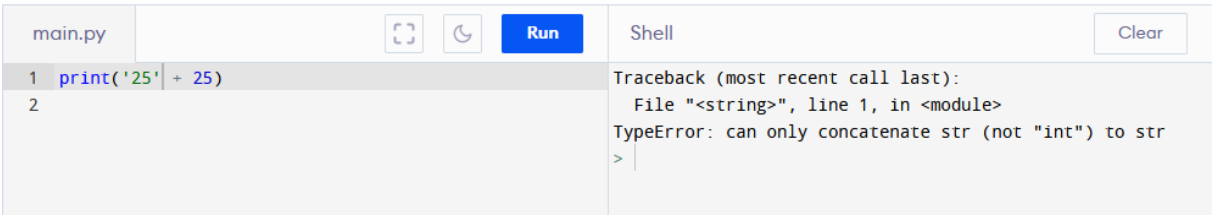
Figur 5-62: Skrive ut addisjon av tall, men som er av typen tekststreng

Om man fjerner anførselstegnene, så skjønner Python at her er det tall, og adderer på vanlig måte:



Figur 5-63: Skrive ut addisjon av tall av typen integer (heltall)

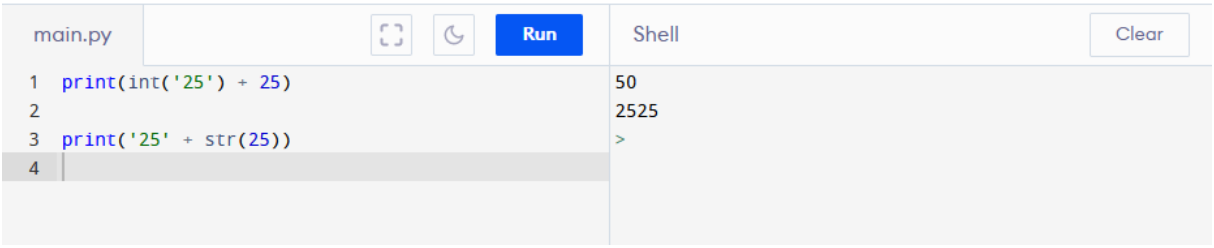
Det er også viktig å merke seg at dersom man prøver å legge sammen en tekststreng og et tall, så skjønner ikke Python hva man prøver på, og man får en feilkode:



Figur 5-64: Skrive ut tall og tekst i samme print()-setning

I feilkoden kan man se at feilen har oppstått i linje 1, og at feilen er at Python ikke kan sette sammen/konkatenere (concatenate) en tekststreng (str) og et heltall (int).

Det man derimot kan gjøre, er å endre teksten 25 til et heltall ved hjelp av å sette int() rundt, eller endre tallet 25 til en streng ved å sette str() rundt tallet:

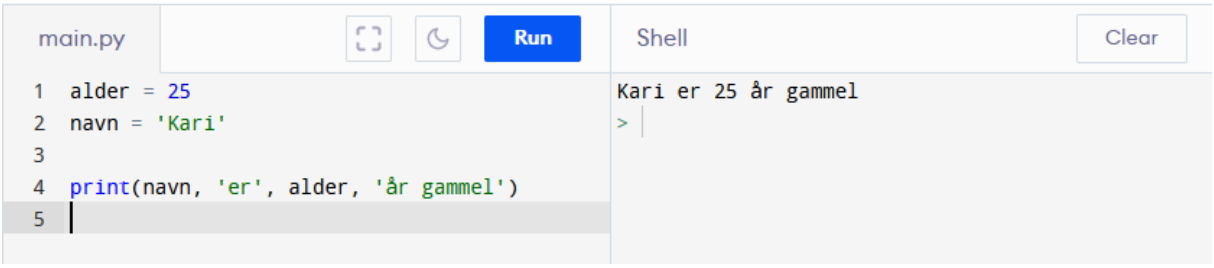


Figur 5-65: Endre datatype i print()-setningen

Dette ser kanskje ikke så logisk ut med et såpass enkelt eksempel, men er veldig nyttig å kunne dersom man benytter variabler av ulike typer i for eksempel en print()-setning:

Figur 5-66: Skrive ut variabler og tekststrenger i samme print()-setning med plusstegn

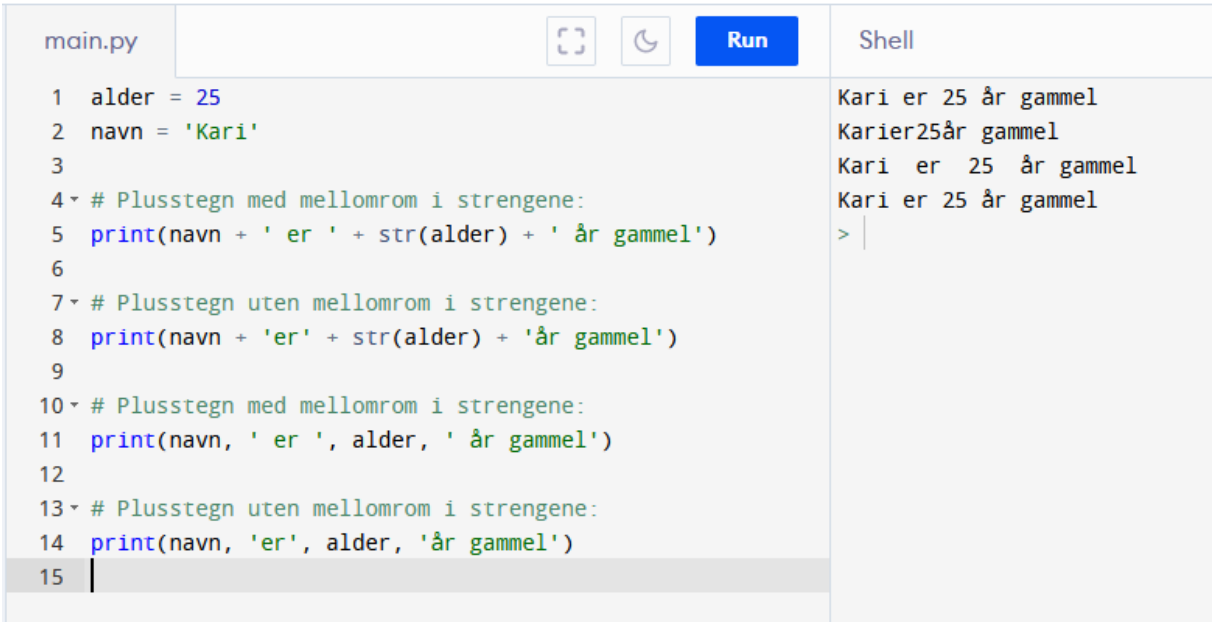
En annen måte å sette sammen ulike variabler i en utskrift, er å bruke komma istedenfor plusstegn:



Figur 5-67: Skrive ut variabler og tekststrenger i samme print()-setning med komma

Da forteller Python at "her skal du sette disse elementene etter hverandre i en print()-setning". Da kan man bruke hvilke datatyper man ønsker, her er det både strenger (navn, 'er', 'år gammel') og heltall (alder).

Grunnen til at vi ønsker å vise begge metodene, er at de kan brukes på litt forskjellige måter. Når man bruker plusstegn, legges alle elementene sammen. Det vil si at dersom man ikke har mellomrom i de skrevne tekst-strengene, så blir utskriften en sammenhengende tekst. Når man istedenfor bruker komma, settes det automatisk et mellomrom mellom de ulike elementene:

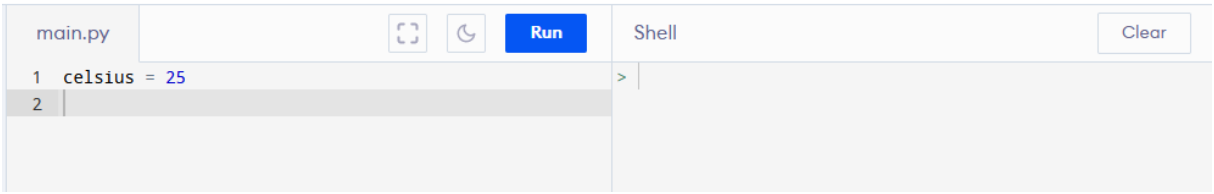


Figur 5-68: Visualisering av forskjellen på plusstegn og komma

Når man skal bruke plusstegn og når man skal bruke komma, avhenger litt av hva man ønsker å skrive ut. I de fleste tilfeller er det tilstrekkelig å bruke komma. Da slipper man også å endre typen til variablene slik man må med plusstegn, som at man rundt alder må skrive str() for å endre variabeltypen fra heltall til tekststreng.

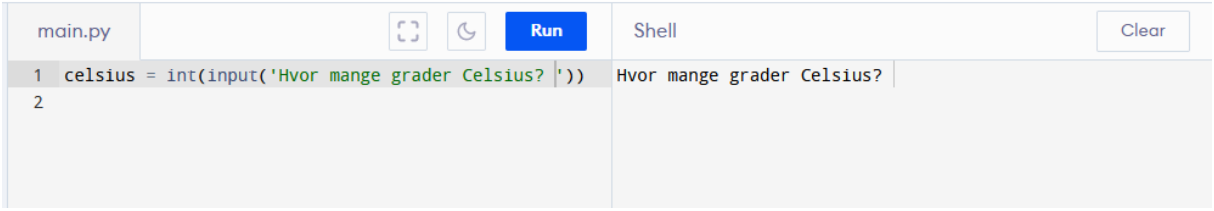
5.1.8.1 Python eksempel 2 – Å legge sammen to tall: Konvertere fra Celsius til Kelvin

På samme måte som i forrige oppgave, så kan man skrive en variabel som inneholder celsiusverdien direkte:



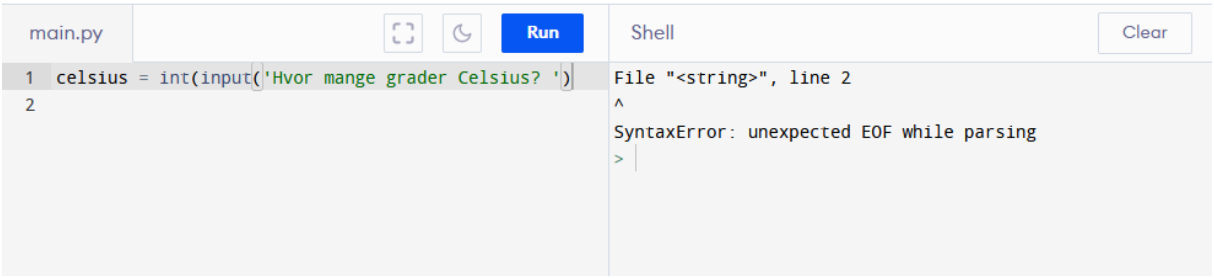
Figur 5-69: Opprette variabel med tall

Istedenfor å gjøre det slik, så ber vi heller brukeren om å taste inn et tall. Vi vil at dette tallet skal være et heltall. Når vi nå skriver en input-setning, så skriver vi int() rundt hele input-setningen. Som vi nevnte tidligere, så blir et input alltid tolket som en tekststreng. Ved å skrive int() rundt input-setningen, gjør vi om tekststrengen til heltall.



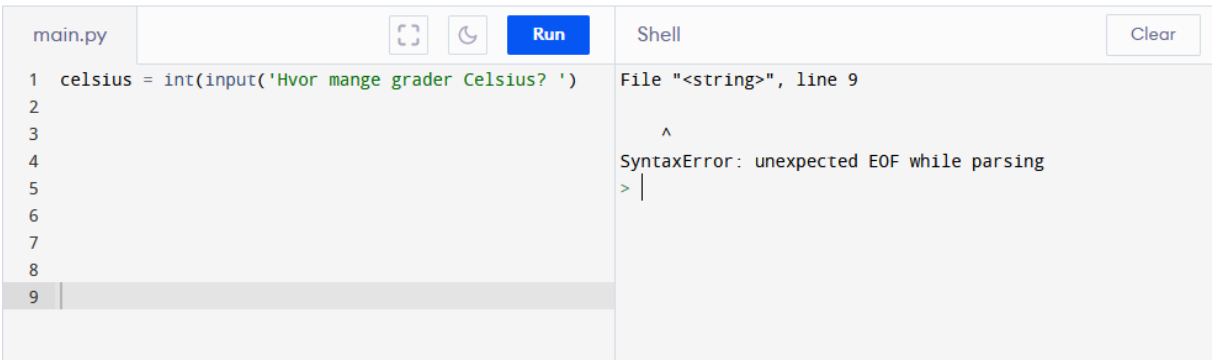
Figur 5-70: Opprette en tall-variabel med input()

Her er det viktig å sørge for at parentesene er på riktig sted. Om man for eksempel glemmer en parentes på slutten, så skjønner ikke Python hvor den delen av koden slutter, og man får en feilkode:



Figur 5-71: Feil bruk av parenteser

Her står det at feilen er på linje 2, selv om feilen egentlig er på linje 1. EOF betyr "End of file". Feilkoden sier derfor at det er en uventet linjeslutt i linje 2. Det er som nevnt fordi Python ikke skjønner hvor koden slutter. Om man legger til flere linjer, vil Python si at det er der feilen er:

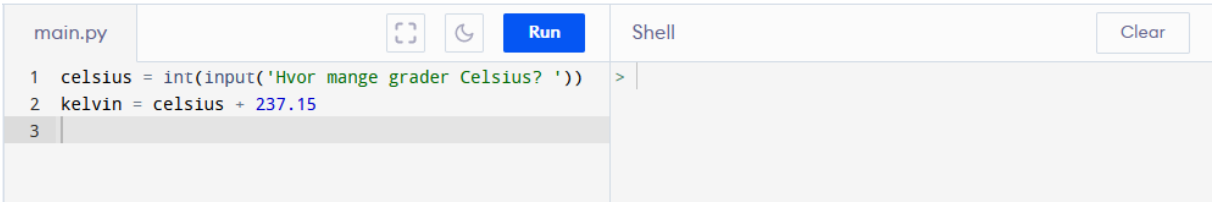


Figur 5-72: Feil bruk av parenteser og forklaring av feilkode

Å oppdage slike feil kan derfor være litt frustrerende, fordi det ikke er noe hjelp i linjeangivelsen til feilkoden. Et tips er derfor å ikke stole blindt på at linjeangivelsen er riktig, selv om den i mange tilfeller er det.

La oss gå tilbake til den koden hvor parentesene er korrekt plassert. Det neste vi nå skal gjøre, er å regne ut hvor mye dette blir i Kelvin. Vi kan gjøre det direkte i print-setningen, slik som vi gjorde i Scratch (figur 5-17):

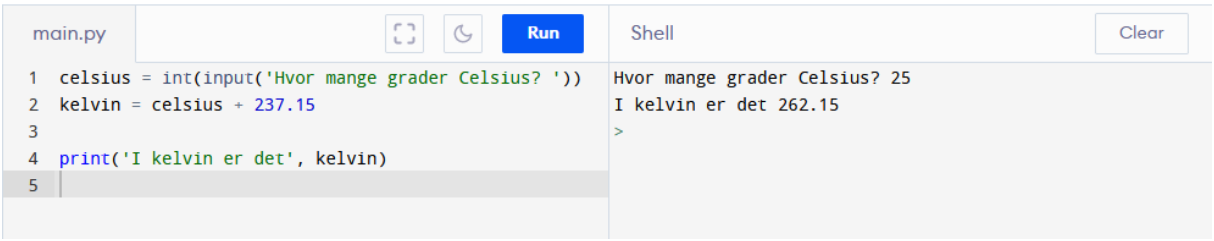
Her er det dog mer fornuftig å opprette en ny variabel som heter kelvin, og addere celsius med 237.15:



Figur 5-73: Opprette en variabel ved å addere en annen variabel med et tall

Her kan vi skrive celsius direkte, fordi vi har angitt at celsius skal være et tall (int). Legg merke til at vi ikke har anførselstegn rundt celsius på linje 2, for det ville ført til at Python tolker dette som tekst. Celsius er i dette tilfellet ikke tekst, men en variabel.

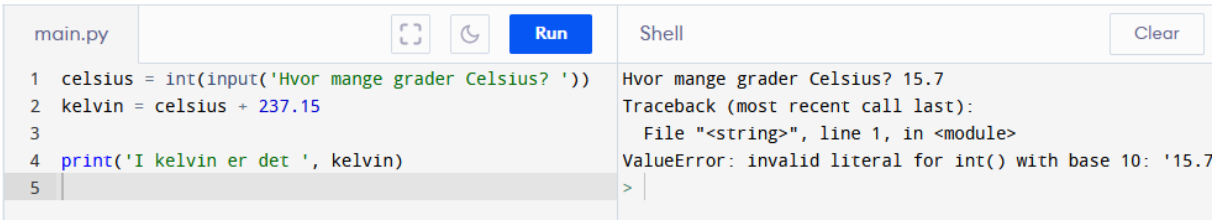
Så er det bare å skrive det vi vil i en print-setning, og koden er ferdig:



Figur 5-74: Konvertere Celsius til Kelvin

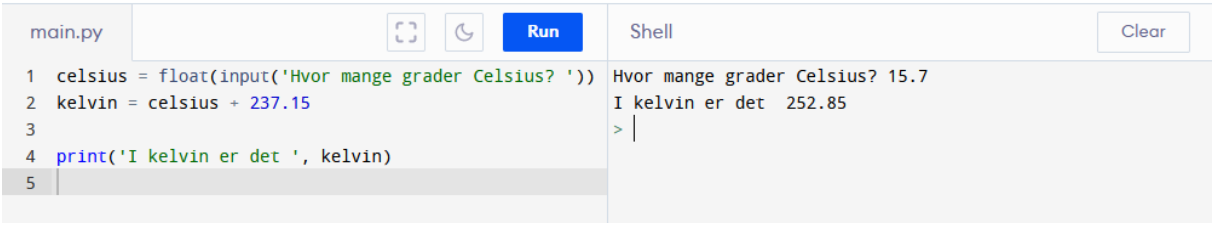
Legg her merke til at i print-setningen har vi brukt komma istedenfor et pluss-tegn. Først har vi skrevet en tekststreng, og så har vi skrevet et tall (variabelen kelvin er av typen int).

Vi har med denne koden tvunget brukeren til å skrive inn et heltall. Om man prøver å skrive inn et desimaltall i konsollen når man blir bedt om grader Celsius, vil man få en feilkode:



Figur 5-75: Skrive inn desimaltall når koden forventer heltall

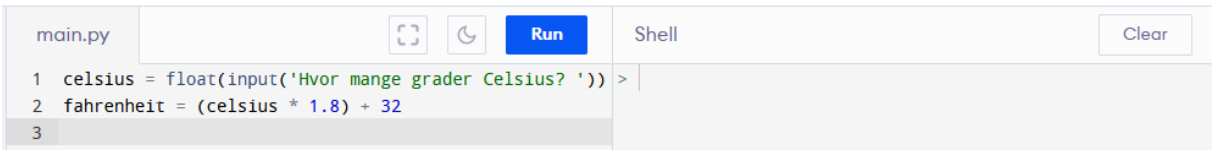
Den sier at man ikke kan skrive inn desimaltall når variabelen ber om et heltall. Dette kan enkelt endres ved å skrive float istedenfor str i variabelen celsius:



Figur 5-76: Tillate input() av desimaltall

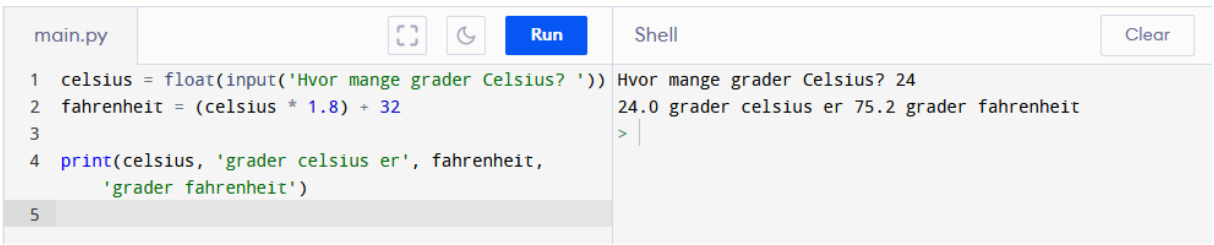
5.1.8.2 Python eksempel 3 – Å sette sammen mange elementer: Konvertering fra Celsius til Fahrenheit

Vi starter med å definere variabelen celsius med et input-felt som tillater desimaltall, og en variabel fahrenheit som regner ut den tilsvarende verdien i Fahrenheit:



Figur 5-77: Sette inn flere matematiske operasjoner direkte i en variabel

Deretter skriver vi print()-setningen. Her skal vi bruke både celsius-variabelen og fahrenheit-variabelen, og skrive litt tekst mellom disse variablene:



Figur 5-78: Konvertere Celsius til Fahrenheit

5.1.9 Sammenligningsoperatorer

Vi har allerede sett at likhetstegn brukes for å tilordne variabler. Når vi så skal sammenligne to elementer, må vi bruke dobbelt likhetstegn. Resultatet av en sammenligning er en boolsk verdi, True (sant) eller False (usant). Her er en oversikt over alle sammenligningsoperatorene (en operand er et objekt som sjekkes av en operator):

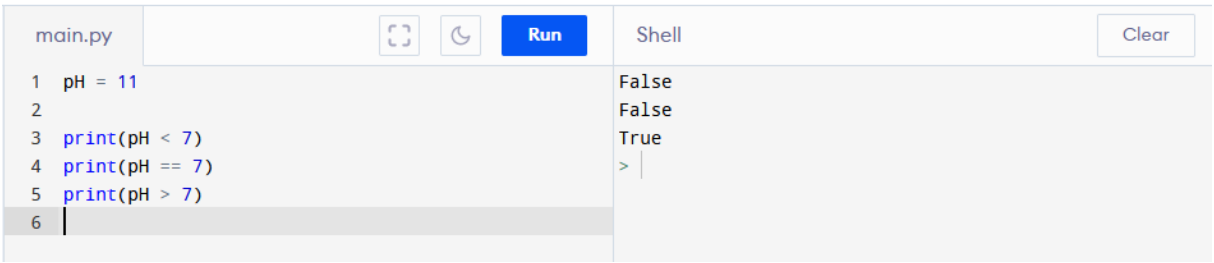
Tabell 5-4: Sammenligningsoperatorer

Operator	Betydning	Beskrivelse	Eksempel	Output
==	Lik	Hvis verdien til to operander er like, er uttrykket sant	10 == 5	False
!=	Ulik	Hvis uttrykket til to operander ikke er like, er uttrykket sant.	10 != 5	True
<>	Ulik	Samme som !=	10 <> 5	True
>	Større enn	Hvis operanden til venstre er større enn operanden til høyre, er uttrykket sant.	10 > 5	True
<	Mindre enn	Hvis operanden til venstre er mindre enn operanden til høyre, er uttrykket sant.	10 < 5	False
>=	Større enn eller lik	Hvis operanden til høyre er større enn eller lik operanden til høyre, er uttrykket sant.	10 >= 5	True
<=	Mindre enn eller lik	Hvis operanden til venstre er større enn eller lik operanden til høyre, er uttrykket sant.	10 <= 5	False

Man kan bruke sammenligningsoperatorer mellom alle datatyper, altså sammenligne to tekststrenger, to tall, to lister eller to boolske verdier.

5.1.9.1 Python eksempel 4 – Sammenligningsoperatorer: Sjekke om en vannløsning er sur, basisk eller nøytral

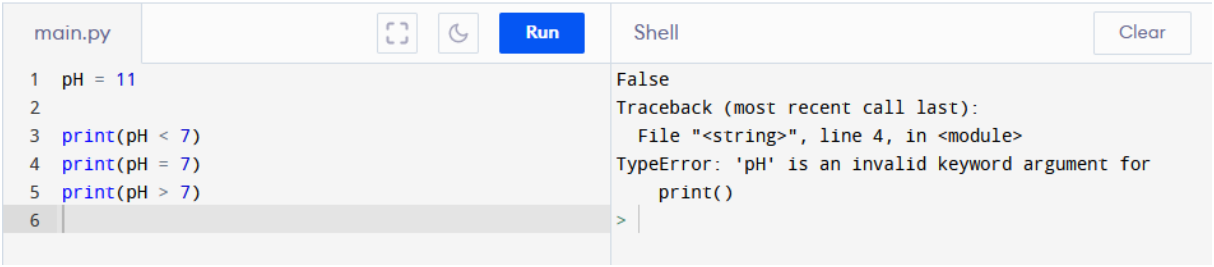
La oss se på et eksempel:



Figur 5-79: Sammenligningsoperatorer og boolske verdier

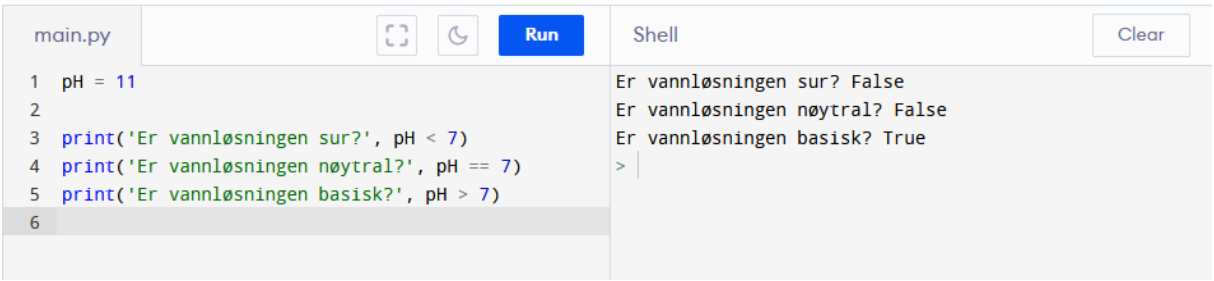
Først har vi definert at variabelen pH skal inneholde verdien 11. Deretter har vi brukt print-setninger for å sjekke om pH er mindre enn, lik eller større enn 7. Verdiene som kommer ut i konsollen til høyre kalles boolske verdier: Altså True (sant) eller False (usant). Som vi kan se, så ser Python at vår definerte verdi for pH (11) ikke er mindre enn 7, ikke er lik 7, men at den er større enn 7.

Om vi istedenfor dobbelt likhetstegn hadde brukt enkelt, så ville ikke Python kunne ha kjørt koden. Dette fordi man ikke kan definere variabler direkte i print-setninger:



Figur 5-80: Visualisering av at man ikke kan bruke enkelt likhetstegn for sammenligning

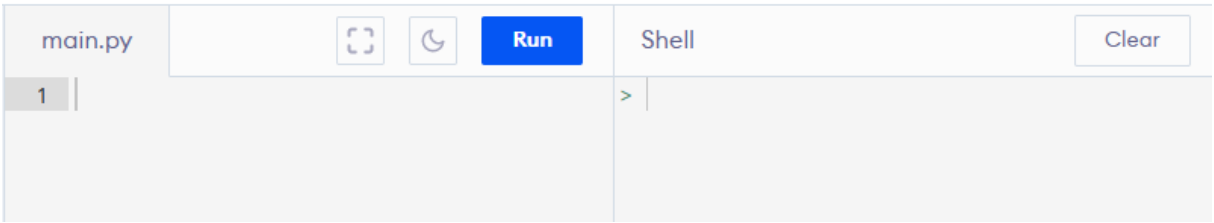
La oss gå tilbake til den koden som var korrekt. Også her kan man legge til forklarende tekst i print-setningene, samtidig som man bruker sammenligning. Da vil utskriften i konsollen være litt mer forklarende:



Figur 5-81: Sjekke om en vannløsning er sur, basisk eller nøytral

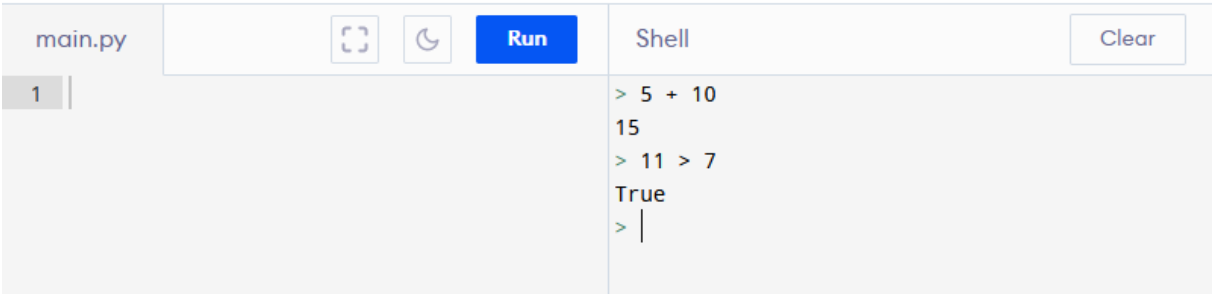
5.1.9.2 Å bruke konsollen til å skrive kode

Vi har sett på at man kan skrive input i konsollen når koden som er skrevet i tekstfilen ber om det, men vi kan også skrive direkte i konsollen. Dette kan være svært hjelpsomt for eksempelvis enkle utregninger man ikke ønsker å ta vare på.



Figur 5-82: Pilen i konsollen indikerer at man kan skrive tekst

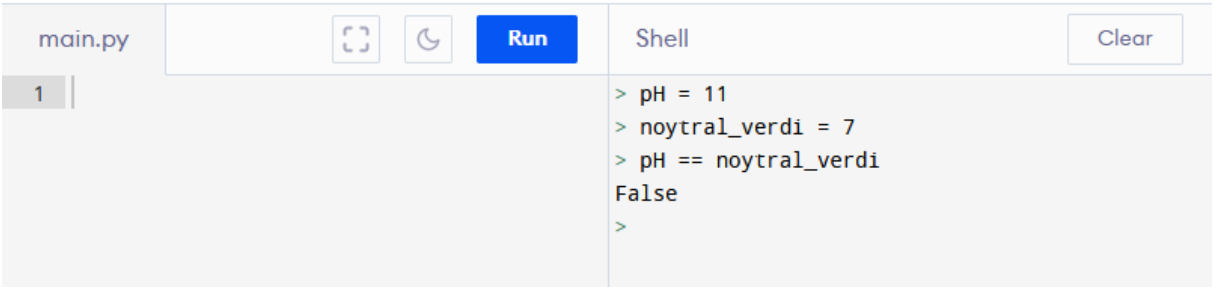
Pilen > i konsollen indikerer at her kan man begynne å skrive. La oss sjekke hva 5 + 10 er, og om 11 er større enn 7:



Figur 5-83: Regne- og sammenligningsoperasjoner i konsollen

Teksten er skrevet rett i konsollen, uten noe som helst kode i tekstfilen. Som dere kan se, så er den en pil der vi har skrevet kode, så er det ingen pil der svaret har kommet som 15, så en ny pil der vi har skrevet mer kode, og så ingen pil igjen der svaret har kommet som True.

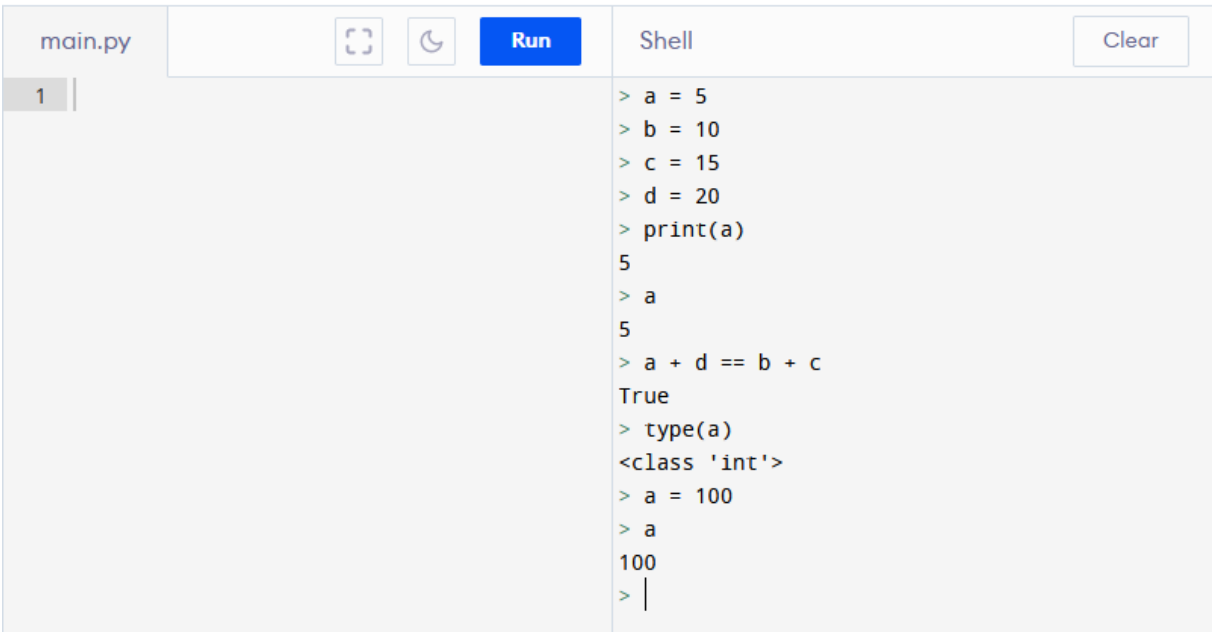
Man kan også bruke variabler i konsollen:



Figur 5-84: Definere variabler i konsollen

Her har man sagt at variabelen pH skal inneholde verdien 11, variabelen noytral_verdi skal inneholde verdien 7, og så har man brukt en sammenligningsoperator mellom de to variablene. Her kan dere se at konsollen ikke gjør noe annet med variabelen enn å ta vare på den. Den skriver ikke ut noe etter at man tilordner variablene, for vi har ikke bedt den om å skrive ut noe. Akkurat som når man skriver i tekstfilen, må programmet få beskjed om hva den skal gjøre for å gjøre noe. Å tilordne en verdi til en variabel, ber bare programmet om å ta vare på denne verdien – ikke gjøre noe med den. Når man derimot bruker en sammenligningsoperator, dobbelt likhetstegn, så spør man "Er disse to verdiene like?", og det svarer programmet på.

Man kan gjøre mye det samme i konsollen som i tekstfilen. Etter at man har tilordnet variabler, kan man skrive de ut. Man kan bruke en print()-setning, men det er ikke nødvendig i konsollen, slik man må i tekstfilen om man vil at noe skal skrives ut til nettopp konsollen. Man kan også sammenligne flere variabler i en og samme linje, og man kan sjekke hvilken type en variabel inneholder. Vær oppmerksom på at man også her kan tilordne en ny verdi til en variabel.



Figur 5-85: Mange utregninger blir raskt uoversiktlig i konsollen

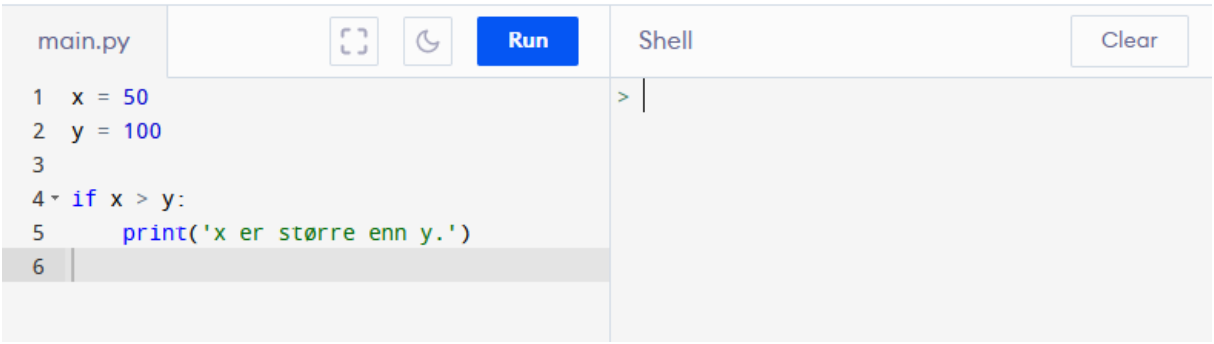
Å bruke tekstfilen til lange operasjoner blir raskt uoversiktlig, men til små regneoperasjoner kan den enkelt brukes som en kalkulator.

5.1.10 If, else og else if

5.1.10.1 If

En if-setning brukes for å sjekke om en betingelse er oppfylt. If (hvis) en betingelse er oppfylt, gjør det som står etter if-setningen.

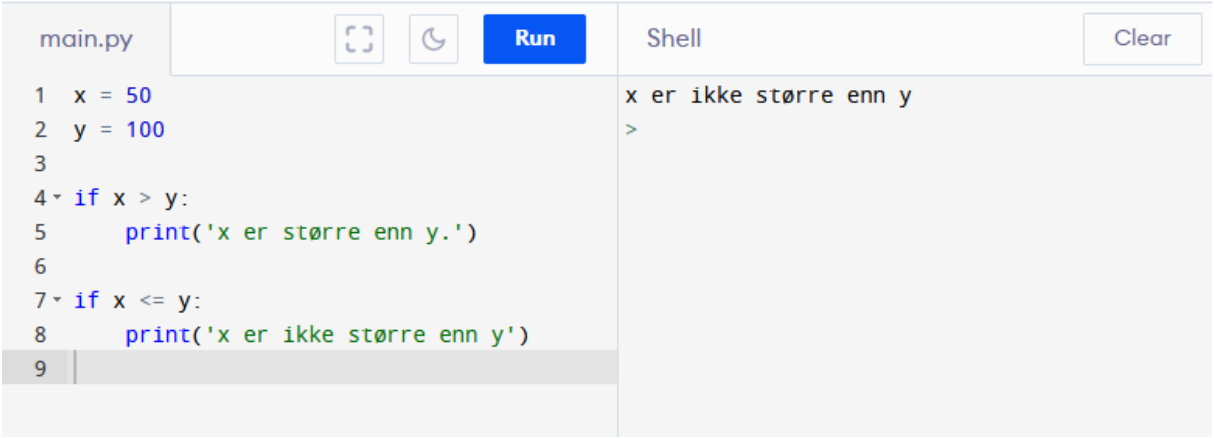
Man kan for eksempel bruke sammenligningsoperatorer med en if-setning, og bruke print() skrive ut en setning dersom betingelsen er oppfylt:



Figur 5-86: En enkel if-setning

Som dere kan se, sjekkes det i linje fire om x er større enn y. Hvis den er det, skrives det ut det som står i linje 5. Siden 50 ikke er større enn 100, er ikke if-betingelsen oppfylt. Det skrives derfor ikke ut noe, for vi har ikke skrevet hva som skal skje dersom betingelsen ikke er oppfylt.

Man kan velge å skrive opp betingelser for hver enkelt mulighet:

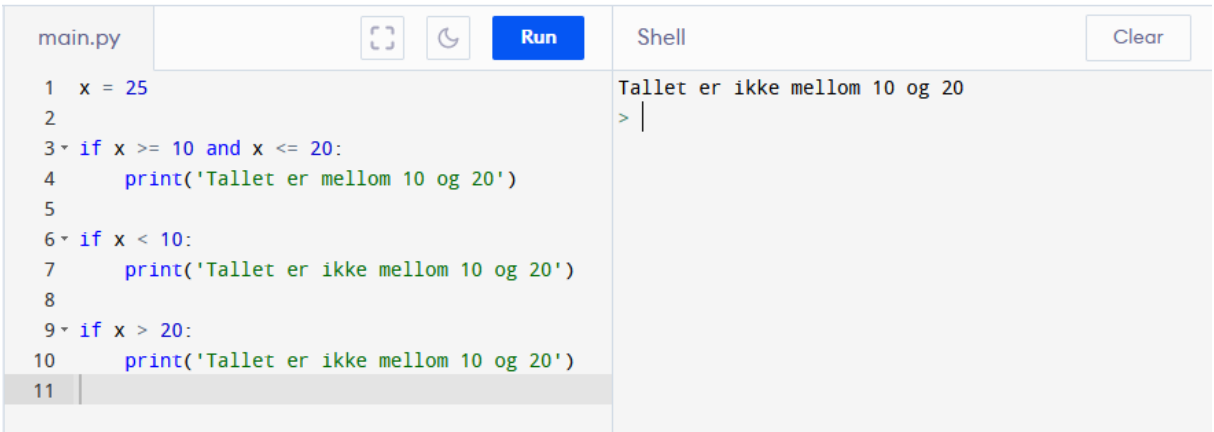


Figur 5-87: Flere if-setninger

Betingelsen i linje 4 er fortsatt ikke oppfylt, så programmet hopper over linje 5. Betingelsen i linje 7 er derimot oppfylt, og derfor vil det som står i linje 8 skrives ut.

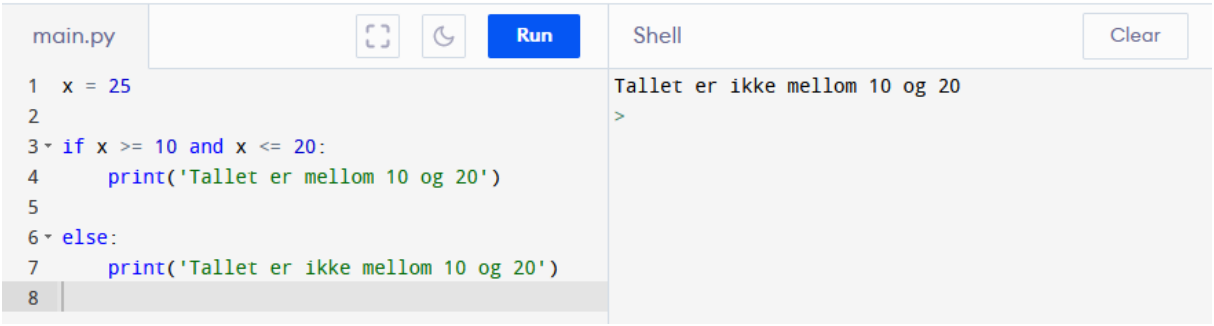
5.1.10.2 Else

Det kan i enkelte tilfeller bli svært tidkrevende å skrive if-setninger for hver enkelt betingelse. Man kan for eksempel sjekke om et tall er en verdi fra 10 til og med 20 på følgende måte:



Figur 5-88: Unødvendig mange if-setninger

I mange situasjoner vil man ikke kunne komme på alle unntak, eller man må skrive svært mange linjer med kode for å ta høyde for alle forutsetninger. Heldigvis finnes det også en funksjon som heter else, som brukes sammen med if-setninger. If (hvis) en betingelse er oppfylt, kjør den koden her. Ellers (else), kjør den koden her:



Figur 5-89: Setninger med if og else

Mye mindre kode, med akkurat samme utfall.

Her er det noen ting som er viktige:

1. På slutten av en if-setning (linje 3) må man ha med kolon. Dette for å fortelle at "Nå er jeg ferdig med if-setningen, her er det jeg vil at skal skje dersom den inntreffer" (altså at tallet er mellom 10 og 20, inkludert 10 og 20).
2. På linjen etter if-setningen er det et inntrykk (tabulator). Dette fordi at Python skal skjønne at "den koden som står her, den hører til if-setningen ovenfor". Man kan ha mange linjer her, og alle må være med innrykk.
3. Etter else må man også huske kolon.
4. Og som etter if-setningen, må man etter else-setningen ha innrykk på den koden som man ønsker å kjøre. Koden under samme innrykk kalles gjerne en blokk.

Punkt 2 og 4 går vi nærmere inn på neste eksempel om PIN-kode.

Vi har også brukt operasjonen and. Dette skal vi gå nærmere inn på etter at vi har sett på else if.

5.1.10.3 Else if (Elif)

La oss gå litt tilbake, til eksempelet om x = 50 og y = 100. Der skrev vi opp tre if-setninger etter hverandre. Python fungerer sånn at da sjekker den alle tre betingelsene, siden alle tre er if-setninger. Det trenger den jo streng tatt ikke. Hvis første betingelse er oppfylt, vet vi at nummer to og tre ikke vil være oppfylt. Det vil være mer naturlig å sjekke betingelse nummer to dersom betingelse én ikke var riktig. Til dette har vi en funksjon som heter "else if", eller elif. Her setter vi

Figur 5-90: Setninger med if, else if (elif) og else

Her sjekker den først if-setningen i linje 4. Den var ikke riktig, så programmet hoppet til linje 7. Denne var riktig, så den kjører koden som står i linje 8. Da tolker programmet det slik at den ikke trenger å kjøre resten av denne koden, så den kjører ikke else-setningen i linje 10.

Forskjellen på å bruke mange if-setninger versus det å bruke if/elif/else kan illustreres med et eksempel. Vi setter opp tre like if-betingelser, ett med tre if-setninger, og ett med én if/elif/else-setning.

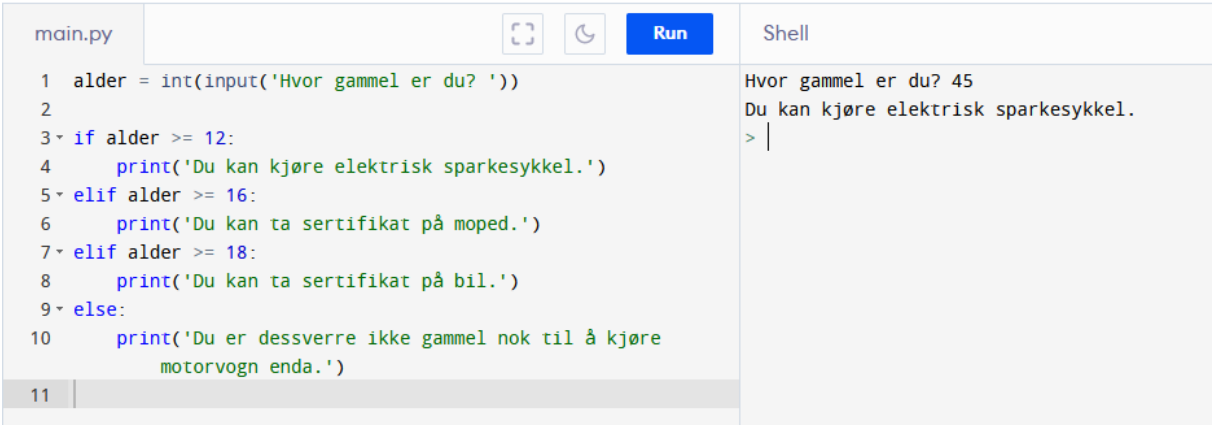
Med én if/elif/else-setning kan dere se at programmet slutter å sjekke etter at den første betingelsen har inntruffet, selv om det i begge tilfeller står at den betingelsen er x == y. Elif (else if) skal kun kjøres dersom den første betingelsen ikke inntreffer:

Figur 5-91: Visualisering av forskjellen på if-elif-else og if-if-if, del 1

Om man setter opp tre if-setninger, så sjekker programmet hver av disse setningene. Alle tre har jo riktig betingelse oppfylt, og vil derfor kjøre tilhørende kode:

Figur 5-92: : Visualisering av forskjellen på if-elif-else og if-if-if, del 2

Dette kan være nyttig å vite om dersom man for eksempel skal foreta en alderskontroll i forbindelse med hvilke motoriserte kjøretøy man kan ta sertifikat på. Vi sjekker for sparkesykkel, moped og bil:



Figur 5-93: Alderssjekk, del 1

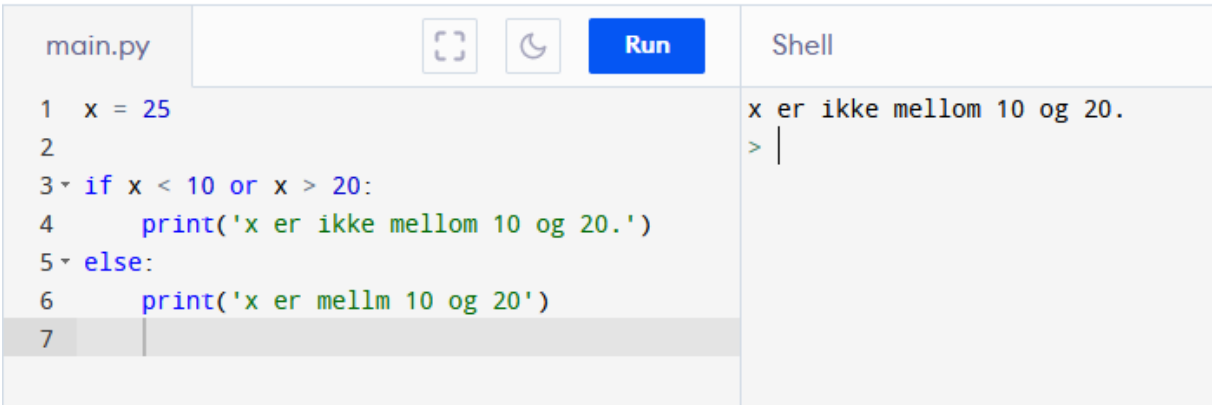
Det er jo uheldig at en bruker som er 45 år får beskjed om at den kun kan kjøre elektrisk sparkesykkel, når tilfellet er at alle tre burde vært oppfylt:

Figur 5-94: Alderssjekk, del 2

5.1.10.4 Logiske operatorer: And og Or

De logiske operatorene brukes dersom man har flere betingelser man vil sjekke. Som vi så i eksempelet om tall mellom 10 og 20, så sjekker man både at tallet er 10 eller over, og at det er 20 eller mindre ved hjelp av And.

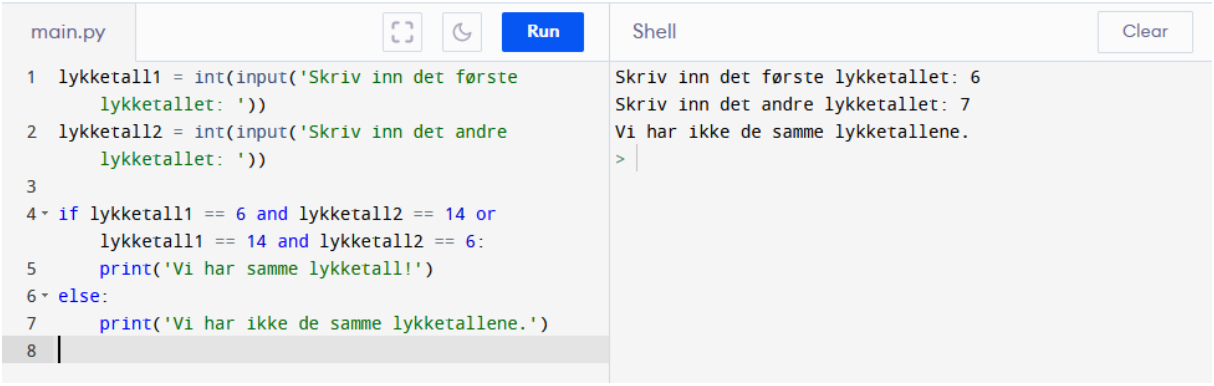
Det andre alternativet er at man sjekker om en betingelser er oppfylt enten med den første betingelsen, eller (or) med den andre betingelsen:



Figur 5-95: Logisk operator or

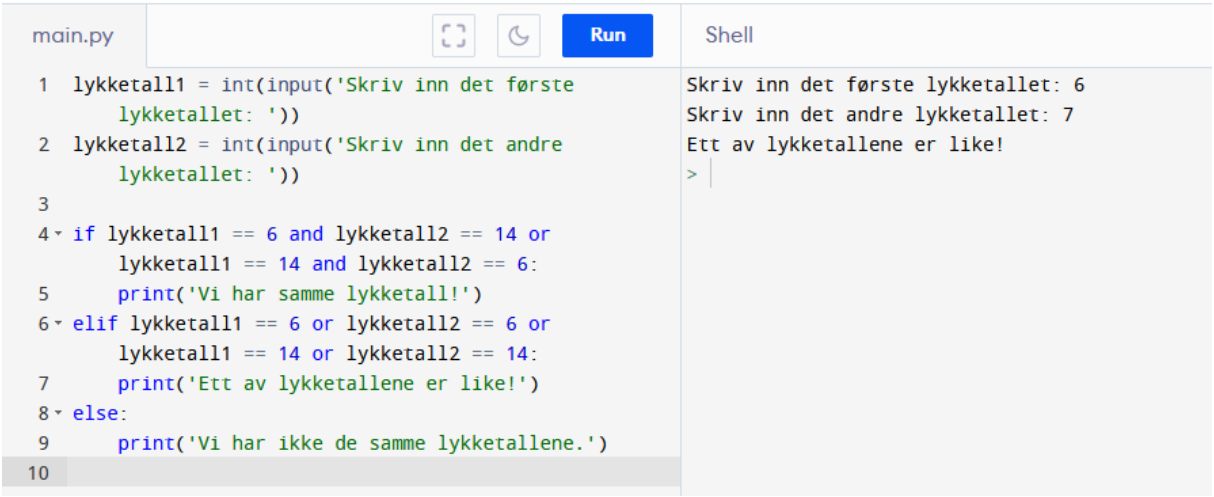
Hvis (if) variabelen x er under 10 **eller (or)** variabelen x er over 20, skriv ut denne tekststrengen.

Man kan også bruke flere and/or-operatorer i samme setning. Her er koderen sine lykketall 6 og 14. Den ber brukeren skrive inn sine to lykketall, og sjekker om det er de samme som koderen sine. Begge to må være de samme for at man skal få beskjed om at de har samme lykketall:



Figur 5-96: Logiske operatorer and og or

Ved å legge til en "else if" hvor man kun bruker den logiske operatoren or, kan man også gi beskjed til brukeren om at ett av tallene er de samme:

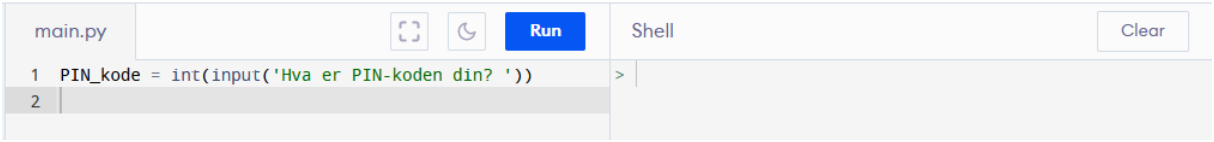


Figur 5-97: Program som sjekker om brukeren har samme lykketall som koderen

5.1.10.5 Python eksempel 5 – Bruk av løkker: Å sjekke om PIN-koden er riktig.

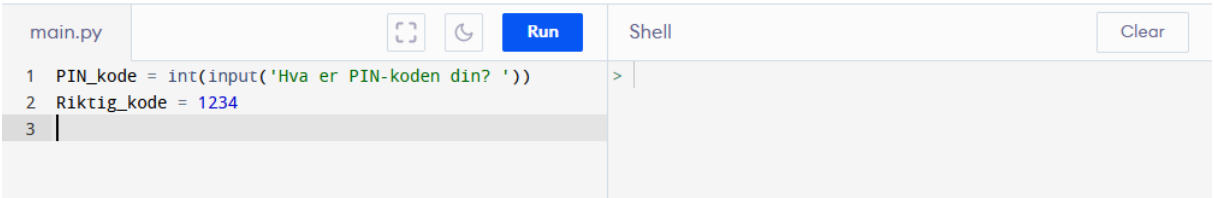
Nå skal vi sette sammen det meste av det som er gått gjennom i dette heftet. Som eksempel bruker vi det samme som for en hvis/ellers-setning i Scratch: Å sjekke om PIN-koden er riktig. Koden i Scratch så slik ut, og nå skal vi skrive det samme i Python (figur 5-36):

Vi ønsker et input som er et tall. Husk derfor å skrive int() rundt feltet input().



Figur 5-98: Opprette variabelen PIN_kode

Deretter må vi definere hva den riktige PIN-koden er for noe:

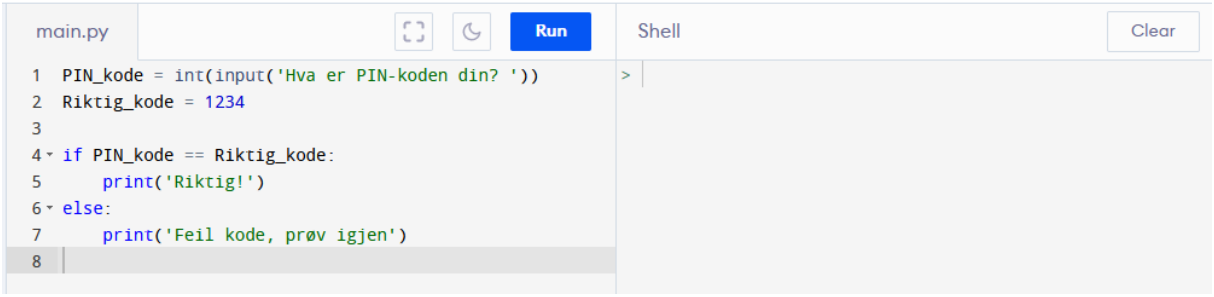


Figur 5-99: Legge til variabel Riktig_kode

Så skal vi skrive if/else-setningen vår:

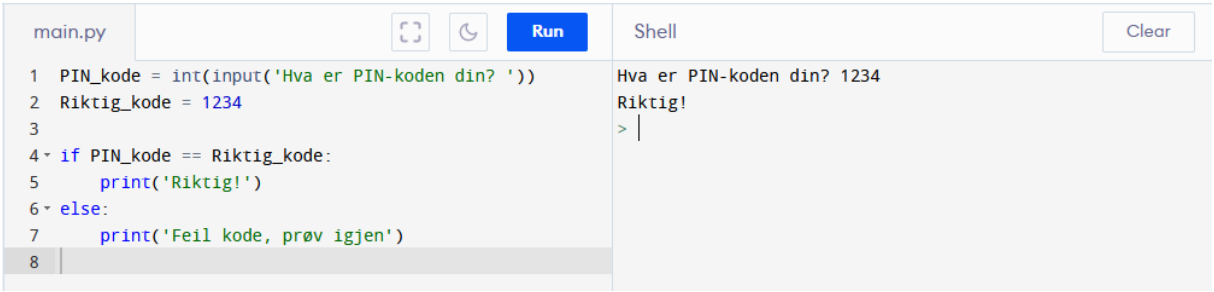
If (hvis) koden er riktig: Kjør denne print-setningen.

Else (ellers): Kjør denne print-setningen.

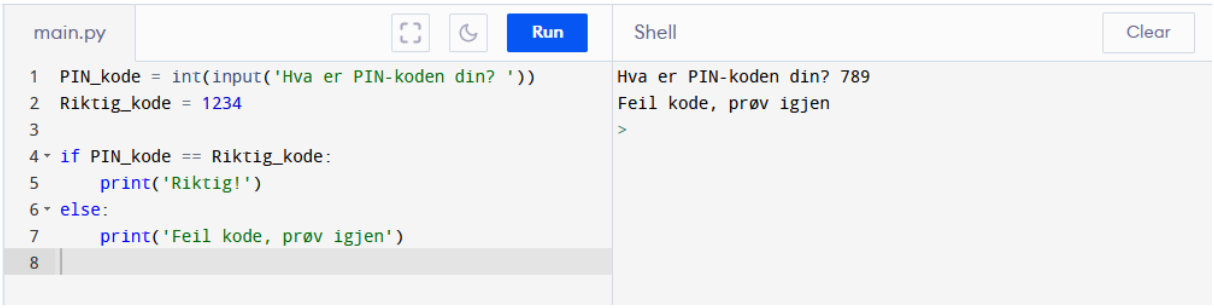


Figur 5-100: Program som sjekker om PIN-koden er riktig

La oss se at koden fungerer:

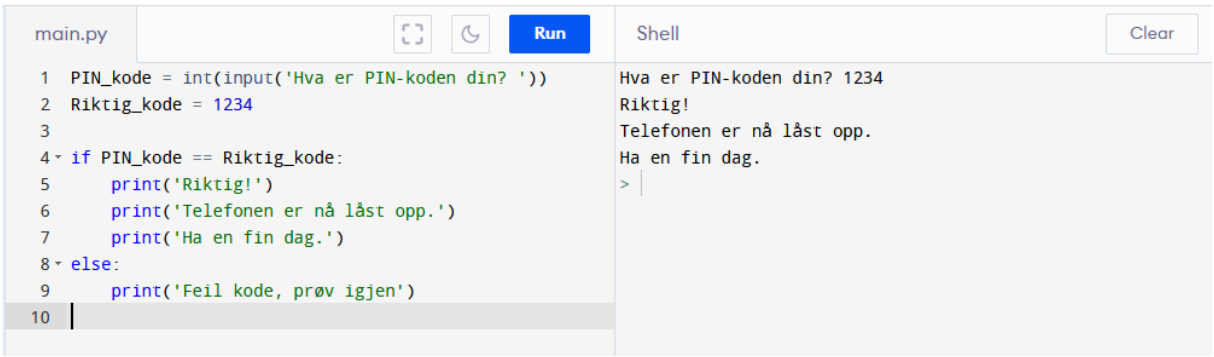


Figur 5-101: Program som sjekker om PIN-koden er riktig: Taste inn korrekt kode



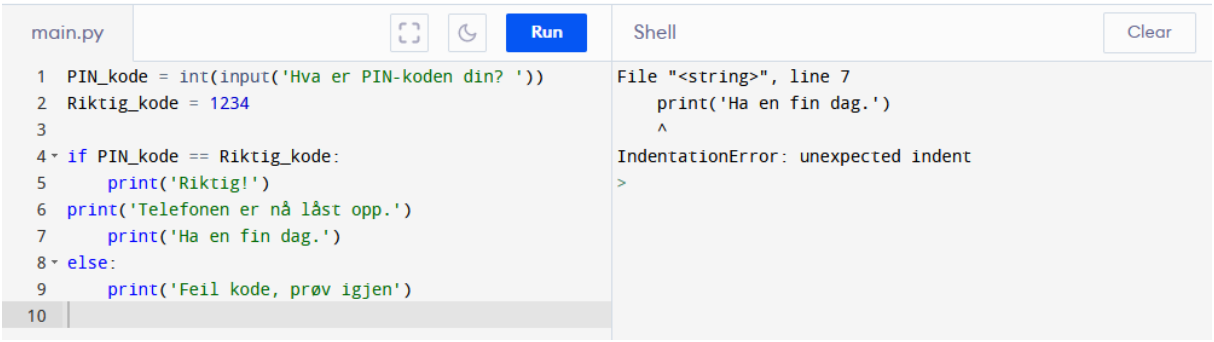
Figur 5-102: Program som sjekker om PIN-koden er riktig: Taste inn feil kode

Som nevnt i forrige delkapittel, kan man ha mange kommandoer i en og samme if-setning. Altså mange linjer med kode som skal kjøres dersom betingelsen er oppfylt. Med riktig innrykk (tabulator), vil Python skrive ut alle tre setningene etter at man har tastet inn den riktige koden:



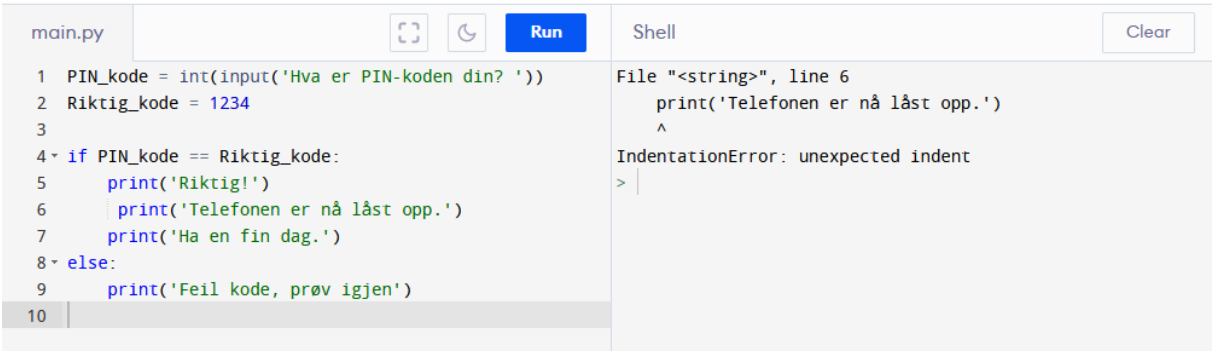
Figur 5-103: Flere print-setninger i en if-betingelse

Dersom man har feil innrykk, så skjønner ikke Python hva du prøver på:



Figur 5-104: Feil tabulering, bruk av tabulator

Det skal ikke mer enn ett enkelt mellomrom til før Python sliter:



Figur 5-105: Feil tabulering, bruk av mellomrom

Dette er en av grunnen til at man sier at programmeringsspråk er "dumme". De responderer kun på klare, forhåndsdefinerte instrukser.

Det er også viktig å nevne at dette kan være ulikt for ulike programmeringsspråk. Vi skal ikke ta for oss andre språk i denne gjennomgangen, men syntes det var viktig å nevne at ikke regler er like for alle språk.

5.1.11 While-løkker

Når man skal gjøre den samme tingen mange ganger, er løkker til god hjelp. Da kan man repetere koden så mange ganger man vil. Vi skal nå se på for-løkker og while-løkker. I hovedsak så brukes while-løkker når man ikke vet hvor mange ganger man trenger å kjøre koden, mens for-løkker benyttes når vi på forhånd vet hvor mange gjentakelser (iterasjoner) vi vil ha.

5.1.11.1 Definisjon og syntaks

1.1.1.1.1.1.1 while-løkke
Definisjon:

Itererer en blokk med kode så lenge testuttrykket er sant.

Syntaks:

```
while test_expression:
      statement(s)
```

Man starter en while-løkke med å skrive while, deretter skriver man betingelsen man vil teste, etterfulgt av kolon. I de tabulerte linjene under skriver man koden som skal repeteres, så lenge betingelsen man skrev er sann.

5.1.11.2 Eksempel på bruk av while-løkke

Vi kan for eksempel skrive ut alle tall helt til man kommer opp til 10, inkludert starttallet. La oss kalle testuttrykket for i. Da blir koden som følger:

```
while i < 11:
print(i)
i += 1
```

Det som skjer her, er at så lenge man skriver inn et tall som er mindre enn 10, vil while-løkken først printe ut tallet. Deretter legges 1 til tallet (i += 1). Deretter starter løkken på nytt. Koden fortsetter å gjenta seg selv helt til man kommer til tallet 11. Da er ikke lenger i mindre enn 11, og Python går videre i programmet.

Man kan selv velge hvor man vil starte, for eksempel ved tallet 5:

Figur 5-106: While-løkke som printer tallene 5 til 10

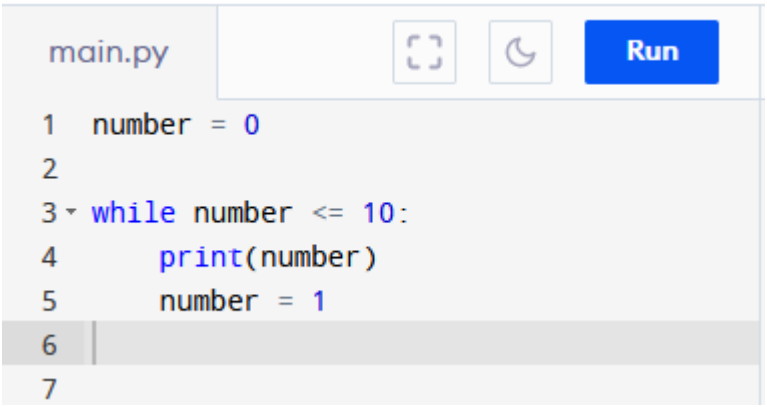
Rekkefølgen her er viktig. Først printer vi ut tallet, deretter legger vi til verdien 1. Dersom vi hadde byttet om på print setningen og i += 1 (linje 4 og 5), ville utskriften sett annerledes ut.

Figur 5-107: While-løkke som printer tallene 6 til 11

I dette tilfellet sender vi inn tallet, så legger vi til 1 og deretter printer vi ut tallet. Derfor er første utskrift 6 istedenfor 5, og siste utskrift 11 istedenfor 10. Testuttrykket er det samme i begge tilfeller, og siste tall som testes er 10. Forskjellen er om vi skriver ut tallet før eller etter vi har lagt til 1.

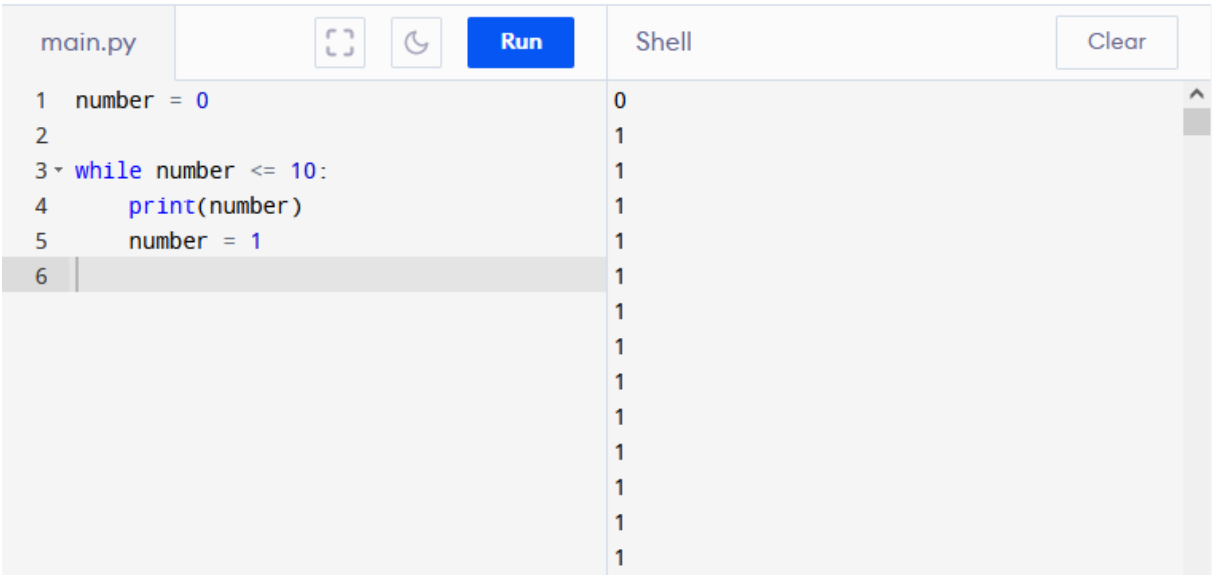
5.1.11.2.1 Uendelige while-løkker

Siden en while-løkke stopper når en viss betingelse ikke lenger gjelder, og ikke et visst antall iterasjoner, kan det raskt oppstå feil. Se for eksempel på denne koden her:



Figur 5-108: Kode med en uendelig while-løkke

Her har vi satt startverdien til 0, og vi har sagt at while-løkken skal kjøre så lenge tallet er mindre eller lik 10. Deretter ber vi om at hvert tall skal printes ut i linje 6. Men i linje 7 har vi skrevet at number = 1. Dette vil jo føre til at løkken starter på nytt, number er under 10, tallet 1 skrives ut, vi setter igjen number = 1, og så gjør den akkurat det samme i det uendelige.



Figur 5-109: Utksrift av en uendelig while-løkke

Da må man redigere løkken sin slik at den kommer seg ut av den uendelige løkken, for eksempel ved å skrive number += 1. Da vil utskriften være tallene 0 til 10.

5.1.12 For-løkker

En for-løkke er som en løkke som går et bestemt antall iterasjoner.

5.1.12.1 Definisjon og syntaks

1.1.1.1.1.2 for-løkke
Definisjon:

Itererer en blokk med kode et bestemt antall ganger.

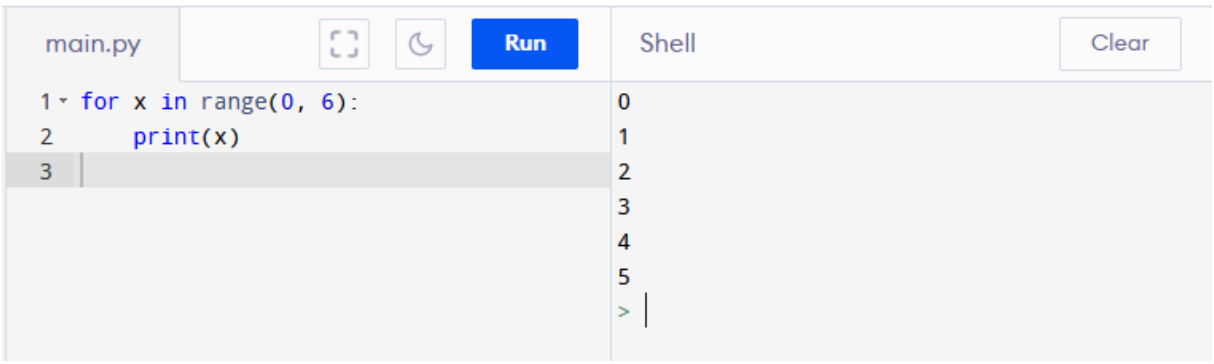
Syntaks:

```
for x in sequence:
      statement(s)
```

Man starter løkken med ordet "for". Deretter en bokstav eller et ord man kan bruke senere i koden, så ordet "in", så den sekvensen man vil sjekke og til slutt et kolon. Det kan være både et tallintervall, en tekststreng, en liste, en tuple, en dictionary eller et sett. Noen av disse har vi gått gjennom, noen går vi gjennom senere i dette heftet og noen overlater vi til dere å undersøke på egenhånd.

5.1.12.2 For-løkke og range

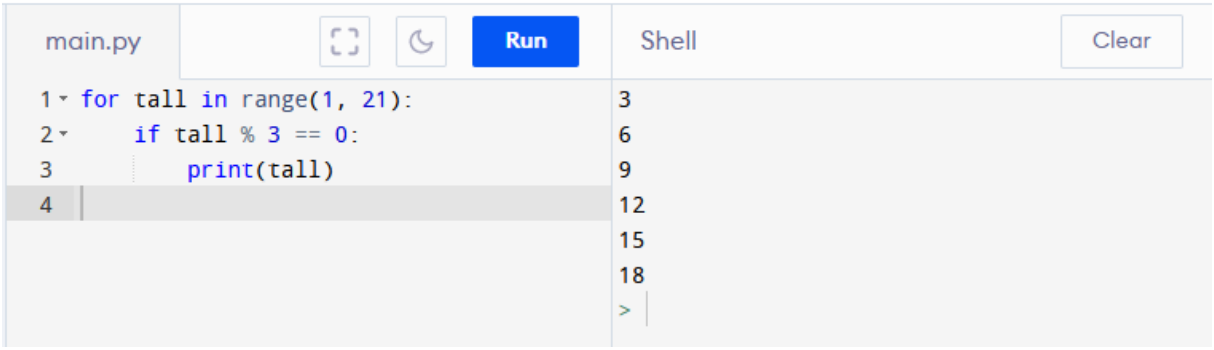
Man kan bruke en for-løkke til å gå gjennom en løkken et visst antall ganger gitt av et intervall, for eksempel 0 til 5. Dersom man vil skrive ut tallene 0 til 5, kan man skrive følgende kode:



Figur 5-110: For-løkke og range()

Det er viktig å merke seg at når man skrive inn intervaller, så kjører koden fra første verdi **til** siste verdi, ikke til og med. Det vil si at når vi ønsker å skrive ut 0 til 5, må vi skrive range(0, 6). Standard startverdi er null, så man om man ønsker å starte på null som i dette tilfellet er det tilstrekkelig å skrive range(6). Da skriver den ut tallene fra 0 til 5. Sluttverdi må alltid være med.

Man kan da for eksempel bruke en for-løkke til å printe ut alle tall som er delelig med 3 i et intervall fra 1 til 20, ved hjelp av en if-setning:

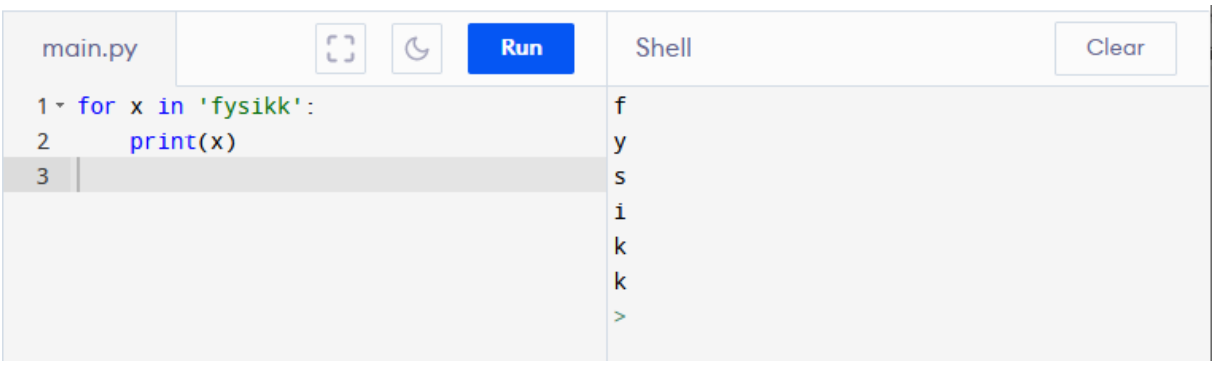


Figur 5-111: For-løkke med range() og en if-setning

Her har vi kombinert en for-løkke med en if-setning. Hvis tallet delt på 3 = 0 (if tall % 3 == 0), så skal det tallet skrives ut. Husk også kolon både etter for-uttrykket og etter if-setningen.

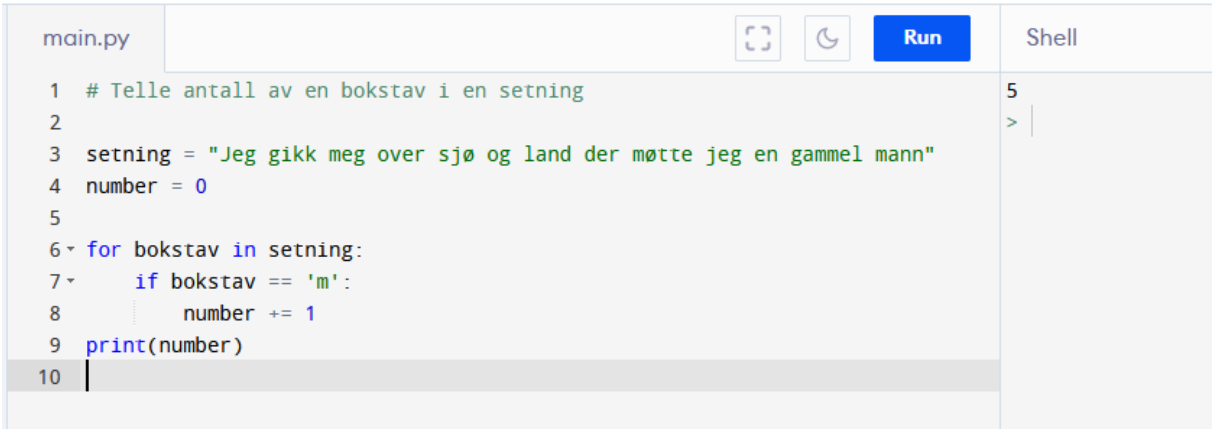
5.1.12.3 For-løkke og tekststrenger

En tekststreng er et gitt antall symboler. Ordet "fysikk" består av 6 bokstaver, og dermed ønsker vi at løkken skal gå et gitt antall ganger. Dersom vi vil printe ut hver enkelt bokstav i det ordet, kan vi gjøre det med følgende kode:



Figur 5-112: For-løkke og tekststrenger

Dette ser kanskje ikke så nyttig ut, men vi kan også bruke denne metoden til å for eksempel telle antall bokstaver i en lengre tekststreng. La oss si at vi vil telle antall m-er i en setning. Da oppretter vi først en variabel som inneholder tekststrengen, slik at vi ikke trenger å skrive hele tekststrengen i for-løkken. Så oppretter vi en variabel som kan holde styr på tellingen: Deretter ber vi for-løkken om å skrive ut alle tilfeller av bokstaven m:



Figur 5-113: Telle antall av bokstaven 'm' i en tekststreng

Variabelen setning er tekst-strengen vi vil undersøke.

Variabelen number er telleren vår.

Deretter skriver vi for-løkken vår. Den sier at for hvert tegn i setning (for x in setning), skal vi sjekke om tegnet er lik x (if x == 'm'). Hvis det er det, skal telleren vår (number) økes med 1 (number += 1 kan også skrives som number = number + 1). Til slutt skal tallet skrives ut.

Utskriften av denne koden blir 5.

5.2 Oppgavehefte i Scratch og Python for ungdomsskolen

Å skrive de første programmene

I starten av oppgaveheftet ønsker vi å lage noen oppgaver som gjør at man kommer fort i gang med kodingen.

Formålet med å vise disse oppgavene i Scratch, er å gi litt trening i å lete frem de ulike elementene, sette elementer sammen og å huske å få figuren til å si svaret, altså skrive svaret ut til skjermen.

Tilsvarende er formålet med å vise disse oppgavene i Python å gi en trening i hvordan man skriver kode. Når man er helt fersk som kodeskriver, kan det være vanskelig å holde styr på parenteser, kommaer, plusstegn, likhetstegn og koloner. Dette er derfor mer for å få litt koding i fingrene. Vi håper dere synes elementene som brukes håper er ganske enkle.

Disse oppgavene har ikke et spesielt tema, og er heller ikke kun rettet mot naturfag. Oppgaveteksten er utformet på en slik måte at det skal gå tydelig frem hvilke steg man må gå gjennom for å oppnå ønsket resultat.

5.2.1 Oppgave 1: Fornavn og etternavn

Lag et program med to variabler. Den ene variabelen skal inneholde fornavn, den andre skal inneholde etternavn. Skriv deretter ut de to navnene til skjermen, i samme setning.

5.2.1.1 Algoritme:

1) Opprett en variabel fornavn og en variabel etternavn.

2) Sett disse variablene sammen og skriv de ut til skjermen.

    a. Scratch: Bruk en sett-sammen blokk.
    b. Python: Sett variablene i samme print()-setning med et komma mellom.

5.2.1.2 Program Scratch:

Figur 5-114: Oppgave 1: Fornavn og etternavn i Scratch

5.2.1.3 Program Python:

```
fornavn = 'Kari'

etternavn = 'Nordmann'


print(fornavn, etternavn)
```

5.2.2 Oppgave 2: Bruk av input()

Gjør det samme en gang til i Python, men med input istedenfor å skrive navnene direkte i koden.

5.2.2.1 **Program Python:**

```
fornavn = input('Hva er fornavnet ditt? ')
etternavn = input('Hva er etternavnet ditt? ')
print(fornavn, etternavn)
```

5.2.3 Oppgave 3: Sammenligning av to tall, bruk av if/else

Lag et program som sjekker om et tall er over 37,5. Dersom tallet er høyere, skal det skrives ut at brukeren har feber. Dersom tallet er 37,5 eller lavere, skal det skrives ut at brukeren ikke har feber. Dette er en kraftig forenkling av konseptet feber, men formålet er som sagt mer å bli kjent med kodingen enn at oppgaven er helt korrekt formulert. Husk at i programmering så må man bruke punktum i desimaltall, ikke komma.

5.2.3.1 Program Scratch:

Figur 5-115: Oppgave 3: Sammenligning av to tall, bruk av hvis/ellers i Scratch

5.2.3.2 Program Python:

```
# Input tolkes alltid som tekststreng, og kan da ikke brukes til
sammenligning med et tall.

# Husk derfor float() rundt input(), slik at brukeren kan skrive inn
desimaltall.


tall = float(input('Hvilket tall vil du sjekke? '))


# Husk kolon når man har skrevet if eller else.

# Husk å tabulere for det som hører til if og det som hører til else.

if tall > 37.5:

    print('Du har feber!')

else:

    print('Du har ikke feber.')
```

5.2.4 Oppgave 4: Sammenligning av tall, bruk av if/elif/else

Gjør det samme som i forrige oppgave, men hvis tallet er akkurat 37,5 skal det skrives ut at temperaturen er 37,5 grader.

5.2.4.1 Program Scratch:

Figur 5-116: Oppgave 4: Sammenligning av tall, bruk av hvis/ellers hvis/ellers i Scratch

5.2.4.2 Program Python:

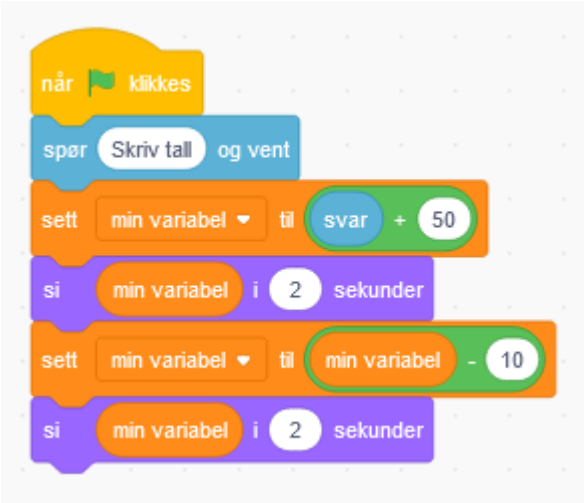
```
# Husk float() rundt input().

tall = float(input('Hvilket tall vil du sjekke? '))

# Husk dobbelt likhetstegn for sammenligning
if tall == 37.5:
    print('Temperaturen er 37,5 grader.')
elif tall > 37.5:
    print('Du har feber!')
else:
    print('Du har ikke feber.')
```

5.2.5 Oppgave 5: Tilordne ny verdi til en variabel

Dette er en Python-oppgave, men vi viser også et eksempel i Scratch.
Opprett en variabel a som inneholder et tall. Bruk ulike tilordningsoperatorer for å gi variabelen ulike nye verdier. Skriv print(a) mellom hver gang du endrer variabelen, slik at du får se at verdien virkelig endrer seg underveis.

5.2.5.1 Eksempel program Scratch

Man kan gjøre det samme i Scratch, men det er litt mer jobb. Det kan uansett være verdt å nevne at dersom man skal bruke flere variabler i Scratch, er det viktig at de har forskjellige navn.



Figur 5-117: Oppgave 5: Tilordne ny verdi til en variabel i Scratch

Om man skriver inn verdien 20, er "min variabel" 80 i den første lilla blokken, og 70 i den andre lilla blokken.

5.2.5.2 Eksempel program Python:

```
a = 50                  # a = 50
print(a)                # a er 50

a += 40                 # a = 50 + 40
print(a)                # a er 90

a -= 30                 # a = 90 - 30
print(a)                # a er 60

a *= 2                  # a = 60 * 2
print(a)                # a er 120
```

```
a /= 4                  # a = 120 / 4
print(a)                # a er 30


a //= 3                 # a = 30 // 3
print(a)                # a er 10

a %= 7                  # a = 10 % 7
print(a)                # a er 3

a **= 3                 # a = 3 ** 2
print(a)                # a er 27
```

5.2.6 Oppgave 6: Bølgelengde

Inspirasjon til denne oppgaven er hentet fra Element 10 (Arntzen, Bækkedal, Fossestøl, & Fægri, 2022, s. 50).

5.2.6.1 Kort beskrivelse av tema

Alle bølger har en frekvens f som er antall svingninger, og en bølgelengde. Også all elektromagnetisk stråling har en frekvens og en bølgelengde. Bølgelengden er definert som lengden mellom to bølgetopper. Frekvens er et mål på hvor mange bølgelengder som passerer i løpet av ett sekund.

Mennesker kan oppfatte bølger som har en bølgelengde mellom ca. 400 og 750 nanometer. Med andre ord: Vi kan se det lyset dersom bølgelengdene er i det spekteret. Eksempel på lys vi ikke kan se, er ultrafiolett lys. Det har bølgelengde mellom 100 og 400 nanometer.

En nanometer er 0,000 000 0001 meter, eller $1*10^{-9}$ meter.

5.2.6.2 Oppgavetekst:

Skriv et program som sjekker om en oppgitt bølgelengde kan ses av mennesker eller ikke. Anta at mennesker kan se lys med bølgelengder fra 400 til 750 nanometer.

5.2.6.3 Problemløsning/Problemløsnings-strategi:

Her etterspørres det en sjekk – altså om en betingelse er oppfylt. Vi ønsker derfor å benytte en if/else-setning. Oppgaven sier ikke noe om hvorvidt vi skal skrive inn tallet direkte i koden, eller om brukeren skal oppgi tallet, så her står vi helt fritt til å velge.

5.2.6.4 Algoritme:

1) Legg inn en variabel som tar vare på tallet som skal sjekkes. I denne gjennomgangen velger vi å la brukeren skrive inn tallet som skal sjekkes.

2) Skriv en if-setning som sjekker om et tall er over 400, men samtidig under 750.

3) Skriv ut svaret til skjermen.

5.2.6.5 Program Scratch:

Figur 5-118: Oppgave 6: Bølgelengde i Scratch

5.2.6.6 Program Python:

```
b_lengde = int(input('Hva er bølgelengden i nanometer? '))

if b_lengde >= 400 and b_lengde <= 750:
    print('Ja, vi kan se dette lyset.')
else:
    print('Nei, vi kan ikke se dette lyset.')
```

5.2.6.7 Eksempler input og output:

Tabell 5-5: Input of output, oppgave 6: Bølgelengde i Python

Input	Output
100	Nei, vi kan ikke se dette lyset.
399	Nei, vi kan ikke se dette lyset.
400	Ja, vi kan se dette lyset!
525	Ja, vi kan se dette lyset!
750	Ja, vi kan se dette lyset!
751	Nei, vi kan ikke se dette lyset.
1000000	Nei, vi kan ikke se dette lyset.

5.2.7 Oppgave 7: Bølgelengde og tilhørende farge

Skriv et program som sjekker hvilken farge en oppgitt bølgelengde har. Brukeren skal selv skulle skrive inn tallet. Bruk disse verdiene for farger:

Tabell 5-6: Bølgelengder med tilhørende farger

Bølgelengde	Output
400-430	Fiolett
431-500	Blå
501-570	Grønn
571-590	Gul
591-650	Oransje
651-750	Rød

Til skjermen skal det skrives både hvilket tall brukeren tastet inn, og hvilken farge det eventuelt tilsvarer.

5.2.7.1 Problemløsning/Problemløsnings-strategi:

Også her etterspørres det en sjekk av et tall. Tall mellom 400 og 750 skal skrives ut med tilhørende farge, alle andre farger skal resultere i et svar som sier at lyset ikke kan oppfattes. Derfor er det også her mulig å bruke if-setninger, men da med en if, flere elif og en else. Til slutt skal både det inntastede tallet og fargen skrives til skjermen.

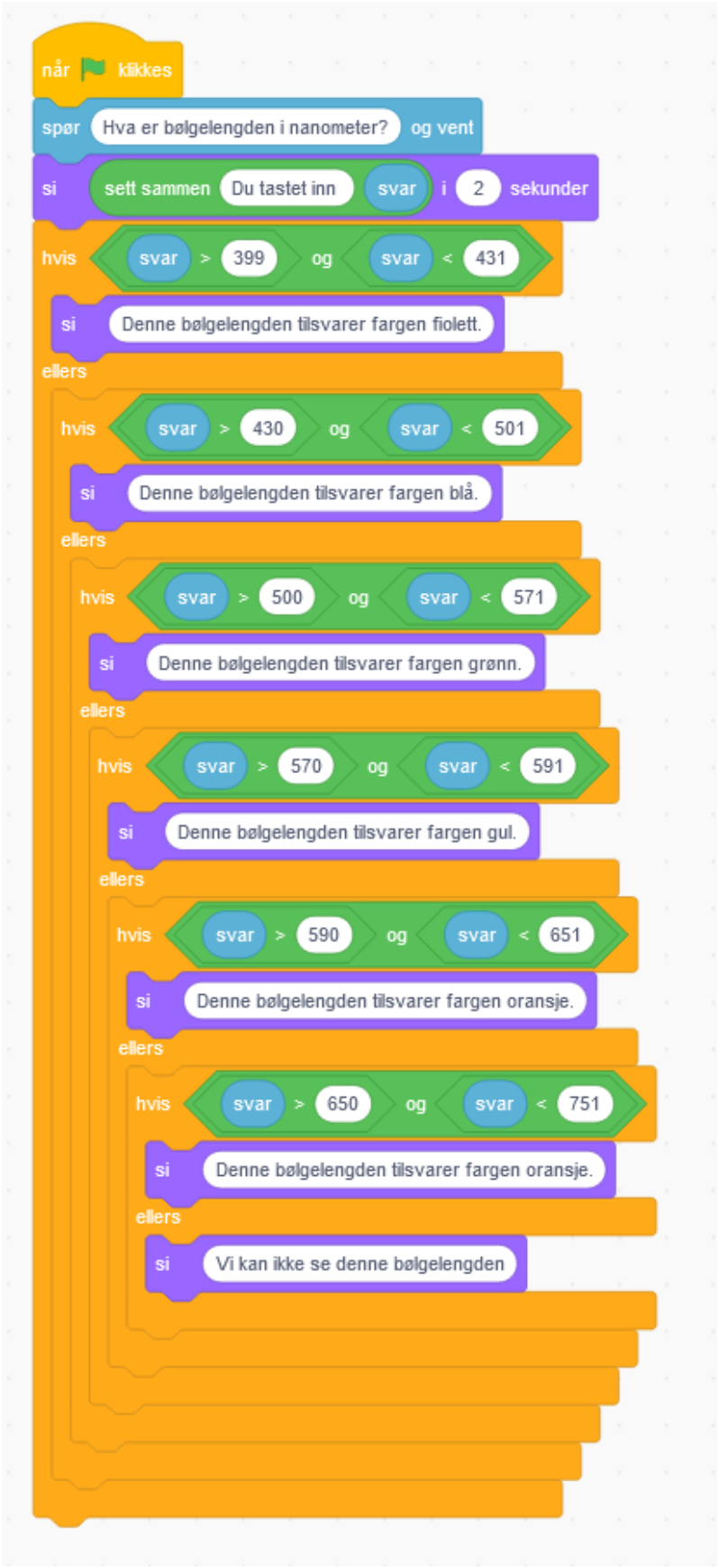
5.2.7.2 Algoritme:

1) Legg inn en variabel som tar vare på tallet som skal sjekkes.
    a. Scratch: Scratch har ikke en egen hvis/ellers hvis/ellers-blokk. Vi må derfor legge flere hvis/ellers-blokker inni hverandre. Husk også at i Scratch så er det ikke noe symbol for "større enn eller lik" eller "mindre enn eller lik". Vi må derfor legge inn verdier "større enn" ett tall lavere enn den første verdien og "mindre enn" ett tall høyere enn den siste verdien.
    b. Python: Når man bruker input() i Python må man huske å legge int() rundt input, slik at programmet skjønner at det er et tall som ble tastet inn.

2) Skriv en if-setning som sjekker om er mellom 400 og 430, mellom 431 og 500 og de resterende intervallene.

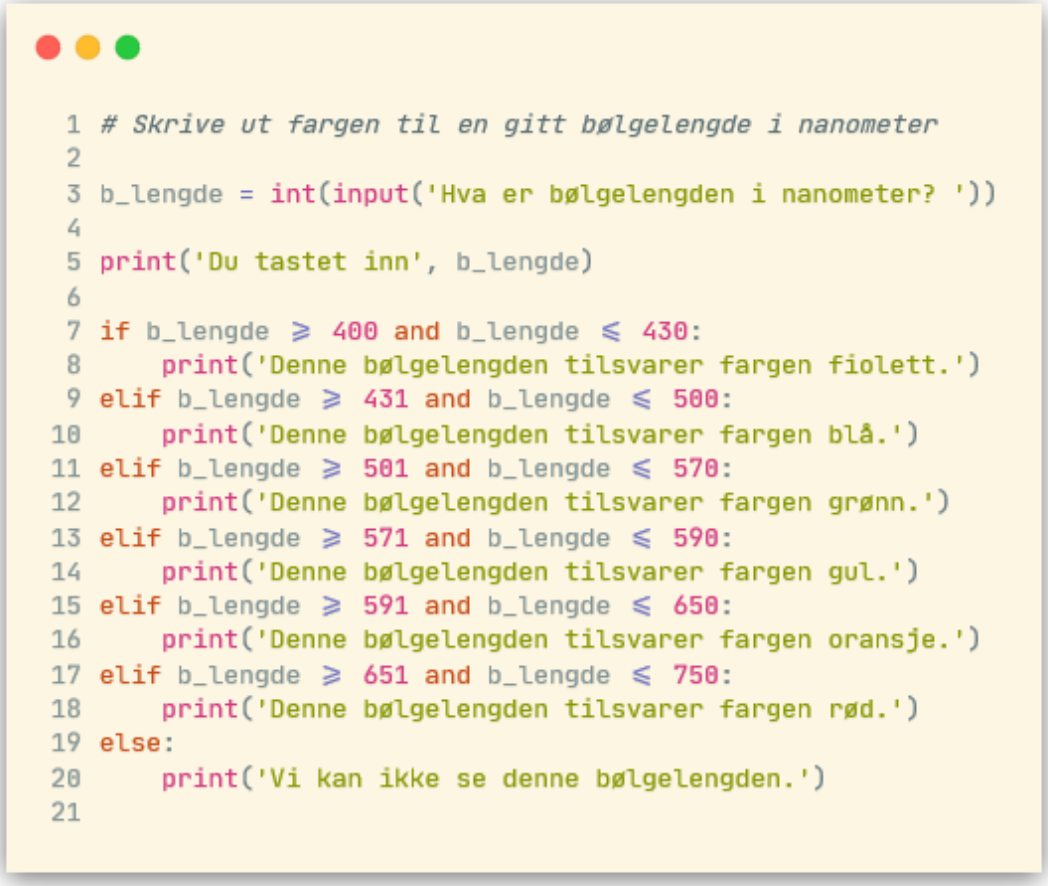
3) Legg til en print-setning i hver av if-setningene som henter variabelen hvor tallet ble skrevet inn, og samtidig skriver ut fargen det tilsvarer.

5.2.7.3 Program Scratch:



Figur 5-119: Oppgave 7: Bølgelengde og tilhørende farge i Scratch

5.2.7.4 Program Python:



```
# Skrive ut fargen til en gitt bølgelengde i nanometer

b_lengde = int(input('Hva er bølgelengden i nanometer? '))

print('Du tastet inn', b_lengde)

if b_lengde ⩾ 400 and b_lengde ⩽ 430:
    print('Denne bølgelengden tilsvarer fargen fiolett.')
elif b_lengde ⩾ 431 and b_lengde ⩽ 500:
    print('Denne bølgelengden tilsvarer fargen blå.')
elif b_lengde ⩾ 501 and b_lengde ⩽ 570:
    print('Denne bølgelengden tilsvarer fargen grønn.')
elif b_lengde ⩾ 571 and b_lengde ⩽ 590:
    print('Denne bølgelengden tilsvarer fargen gul.')
elif b_lengde ⩾ 591 and b_lengde ⩽ 650:
    print('Denne bølgelengden tilsvarer fargen oransje.')
elif b_lengde ⩾ 651 and b_lengde ⩽ 750:
    print('Denne bølgelengden tilsvarer fargen rød.')
else:
    print('Vi kan ikke se denne bølgelengden.')

```

Figur 5-120: Oppgave 7: Bølgelengde og tilhørende farge i Python

```
# Skrive ut fargen til en gitt bølgelengde i nanometer

b_lengde = int(input('Hva er bølgelengden i nanometer? '))

print('Du tastet inn', b_lengde)

if b_lengde >= 400 and b_lengde <= 430:
    print('Denne bølgelengden tilsvarer fargen fiolett.')
elif b_lengde >= 431 and b_lengde <= 500:
    print('Denne bølgelengden tilsvarer fargen blå.')
elif b_lengde >= 501 and b_lengde <= 570:
    print('Denne bølgelengden tilsvarer fargen grønn.')
elif b_lengde >= 571 and b_lengde <= 590:
```

```
        print('Denne bølgelengden tilsvarer fargen gul.')
    elif b_lengde >= 591 and b_lengde <= 650:
        print('Denne bølgelengden tilsvarer fargen oransje.')
    elif b_lengde >= 651 and b_lengde <= 750:
        print('Denne bølgelengden tilsvarer fargen rød.')
    else:
        print('Vi kan ikke se denne bølgelengden.')
```

5.2.7.5 Eksempel input og output

Tabell 5-7: Input og output, oppgave 7: Bølgelengde og tilhørende farge i Python

Input	Output
100	Du tastet inn 100 Vi kan ikke se denne bølgelengden.
410	Du tastet inn 410 Denne bølgelengden tilsvarer fargen fiolett.
550	Du tastet inn 550 Denne bølgelengden tilsvarer fargen grønn.
651	Du tastet inn 651 Denne bølgelengden tilsvarer fargen rød.
750	Du tastet inn 750 Vi kan ikke se denne bølgelengden.

5.2.8 Oppgave 8: Rettlinjet bevegelse

Inspirasjon til denne oppgaven er hentet fra Ergo 1 (Callin et al., 2021)

5.2.8.1 Kort beskrivelse av tema

Fart v er definert som v = s / t der s er strekningen et legeme beveger seg i løpet av en tidsperiode t.

5.2.8.2 Oppgave:

Et legeme holder konstant fart v. Hvor lang distanse s forflytter legemet seg i løpet av en gitt tid t?

5.2.8.3 Problemløsning/Problemløsnings-strategi:

Om en kjenner verdien til to av de tre parameteren i ligningen, kan en finne verdien til den tredje parameteren ved enkel omskriving av ligningen. Vi skriver om ligningen med hensyn på s, og får

s = v * t

5.2.8.4 Algoritme:

1) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. v for fart
    b. t for tid
    c. s for strekning

2) Definere formel for beregning av strekning => s = v * t

3) Benytte input()-funksjonen som er inkludert i Python til å ta imot verdiene for fart v og tid t som brukeren taster inn.

4) Det forventes at brukeren skriver inn desimaltall, og programmet tilordner inntastede verdier til tilhørende variabler, etter at de er konvertert til desimaltall ved hjelp av Pythons egen float()-funksjon.

5) Skriver svaret på beregningen til skjerm, med egnet ledetekst.

5.2.8.5 Program:

```python
# Rettlinjet bevegelse med konstant fart

# Informasjon om bevegelsen
v = float(input('Skriv inn farten v (m/s): '))  # fart, m/s
t = float(input('Skriv inn tiden t (s): '))     # tid, s
s = 0.0 # distanse, m

# Beregning
s  = v * t

# Skriver ut resultatet
print(f'Med en fart på {v}m/s i {t}s har legemet beveget seg {s} meter')
```

Figur 5-121: Oppgave 8: Rettlinjet bevegelse i Python

5.2.8.6 Output:

```
Skriv inn farten v (m/s): 12
Skriv inn tiden t (s): 5.5
Med en fart på 12.0m/s i 5.5s har legemet beveget seg 66.0 meter
```

5.3 Innføringshefte i Python for videregående skole

5.3.1 Forkunnskaper

Før man går i gang med dette heftet, er det noen grunnleggende momenter man må vite om, som er gjennomgått i forrige hefte. Her er en kort oppsummering av disse:

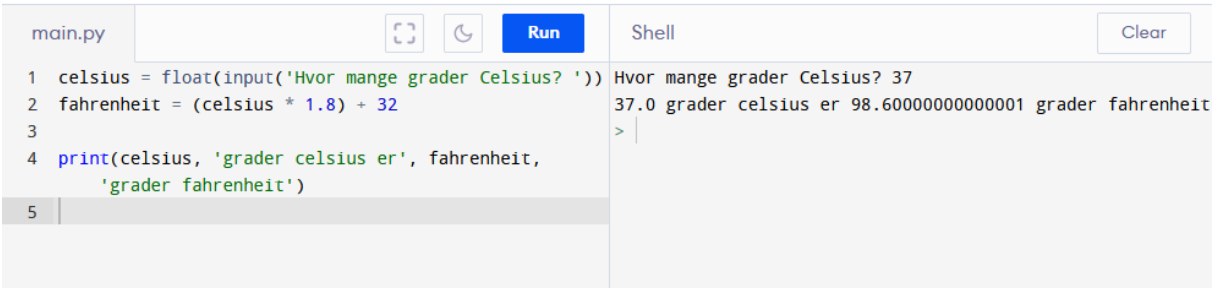
- Forskjell på tekstfil og konsoll
- Print()-funksjonen og kommentarer
- Datatyper
    - String
    - Integer
    - Float
    - Bool
- Variabler, variabelnavn og å gi en variabel ny verdi
    - Variabelnavn kan kun inneholde bokstaver fra A til z, tall fra 0 til 9 og understreker, men kan ikke starte med et tall.
    - Variabelnavn skiller mellom store og små bokstaver
- Input()-funksjonen
    - Tolkes alltid som en tekststreng
- Matematiske operatorer
    - Addisjon (+), subtraksjon (-), multiplikasjon (*), divisjon (/), heltallsdivisjon (//), modulus (%) og eksponent (**)
- Tilordningsoperatorer
    - =, +=, -=, *=, /=, //=, %= og **=
- Konkatenering og å sette sammen variabler av ulike typer
    - Forskjellen på å bruke plusstegn og komma
    - Metoder for å endre en type variabel til en annen type variabel
- Sammenligningsoperatorer
    - Lik (==), ulik, (!= eller <>), større enn (>), mindre enn (<), større en eller lik (>=) og mindre enn eller lik (<=).
- Bruke konsollen til å skrive kode
- Setninger med if, elif (else if) og else
- Bruk av logiske operatorer And og Or.
- For-løkker
- While-løkker

Om du er usikker på noe av dette, foreslår vi at du går gjennom det heftet, eventuelt finner informasjonen på andre måter.

5.3.2 Bestemme antall desimaler med round()

Vi fortsetter med et eksempel fra forrige innføringshefte, hvor vi konverterte celsius til fahrenheit.

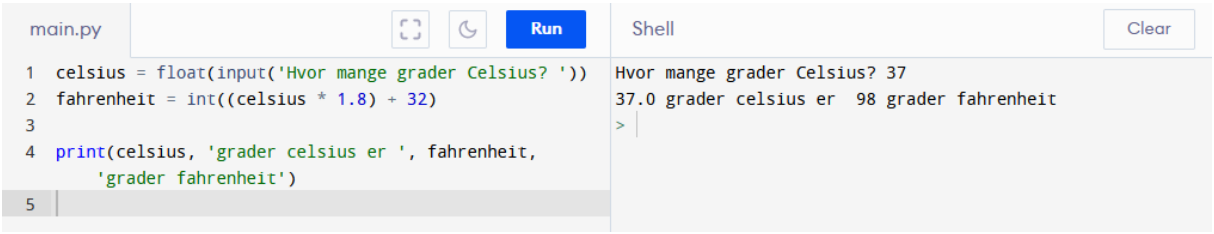
I noen tilfeller når man multipliserer to desimaltall, ender man opp med svar med veldig mange desimaler. Dette ser ikke spesielt pent ut når man skriver det ut:



Figur 5-122: Mange desimaler i output

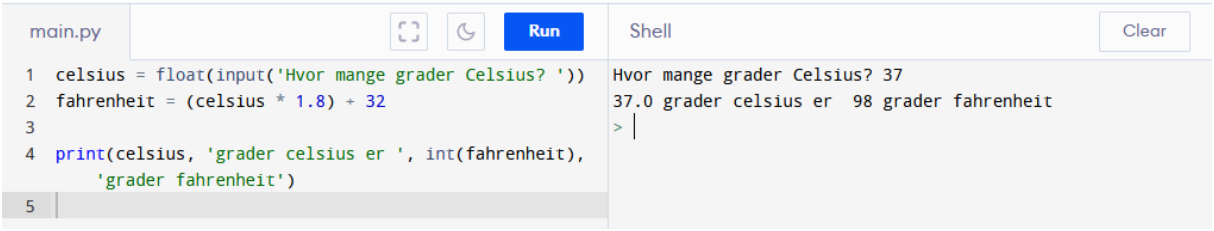
Heldigvis kan man bestemme hvor mange desimaler som skal vises.

Det enkleste er å velge at ingen desimaler skal vises, altså gjøre om desimaltallet (float) til et heltall (integer). Det kan man gjøre enten i variabelen (linje 2):



Figur 5-123: Variabelen settes som int() i en annen variabelen

Eller man kan gjøre det når kan kaller variabelen i print-setningen:



Figur 5-124: Variabelen settes som int() i print()-setningen

Forskjellen er den, at hvis man konvertere desimaltallet direkte i variabelen som på den første måten, så vil fahrenheit-variabelen alltid skrives ut som et heltall. Om man heller endrer variabelen når man kaller på den som på måte nummer 2, så kan man velge å vise desimaltall i noen situasjoner og heltall i andre.

Men som nevnt kan man også bestemme hvor mange desimaler som skal vises, uten å gjøre det om til et heltall. Da skal vi benytte en innebygget funksjon som heter round().

1.1.1.1.1.3 round()

Definisjon:

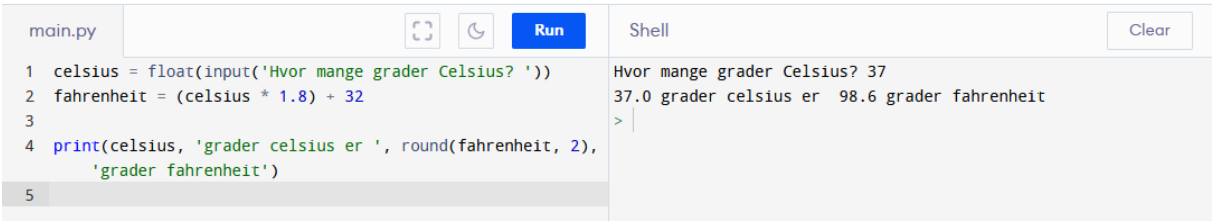
Returnerer et desimaltall som er en avrundet versjon av et spesifisert tall, med et spesifisert antall desimaler.

Syntaks:

```
round(tall, antall sifre)
```

Om man for eksempel skriver round(3.1547, 2), vil utskriften være 3.15. Om man bare skriver round(3.1547), vil Python endre det til et heltall (integer) og skrive ut 3. Standardverdien er altså at sifre er satt til 0.

Man kan også bruke round() både i selve variabelen, og når man kaller på variabelen, slik som kan kunne med int:



Figur 5-125: Bruk av round() for å bestemme antall desimaler

Å bruke round() vil fungere i de aller fleste situasjoner. Legg likevel merke til her at selv om vi har spesifisert at det skal rundes av til to desimaler, så er det bare én desimal som dukker opp i utskriften. Dette er fordi desimal nummer to og tre er tallet 0, og da skrives ikke det ut på skjermen.

I noen situasjoner ønsker man kanskje å skrive ut et visst antall desimaler, selv om den siste desimalen er 0. Python har heldigvis en funksjon for det også:

5.3.3 Funksjoner

En viktig og grunnleggende del av det å skrive kode, er å kunne skrive og bruke funksjoner. En funksjon er en blokk eller gruppering av kode som kun kjører dersom man kaller på funksjonen med et funksjonskall.

Når man kaller på funksjonen, kan man sende data inn til funksjonen kalt argumenter. Funksjonen bruker så argumentene som er sendt til den, kjører de sammen med koden som står i funksjonen, og sender ut igjen et resultat kalt statements.

Dataene man sender inn kalles som sagt argumenter, men inne i funksjonen kalles de parametere. I en og samme funksjon blir parameterne brukt likt hver gang, men verdien parameterne inneholder kan være forskjellige. Parameter er et ord man bruker som "plassholder", slik at funksjonen vet hvoran den skal behandle dataene som kommer inn som argumenter.

Formålet med å bruke funksjoner, er å slippe å skrive den samme koden om igjen mange ganger. Python har også mange innebygde funksjoner du nok allerede har brukt, som print() og input(). Dersom vi ikke hadde hatt disse funksjonene, måtte vi skrevet hele koden inne i for eksempel print()-funksjonen hver gang vi ville skrive noe til skjermen. Så dersom vi i koden vi skriver skal gjøre det samme mange ganger, er det bedre å lage en funksjon vi kan kalle på og benytte flere ganger når vi trenger den.

5.3.3.1 Definisjon og syntaks

1.1.1.1.1.4 function()
Definisjon:

En gruppering av kode som kun kjøres dersom man kaller på den med et funksjonskall.

Syntaks:

```
def function_name(parameters)
      statement(s)
```

5.3.3.2 Definere en funksjon

I Python bruker man ordet **def** for å definere en funksjon, skriver navnet på funksjonen, parenteser og avslutter med et kolon:

```
def min_funksjon():
    print('Tekst fra funksjonen.')
```

```
min_funksjon()
```

Her har man definert en funksjon som heter min_funksjon. Det blir ikke sendt noen argumenter inn til funksjonen. Det kan man se fordi parentesene er tomme. Det eneste denne funksjonen gjør, er å skrive ut setningen 'Tekst fra funksjonen'.

I siste linje kaller man på denne funksjonen ved å skrive navnet på funksjonen med parenteser til slutt: min_funksjon().

5.3.3.3 Argumenter

Dersom man definerer at funksjonen skal ta imot argumenter, må disse også brukes i løpet av funksjonen:

```
def min_funksjon(fag):
    print('I dag skal vi lære ' + fag)


min_funksjon('fysikk')
min_funksjon('kjemi')
min_funksjon('norsk')
```

I dette tilfellet heter argumentet fag. Dette argumentet blir brukt i print()-setningen, sammen med litt tekst. Deretter kaller man på funksjonen tre ganger. Utskriften denne kodebiten blir som følger:

```
I dag skal vi lære fysikk
I dag skal vi lære kjemi
I dag skal vi lære norsk
```

5.3.3.4 Flere argumenter i samme funksjon

Man kan også sende flere argumenter i en og samme funksjon:

```
def min_funksjon(fag, dag):
    print(dag + ' skal vi lære ' + fag)


min_funksjon('fysikk', 'mandag')
min_funksjon('kjemi', 'tirsdag')
min_funksjon('norsk', 'torsdag')
```

Utskriften fra kodesnutten over blir som følger:

```
mandag skal vi lære fysikk

tirsdag skal vi lære kjemi

torsdag skal vi lære norsk
```

Her har vi kalt den første parameteren fag, og den andre parameteren dag. Da er det viktig at vi sender argumentene inn i den rekkefølgen som funksjonen angir. I utskriften bruker vi parameterne i motsatt rekkefølge i dette eksempelet. Når man sender argumentet inn til funksjonen, blir dette midlertidig lagret. Så når vi i første kall sender inn 'fysikk' og 'mandag', tar funksjonen vare på disse verdiene helt til funksjonen er kjørt ferdig. Deretter glemmer funksjonen denne informasjonen, og er klar for at nye argumenter sendes. Derfor er funksjoner en veldig praktisk måte å behandle informasjon på dersom man skal gjøre den samme operasjonen mange ganger.

5.3.3.5 Bruke variabler som argumenter

I noen tilfeller er det bedre å kalle på en funksjon ved hjelp av variabler. Da definerer man først variabelen, og så kan man senere bruke denne som argument:

```python
# Bruke variabler som argumener

navn = 'Per'
fag = 'fysikk'

def my_function(name, course):
    print(name + ' skal undervise i ' + course)

my_function(navn, fag)

```

Figur 5-126: Bruke variabler som argumenter

Her har vi definert to variabler, navn og fag. Parameterne i funksjonen er name og course. Når vi så kaller på funksjonen, sender vi inn variablene navn og fag som argumenter. Navn blir sent inn som argument til parameteren name, og fag blir sendt inn som argument til parameteren course. Utskriften blir som forventet:

```
Per skal undervise i fysikk
```

5.3.3.6 Returnere resultatet fra en funksjon

Til nå har vi hele veien brukt en print()-setning inne i funksjonen, men vi kan også returnere et resultat ved hjelp av kommandoen return:

```
def areal_av_kvadrat(side):
    return side*side
```

Denne funksjonen beregner arealet av et kvadrat, og parameteret i funksjonen er lengden på sidene i kvadratet. Når man kaller på funksjonen, legger man inn lengden man har, for eksempel 4:

```
areal_av_kvadrat(4)
```

Når funksjonen blir kallet på denne måten, skrives det ikke noe ut til konsollen. Det eneste vi har bedt funksjonen om å gjøre, er å regne ut arealet. Vi har ikke bedt den gjøre noe med dette resultatet. For å skrive ut til konsollen, legger vi til print() rundt funksjonskallet:

```
print(areal_av_kvadrat(4))
```

Da kommer output'et 16 i konsollen.

På samme måte som man kan sette sammen variabler og annen tekst i en print()-setning, kan man også gjøre det med funksjoner. Vi skal se på et eksempel der vi ber brukeren skrive inn lengden på siden ved hjelp av input, og deretter bruker en print()-funksjon til å skrive ut hva brukeren skrev inn, men også resultatet av funksjons-kallet:

```
1 # Regne ut arealet av et kvadrat
2
3 lengde = int(input('Hva er lengden av siden på kvadratet? '))
4
5 def areal_av_kvadrat(side):
6     return side*side
7
8 print('Med sidelengde', lengde, 'har kvadratet et areal på', areal_av_kvadrat(lengde))
9
```

Figur 5-127: Regne ut arealet av et kvadrat

Utskriften av denne kodesnutten når vi svarer at lengden er 4 blir som følger:

```
Med lengde 4 har kvadratet et areal på 16
```

5.3.3.7 Globale og lokale variabler

Her har vi kombinert mange av elementene som skal være kjent til nå i en og samme kodesnutt. Først har vi definert to variabler, setning og bokstav, og deretter en funksjon telle_bokstaver() som inneholder både en variabel number, en for-løkke og en if-setning. Her er det veldig tydelig at tabuleringer er viktig, og den viser også forskjellen mellom globale og lokale variabler.

```
# Telle antall av en bestemt bokstav i en setning med en funksjon

setning = "Jeg gikk meg over sjø og land der møtte jeg en gammel mann"
bokstav = 'm'

def telle_bokstaver(sentence, letter):
    number = 0
    for x in sentence:
        if x == letter:
            number += 1
    return number

print('Bokstaven',bokstav, 'finnes', telle_bokstaver(setning, bokstav), 'gang(er) i setningen.')

```

Figur 5-128: 5.3.3.7 Globale og lokale variabler

Variablene setning og bokstav står for seg selv, og kan brukes av absolutt alle kodebiter, funksjoner og annet man har i koden sin. Variabelen number står inne i funksjonen telle_bokstaver(), og kan kun brukes av denne funksjonen. Om vi prøver å skrive print(number) et annet sted i koden, vil vi få denne feilmeldingen:

```
NameError: name 'number' is not defined
```

Det er fordi variabelen number ikke er definert som en global variabel, men en lokal variabel.

Vi har også brukt funksjonskallet i selve print()-setningen: telle_bokstaver(setning, bokstav), sender variablene setning og bokstav inn til funksjonen, og disse tar plassene til sentence og letter inne i selve funksjonen. Funksjonen returnerer tallet 5, og i utskriften er det dét tallet som vil bli skrevet der hvor vi har kallet på funksjonen. Utksriften av denne koden er:

```
Bokstaven m finnes 5 gang(er) i setningen.
```

5.3.4 Lister

Lister er en form for variabler som er veldig praktiske når man vil lagre flere elementer i én og samme variabel. Det finnes tre andre innebygde former for datalagringsmetoder, tupler, set og dictionaries. Vi skal ikke gå gjennom disse her.

I en liste kan man lagre alle datatyper, både tekststrenger, tall, boolske verdier og annet. Man kan også lagre en liste i en liste, såkalte nøstede lister, men de får ikke vi bruk for i denne gjennomgangen.

5.3.4.1 Definisjon og syntaks

1.1.1.1.1.5 list
Definisjon:

En metode for å lagre flere elementer i én variabel.

Syntaks:

```
list = [listeelement 1, listeelement 2, ...]
```

5.3.4.2 Definere en liste

Man definerer en liste ved å skrive et variabelnavn, likhetstegn og så klammeparenteser. I klammeparentesene skriver man hvert listeelement separert av et komma:

Liste med tekststrenger

```
min_liste = ['Hei', 'Hallo', 'Ha det bra']
```

Liste med heltall

```
min_liste = [1, 2, 3, 4, 1]
```

Liste med desimaltall (flyttall)

```
min_liste = [3.14, 5.72, 7.2356, 123.7]
```

Liste med ulike datatyper

```
min_liste = ['Hei', 5.72, 1, True]
```

5.3.4.3 Indeksering og tilgang til elementer

En liste står i en ordnet rekkefølge. Det vil si at rekkefølgen elementene står i spiller enn rolle. Man kan aksessere eller få tilgang til disse elementene ved å bruke kommandoen index (indeks).

Listen starter med indeks 0, som er vanlig i de fleste kodespråk. Det vil si at i listen

['Hei', 5.72, 1, True]

- har index[0] verdien Hei
- har index[1] verdien 5.72

- har index[2] verdien 1
- har index[3] verdien True.

Man kan også bruke motsatt indeksering. I det tilfellet har siste element indeks -1, neste siste har verdi indeks -2 og så videre. Dette kan være hjelpsomt dersom man ikke helt vet hvor lang listen er, men man likevel ønsker å aksessere det siste elementet:

- I listen har index[-4] verdien Hei
- I listen har index[-3] verdien 5.72
- I listen har index[-2] verdien 1
- I listen har index[-1] verdien True.

Det er også mulig å få tilgang til et intervall av listen:

```
min_liste = ['ha det bra', 'adjø', 'på gjensyn', 'vi ses']

print(min_liste[1:3])
```

Utskriften av dette blir:

```
['adjø', 'på gjensyn']
```

Som ellers med indeksering, så skriver man startverdien og **til** verdien hvor man ønsker å avslutte, ikke **til og med**. Dette intervallet vil derfor skrive ut elementene med indeks 1 og indeks 2.

5.3.4.4 Endre, legge til og fjerne elementer

Endre elementer
Vi har en liste elever:

```
elever = ['Per', 'Ola', 'Espen', 'Askepott', 'Snøhvit', 'Elsa']
```

Dersom vi ønsker å endre et element, aksesserer vi det listeelementet og skriver inn den nye verdien:

```
elever[1] = 'Pål'

elever = ['Per', 'Pål', 'Espen', 'Askepott', 'Snøhvit', 'Elsa']
```

Legge til elementer
Dersom vi ønsker å legge til et element til slutt i listen, bruker vi ordet metoden append():

```
elever.append('Anna')

elever = ['Per', 'Pål', 'Espen', 'Askepott', 'Snøhvit', 'Elsa',
'Anna']
```

Vi kan også sette inn elementer der vi ønsker. Da bruker vi metoden insert, og skriver inn hvilken indeks vi ønsker at dette elementet skal ha og verdien til elementet (husk at indekseringen starter på verdien 0):

```
elever.insert(3, 'Mulan')
```

```
elever = ['Per', 'Pål', 'Espen', 'Mulan', 'Askepott', 'Snøhvit',
'Elsa', 'Anna']
```

Fjerne elementer
Dersom vi ønsker å fjerne det siste elementet i en liste, kan vi bruke metoden pop():

```
elever.pop()
```

```
elever = ['Per', 'Pål', 'Espen', 'Mulan', 'Askepott', 'Snøhvit',
'Elsa']
```

Man kan også spesifisere hvilket element man ønsker å fjerne med indeksen til det elementet:

```
elever.pop(4)
```

```
elever = ['Per', 'Pål', 'Espen', 'Mulan', 'Snøhvit', 'Elsa']
```

Eller man kan fjerne et listeelement ved å bruke metoden remove() og skrive inn verdien til elementet man vil gjerne

```
elever.remove('Espen')
```

```
elever = ['Per', 'Pål', 'Mulan', 'Snøhvit', 'Elsa']
```

5.3.4.5 Å bruke lister
Det finnes mange ulike situasjoner det er nyttig å bruke lister.

Man kan med hjelp av en for-løkke regne ut hvor langt en bil kommer seg (strekning) dersom tiden t er den samme for alle verdier, men med ulik fart. Her har vi brukt en for-løkke hvor variabelen v er en liste, og så brukt det samme regnestykket på hvert listeelement i for-løkken. Formelen for å regne ut tiden er som kjent tid = strekning / hastighet:

```python
# Regne ut tid det tar å kjøre en gitt strekning med ulik hastighet

s = 100                 # antall kilomter
v = [20, 40, 50, 80]    # liste over fart i km/t

for fart in v:
    t = s / fart
    print('Med farten', fart, 'km/t tar det', t, 'timer å kjøre', s, 'km.')

```

Figur 5-129: Bruk av lister i en for-løkke

Utskriften blir da:

```
Med farten 20 km/t tar det 5.0 timer å kjøre 100 km.
```

```
Med farten 40 km/t tar det 2.5 timer å kjøre 100 km.

Med farten 50 km/t tar det 2.0 timer å kjøre 100 km.

Med farten 80 km/t tar det 1.25 timer å kjøre 100 km.
```

Man kan også bruke indeksen fra ett listeelement til å finne en verdi i en annen liste. Her har vi en variabel vekt med fødselsvekten til barn. I en annen liste har vi navnene til barna. Ved å loope gjennom listen med en for-løkke, kan vi først finne alle elementene i vekt-listen med en fødselsvekt lik eller større enn 3.5. Deretter kan vi bruke indeksen fra loopen i vekt-listen til å slå opp i navnelisten:

```
1  # Bruke indeksen fra en liste til å gjøre oppslag i en annen liste
2
3  vekt = [2.45, 3.55, 4.22, 3.40, 3.41, 3.12, 3.78]
4  navn = ['Sandra', 'Even', 'Maja', 'Theodor', 'Amanda Marie', 'Sindre', 'Ella']
5
6  for x in vekt:
7      index = vekt.index(x)
8      if x ⩾ 3.5:
9          print(navn[index])
10
```

Figur 5-130: Bruke indeksen fra ett listeelement til å finne en verdi i en annen liste

x i denne koden vil jo være verdiene fra listen vekt (2.45, 3.55, …, 3.78). I linje 7 gjør vi et oppslag i listen vekt, og sier at vi vil vite hvilken indeks denne verdien har. I linje 8 sier vi at dersom x er større enn eller lik 3.5, skal det gjøres en utskrift (linje 8). I print()-setningen i linje 8 sier vi at vi ønsker å skrive ut fra listen navn, og det skal skrives ut den indeksen som tilsvarte en verdi over 3.5. Utskriften fra denne koden blir som følger:

```
Even

Maja

Ella
```

Dersom vi istedenfor å skrive print(navn[index]) hadde skrevet print(index), ville vi fått skrevet ut verdiene 1, 2 og 6. Vi kan se i listen vekt at de indeksene samsvarer med et tall som er over 3.5. Og vi kan se i listen navn at det er navnene Even, Maja og Ella som står i indeks 1, 2 og 6.

5.3.4.6 Ofte brukte listemetoder

Tabell 5-8: Ofte brukte listemetoder

Metode	Forklaring
len()	Brukes for å finne hvor mange elementer som er i listen
max()	Finner den største verdien i listen. På numeriske lister finner den den høyeste verdien, mens i lister med tekst-strenger skriver den ut det siste elementet.
min()	Finner den minste verdien i listen. På numeriske lister finner den den laveste verdien, mens i lister med tekst-strenger skriver den ut det første elementet.

```python
# Finne lenge på liste, samt høyeste og laveste verdi

tall = [78, 34, 5, 2, 43, 2432, 5, 523, 12, 7]

lengde = len(tall)                          # Finner lengden på listen
hoyeste_verdi = max(tall)                   # Finner den høyeste tall-verdien
laveste_verdi = min(tall)                   # Finner den laveste tall-verdien

index_hoyeste = tall.index(hoyeste_verdi)   #Finner indeksen til den høyeste verdien
index_laveste = tall.index(laveste_verdi)   #Finner indeksen til den laveste verdien

print(lengde)                               # Utskrift: 10

print(hoyeste_verdi)                        # Utskrift: 2432
print(laveste_verdi)                        # Utskrift: 2

print(index_hoyeste)                        # Utskrift: 5
print(index_laveste)                        # Utskrift: 3

```

Figur 5-131: Eksempler på ofte brukte listemetoder

5.3.5 Importere og bruke biblioteker

Alt vi har gått gjennom til nå er funksjoner, metoder og annet som er innebygget i Python. I tillegg til dette finnes det mange biblioteker man kan importere og bruke i sin kode, som er utviklet av andre. Vi vil her gå gjennom de mest brukte bibliotekene, som vi kommer til å bruke og vise til i oppgaveheftene.

Man importerer biblioteker ved å skrive import etterfulgt av navnet på biblioteket man vil importere. Man kan importere mange biblioteker i samme kode:

```
import numpy

import math
```

```
import matplotlib.pyplot
```

Hver gang man vil bruke noe fra biblioteket, for eksempel verdien for pi fra math, array fra numpy og muligheten for å plotte en graf fra matplotlib.pyplot, må man skrive biblioteknavnet og så elementet man vil bruke:

```
numpy.array

math.pi

matplotlib.pyplot.plot
```

Det kan for mange biblioteker bli veldig mye å skrive om en skal bruke hele betegnelsen hver gang en bruker noe fra biblioteket, og det er derfor vanlig å importere noen av de med aliaser – et annet navn man kan bruke istedenfor hele navnet. Man velger alias helt selv, men det finnes noen normalt bruke forkortelser. Man angir et alias med kodeordet as:

```
import numpy as np

import math

import matplotlib.pyplot as plt
```

Her har vi importert numpy-biblioteket og kalt det for np og matplitlib.pyplot-biblioteket har vi kalt plt. De samme importene er nå mye kortere å skrive, og med mindre risiko for å skrive feil:

```
np.array

math.pi

plt.plot
```

5.3.5.1 NumPy og arrayer

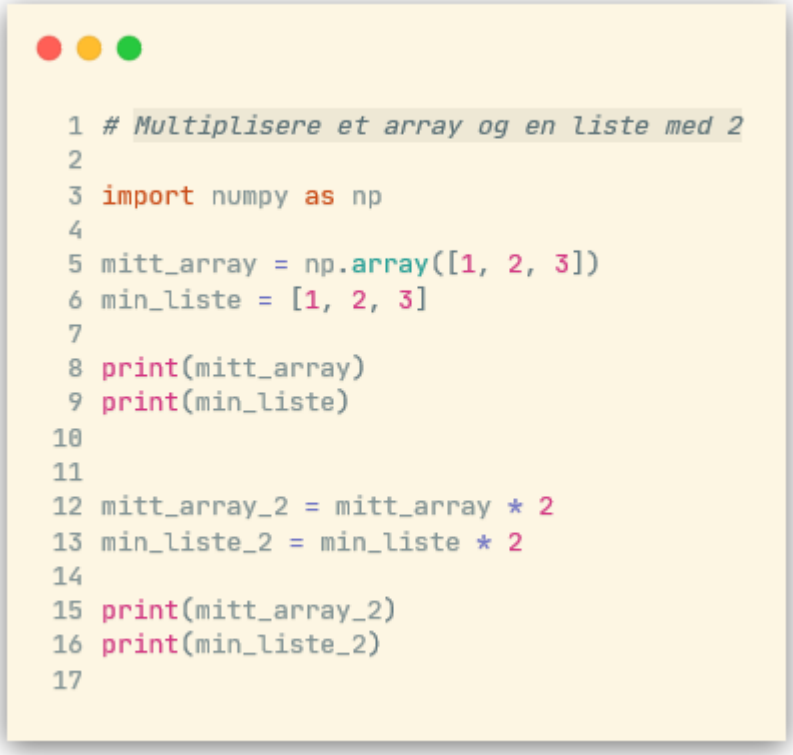
NumPy står for Numerical Python, og det er et bibliotek man blant annet for å bruke arrayer. Arrayer er en avansert form for liste, men med andre bruksområder enn de innebygde listene i Python. De er bedre enn lister til å gjøre numeriske beregninger.

For å opprette et array, må man huske å importere numpy. Deretter lager man en variabel som inneholder et array. Som dere kan se, er det ganske likt som å opprette en liste. Den eneste forskjellen er at vi har satt np.array() rundt klammeparentesene.

```
import numpy as np

min_array = np.array([0, 1, 2, 3])
```

De innebygde Python-listene kan som nevnt inneholde ulike datatyper. Det kan ikke et array, det kan kun bestå av én datatype. Siden arrayer er ment for matematiske beregninger, er det mest fornuftig å holde seg til heltall eller desimaltall, men det er altså mulig å lagre de andre datatypene også. Man kan legge heltall og desimaltall i samme array, men da vil Python tolke hele arrayet som at det inneholde desimaltall.

En vesentlig forskjell mellom lister og array kan illustreres ved å gange de med 2:



```python
# Multiplisere et array og en liste med 2

import numpy as np

mitt_array = np.array([1, 2, 3])
min_liste = [1, 2, 3]

print(mitt_array)
print(min_liste)


mitt_array_2 = mitt_array * 2
min_liste_2 = min_liste * 2

print(mitt_array_2)
print(min_liste_2)

```

Figur 5-132: Forskjell på liste og array

Utskriftene blir dette:

```
[1 2 3]                    # mitt_array

[1, 2, 3]                  # min_liste

[2 4 6]                    # mitt_array_2

[1, 2, 3, 1, 2, 3]         # min_liste_2
```

De to første utskriftene er relativt like, de inneholder samme verdi på samme indeks. Men når man multipliserer en liste med to, blir listen dobbelt så lang. I arrayet multipliseres hver verdi med to, og arrayet er like langt som det var.

Indeksering er det samme som for lister, første element har indeks 0, og siste element kan aksesseres ved å bruke indeks -1.

5.3.5.2 Endre, legge til og fjerne elementer fra et array

Endre elementer
Vi har et array A:

```
A = [1, 2, 3]
```

Dersom vi ønsker å endre et element, aksesserer vi det listeelementet og skriver inn den nye verdien:

```
A[1] = 4

A = [1, 4, 3]
```

Legge til elementer
Dersom vi ønsker å legge til et element til slutt i listen, bruker vi ordet metoden np.append():

```
A = np.append(A, [5])

A = [1, 4, 3, 5]
```

Vi kan også sette inn elementer på den indeksen vi ønsker. Da bruker vi metoden insert, og skriver inn hvilket array vi ønsker at elementet skal legges inn, indeks vi ønsker at dette elementet skal ha og verdien til elementet (husk at indekseringen starter på verdien 0):

```
A = np.insert(A, 2, 7)         # 2 er indeks, 7 er verdien

A = A = [1, 4, 7, 3, 5]
```

Fjerne elementer
Man kan fjerne et element ved å bruke indeksen i arrayet:

```
A = np.delete(A, [0])

A = [4, 7, 3, 5]
```

Som dere kan se, er det å legge til eller å fjerne elementer fra et array litt mer koding enn tilsvarende for lister. Men det å endre et element, er like enkelt for både lister og arrayer. I mange tilfeller kan det derfor være lurt å opprette et array som er så langt som man ønsker, selv om man ikke har verdiene enda, og så heller endre verdiene. Å opprette arrayer kan gjøres på flere forskjellige måter.

5.3.5.3 Flere måter å opprette et array på

Opprette et array med kun verdien 0
Man skriver np.zeros og så antall 0-tall man vil ha:

```
np.zeros(5)
```

```
[0. 0. 0. 0. 0.]              # Utskrift. Verdiene er desimaltall
```

Opprette et array med kun verdien 1
Man skriver np.ones og så antall 1-tall man vil ha:

```
np.ones(5)

[1. 1. 1. 1. 1.]              # Utskrift. Verdiene er desimaltall
```

Opprette et tomt array

```
np.empty(5)

[ 1.   2.   2.5  5.  10. ]    # Utskrift. Verdiene er tilfeldige
                                desimaltall
```

Opprette et array med en range av verdier:

```
np.arange(5)

[0 1 2 3 4]             # Utskrift. Verdiene er heltall
```

Opprette et array med jevnt fordelte intervaller
Man må her definere det første tallet, det siste tallet og størrelsen på trinnet:

```
np.arange(5, 25, 4)

[ 5 9 13 17 21]               # Utskrift. Verdiene er heltall
```

Opprette et array med tall som er lineært fordelt i et spesifisert intervall
Her må man skrive inn det første tallet, det siste tallet og hvor lang arrayet skal være:

```
np.linspace(0, 20, 5)

[ 0.  5. 10. 15. 20.]         # Utskrift. Verdiene er desimaltall
```

I noen tilfeller opprettes et array av typen heltall, og i andre tilfeller desimaltall. Man kan også spesifisere datatypen man ønsker ved å legge til dtype = int eller dtype = float i parentesen. Eksempler på dette:

```
np.arange(5, 25, 4, dtype = float)

[ 5.  9. 13. 17. 21.]         # Utskrift. Verdiene er desimaltall

np.linspace(0, 20, 5, dtype = int)

[ 0  5 10 15 20]              # Utskrift. Verdiene er heltall
```

5.3.5.4 Ofte brukte arraymetdoer

Tabell 5-9: Ofte brukte arraymetdoer

Metode	Forklaring
len()	Brukes for å finne hvor mange elementer som er i listen
np.max()	Finner den største verdien i listen.
np.min()	Finner den minste verdien i listen.
np.where()	Brukes for å finne indeksen i til en verdi i et array
np.mean()	Regner ut gjennomsnittet i arrayet
np.average()	Regner ut gjennomsnittet i arrayet
np.sort()	Lager en kopi av arrayet og sorterer det. Det opprinnelige arrayet har uendret rekkefølge.

```python
# Metoder som ofte blir brukt med arrayer
import numpy as np

tall = np.array([4, 56, 12, 543, 15, 76, 18, 72, 64])

lengde = len(tall)                          # Finner lengden på listen
max_verdi = np.max(tall)                    # Finner den høyeste tall-verdien
min_verdi = np.min(tall)                    # Finner den laveste tall-verdien

max_index = np.where(tall == max_verdi)     # Finner indeksen til den høyeste verdien
min_index = np.where(tall == min_verdi)     # Finner indeksen til den laveste verdien

gjennomsnitt = np.mean(tall)                # Regner ut gjennomsnittet
gjennomsnitt2 = np.average(tall)            # Regner ut gjennomsnittet
median = np.median(tall)                    # Finner medianen

sortert = np.sort(tall)                     # Sorterer listen fra minst til størst

print(lengde)                               # Utskrift: 9

print(max_verdi)                            # Utskrift: 543
print(min_verdi)                            # Utskrift: 4

print(max_index)                            # Utskrift: (array([3], dtype=int64),)
print(min_index)                            # Utskrift: (array([0], dtype=int64),)

print(gjennomsnitt)                         # Utskrift: 95.55555555555556
print(gjennomsnitt2)                        # Utskrift: 95.55555555555556
print(median)                               # Utskrift: 56.0

print(sortert)                              # Utskrift: [  4  12  15  18  56  64  72  76 543]

```

Figur 5-133: Eksempler på ofte brukte arraymetdoer

5.3.5.5 Np.sign()

sign(var) er en funksjon som gjøres tilgjengelig gjennom NumPy biblioteket. Hovedoppgaven til funksjonen er å returnere et tall som indikerer fortegnet på variabelen var som blir sendt inn som argument.

Dersom verdien til var er > 0, returnerer funksjonen tallet 1,

dersom verdien til var = 0, returnerer funksjonen tallet 0,

og dersom verdien til var < 0, returnerer funksjonen tallet -1.

sign()-funksjonen kan eksempelvis være nyttig om en skal regne med vektorer i matematikk eller fysikk, og lurer på hvilken retning de ulike vektorene har, og følgelig hvilket fortegn de har.

5.3.5.6 Matplotlib.pyplot og grafer

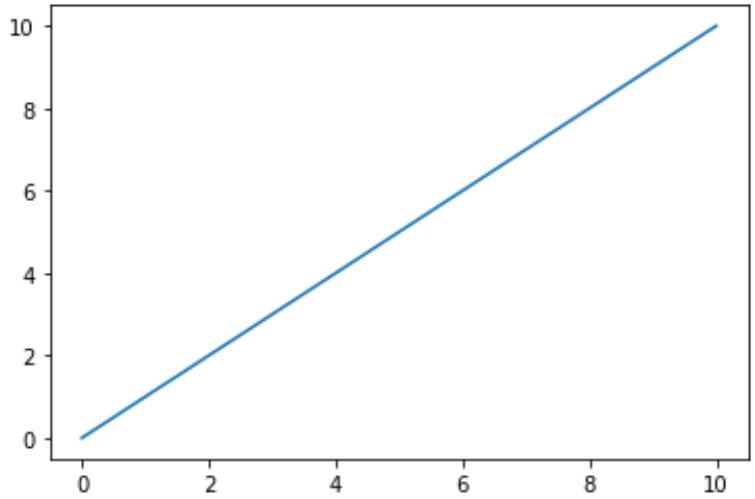
Dette biblioteket brukes i hovedsak for å plotte grafer. Man kan skrive inn legge inn verdier på aksene, aksenavn, fargelegge grafene, velge om det skal vises punkter eller heltrukken linjene og ellers redigere og dekorere som du selv vil. Matplotlib.pyplog brukes mye sammen med numpy arrays, siden arrays er bedre rustet for numeriske beregninger enn Pythons innebygde lister.

Som et enkelt eksempel skal vi vise hvordan vi kan plotte en graf med de to punktene (x, y) = (0, 0) og (x, y) = (10, 10):

```
1  # Plotte linje med punktene (0, 0) og (10, 10)
2
3  import matplotlib.pyplot as plt
4  import numpy as np
5
6  xverdier = np.array([0, 10])
7  yverdier = np.array([0, 10])
8
9  plt.plot(xverdier, yverdier)
10 plt.show
```

Figur 5-134: Plotte punkter i en graf med matplotlib og numpy

Utskriften fra denne koden blir da som følger:



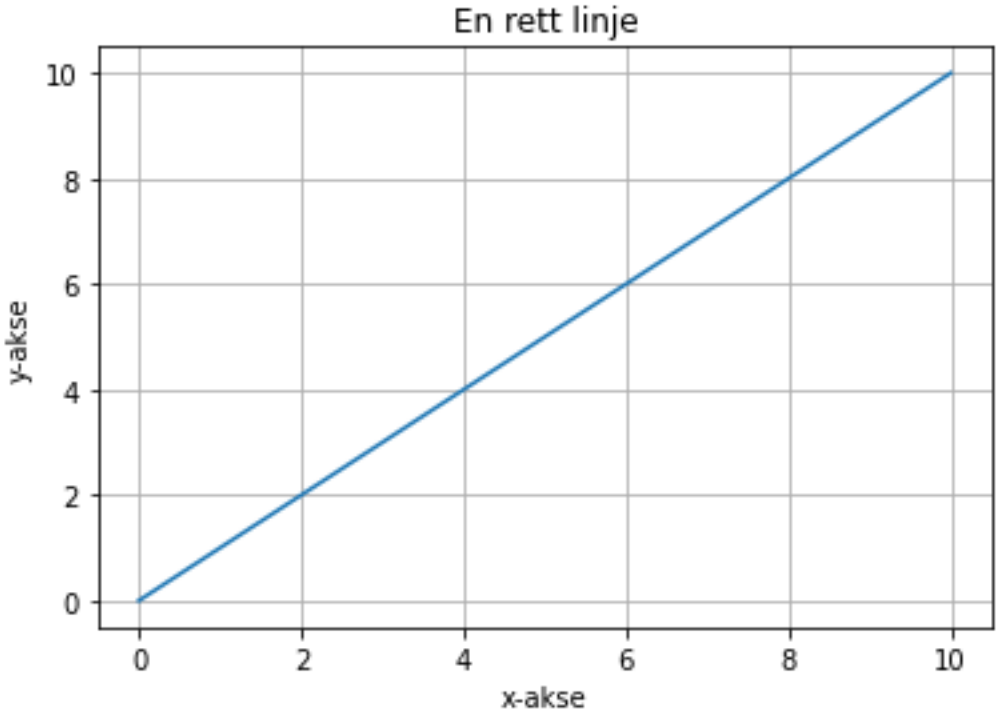
Her er det spesielt viktig å skrive plt.show(). Dersom man ikke skriver plt.show(), vil ikke grafen vises på skjermen. Det kan sammenlignes med print() som vi har jobbet mye med frem til nå. Dersom man ikke skriver plt.show() eller print(), så er kanskje all jobben gjort, men det vises ikke i konsollen. Så lenge vi ikke har bedt programmet om å skrive ut noe/vise noe, så gjør det heller ikke det.

La oss nå se på hvordan denne utskriften kan bli litt penere ved å legge til aksetitler, navn på grafen og et rutenett:

```
1  # Plotte linje med punktene (0, 0) og (10, 10)
2  # og litt styling
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6
7  xverdier = np.array([0, 10])
8  yverdier = np.array([0, 10])
9
10 plt.plot(xverdier, yverdier)
11
12 plt.title("En rett linje")          # Tittel på grafen
13 plt.xlabel("x-akse")                # Tittel på x-akse
14 plt.ylabel("y-akse")                # Tittel på y-akse
15 plt.grid()                          # Legger til rutenett
16
17 plt.show()
```

Figur 5-135: Plotte en graf med aksetitler, navn og et rutenett

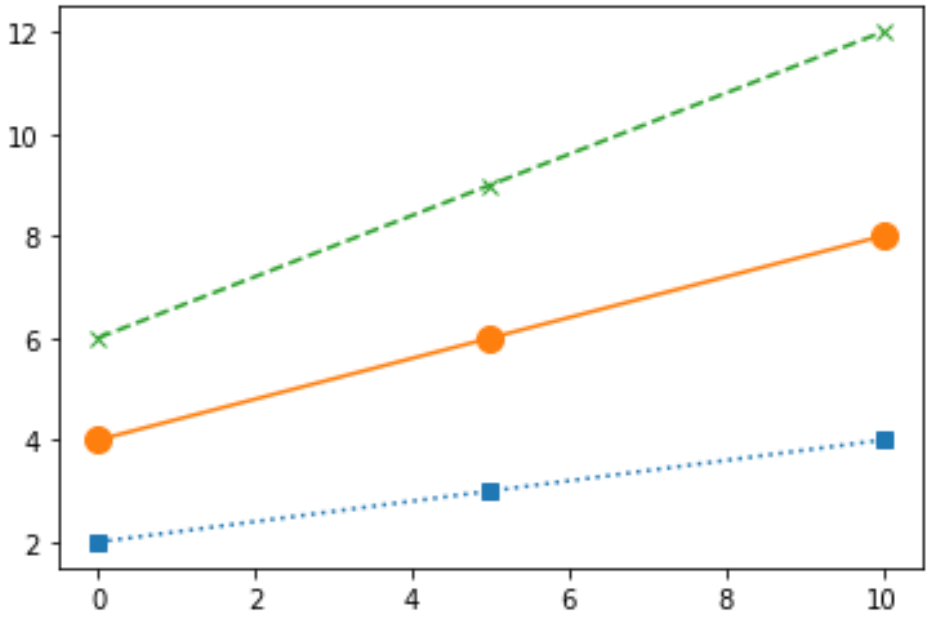
Med disse enkle grepene er grafen litt lettere å tolke, og man ser hva aksene skal representere:



Man kan også style både linjene og punkter med ulike former, størrelser og farger. Her er et par eksempler:

```python
# Style grafene

import numpy as np
import matplotlib.pyplot as plt

# Oppretter array for x-verdier
x = np.array([0, 5, 10])

# Oppretter array for y-verdier.
y1 = np.array([2, 3, 4])

# Med numpy array kan vi gange hele arrayet for å lage nye arrayer
y2 = y1 * 2
y3 = y1 * 3

# Tegner linje 1 med firkanter i punktene, og med prikkete linje
plt.plot(x, y1, marker = 's', linestyle = ':')

# Tegner linje 2 med sirkler i punktene, og forstørrede punkter
plt.plot(x, y2, marker = 'o', markersize = 10)

# Tegner linje 3 med kryss i punktene, og med stiplet linje
plt.plot(x, y3, marker = 'x', linestyle = '--')

# Viser grafen på skjermen
plt.show()

```

Figur 5-136: Style linjer og punkter på grafen



5.3.5.7 PYLAB

PyLab er en modul som samler de mest essensielle verktøyene og funksjonene fra de største og mest brukte bibliotekene som NumPy og matplotlib.pyplot. Modulen ble laget for å forenkle tilgangen og bruken, når en ønsker å skrive kort og konsis kode, som for eksempel når en skal undervise eller lage eksempler, gjerne i interaktiv kode online. Gjennom å importere hele pylab får en tilgang til alle funksjonene som følger med, uten å tenke på alias, eller klassenavn som en vanligvis gjør når en importerer ett og ett bibliotek. PyLab importeres typisk på følgende måte:

```
from pylab import *
```

I data-sammenheng betyr * 'alt', slik at kommandoen over sier «hent alt fra pylab og gjør det tilgjengelig».

5.3.6 Importere og bruke filer

Man kan importere en rekke ulike filer til Python, for eksempel tekstfiler, regneark eller csv-filer. Vi skal her gå gjennom noen av måtene filene kan importeres på.

5.3.6.1 Importere txt-filer med NumPy loadtxt

En txt-fil er et tekstdokument som kun inneholder tekst. Her kan man lagre små eller store mengder data, for så å importere filen til Python og bruke innholdet til ulike formål. I denne gjennomgangen skal vi konsentrere oss om å tegne grafer av innholdet.

NumPy har en funksjon som heter loadtxt, og som kan brukes på denne måten:

```
import numpy as np

data = np.loadtxt("navnet_på_filen")
```

Dersom filen heter temperaturer.txt, og tekst-filen ligger lagret i samme mappe som Python-filen man jobber i, kan man importere den ved å skrive ./ foran filnavnet:

```
import numpy as np

data = np.loadtxt("./temperaturer.txt")
```

./ signaliserer at filen er lagret i den gjeldende katalogen/mappen.

Dersom filen ligger lagret et annet sted, kan man også skrive inn hele filbanen:

```
import numpy as np

data = np.loadtxt("C://Users/Kari/Dokumenter/temperaturer.txt")
```

La oss anta at denne filen inneholder to kolonner med data, høyeste temperatur og laveste temperatur. Hver rad representerer én dag i en måned. Å importere på denne måten forutsetter at dataene er separert av et mellomrom, for eksempel slik:

temperaturer.txt
6 18 4 17 7 21 8 15 … …

Dersom dataene heller var separert med et komma, må man spesifisere det når man importerer filen. Dette gjøres ved å legge til delimiter (skilletegn):

temperaturer.txt
6, 18 4, 17 7, 21 8, 15 … …

```
import numpy as np

data = np.loadtxt("./temperaturer.txt", delimiter = ",")
```

Dersom filen også inneholder tittel på kolonnen i første rad, man kan hoppe over denne ved å bruke skiprows:

temperaturer.txt
Laveste temperatur, høyeste temperatur 6, 18 4, 17 7, 21 8, 15 … …

```
import numpy as np

data = np.loadtxt("./ temperaturer.txt", delimiter = ",", skiprows =
1)
```

Man kan også velge å laste inn én og én kolonne. Om man for eksempel har 4 kolonner, men bare vil ha den tredje kolonnen, skriver man usecols = 2:

```
import numpy as np

data = np.loadtxt("./ temperaturer.txt", delimiter = ",", usecols =
2)
```

Tegne graf over temperaturer

I samme mappe har vi lagret en py-fil og en tekst-fil (Temperaturer Lade mars.txt). Tekstfilen ser slik ut:

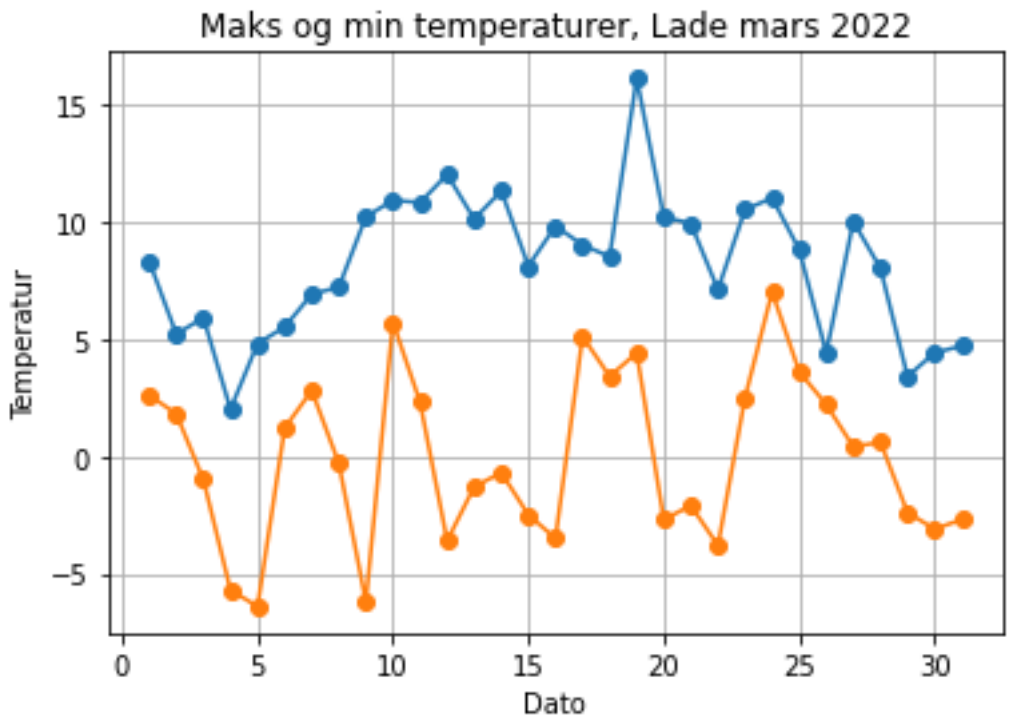
Temperaturer Lade mars.txt
Maksimumstemperatur;Minimumstemperatur
8.3;2.6
5.2;1.8
5.9;-1
2;-5.7
4.8;-6.4
5.5;1.2
6.9;2.8
7.2;-0.3
10.2;-6.2
10.9;5.7
10.8;2.3
12;-3.6
10.1;-1.3
11.3;-0.7
8.1;-2.5
9.8;-3.5
9;5.1
8.5;3.4
16.1;4.4
10.2;-2.7
9.9;-2.1
7.1;-3.8
10.5;2.5
11;7
8.8;3.6
4.4;2.2
10;0.4
8;0.6
3.4;-2.4
4.4;-3.1
4.7;-2.7

Disse temperaturene er lastet ned fra https://seklima.met.no/. Her kan man velge tidsoppløsning, værelementer, tidsrom og værstasjon. Vi valgte å se på temperaturer fordelt på døgn i mars måned 2022, fra værstasjonen på Lade i Trondheim.

Vi skal nå laste inn denne filen og tegne grafer for maksimum- og minimumstemperaturer:

```python
# Grafer over maks- og minimumstemperaturer

import numpy as np
import matplotlib.pyplot as plt

# Det er to kolonner i filen, maks.temp og min.temp.
# I den første raden står det Maksimumstemperatur og Minimumstemperatur
# Aksesserer de to kolonnene og lagrer i hver sin y-liste
ymin = np.loadtxt("./Temperaturer Lade mars.txt", delimiter = ";", usecols = 0, skiprows = 1)
ymax = np.loadtxt("./Temperaturer Lade mars.txt", delimiter = ";", usecols = 1, skiprows = 1)

# For x-verdier lager vi et array med tallene 1 til 31, siden det er 31 dager i mars:
x = np.arange(1, 32, 1)

# Deretter plotter vi grafene
plt.plot(x, ymin, marker = 'o')
plt.plot(x, ymax, marker = 'o')

# Så pynter vi litt på utskriften
plt.title('Maks og min temperaturer, Lade mars 2022')
plt.xlabel('Dato')
plt.ylabel('Temperatur')
plt.grid()

# Skriver grafen ut til skjermen
plt.show()

```

Figur 5-137: Kode for å tegne graf over temperaturer



5.3.6.2 Importere CSV-filer med Pandas

CSV står for "Comma Separated Values", men navnet er litt misvisende. Det er vanlig å bruke komma som skilletegn, men man kan også bruke hvilket som helst annet, som kolon, semikolon, tabulator eller annet. CSV-filer åpnes som regneark.

Å importere med Pandas er relativt likt som å importere med NumPy:

```
import pandas as pd

data = pd.read_csv(r" plassering_av_filen")
```

Det viktige å notere seg her, er at man skriver r foran plasseringen av filen. R står for read; At man skal lese av filen. Det finnes andre kommandoer dersom man skal gjøre andre ting med filen, som w for write (skrive).

Dersom filen heter nedbør.csv, og csv-filen ligger lagret i samme mappe som Python-filen man jobber i, kan man importere den ved å skrive ./ foran filnavnet:

```
import numpy as np

data = pd.read_csv("./nedbør.csv")
```

Dersom filen ligger lagret et annet sted, kan man også skrive inn hele filbanen:

```
import numpy as np

data = pd.read_csv (r"C://Users/Kari/Dokumenter/nedbør.csv")
```

Tegne graf over nedbørsmengde

Igjen har vi brukt https://seklima.met.no/ for å laste ned data. Vi har valgt å se på nedbørsmengde registrert av værstasjonen på Lade i Trondheim, januar 2022. Filen er lastet direkte ned som en CSV-fil, og plassert i samme mappe som Python-filen vi jobber i.

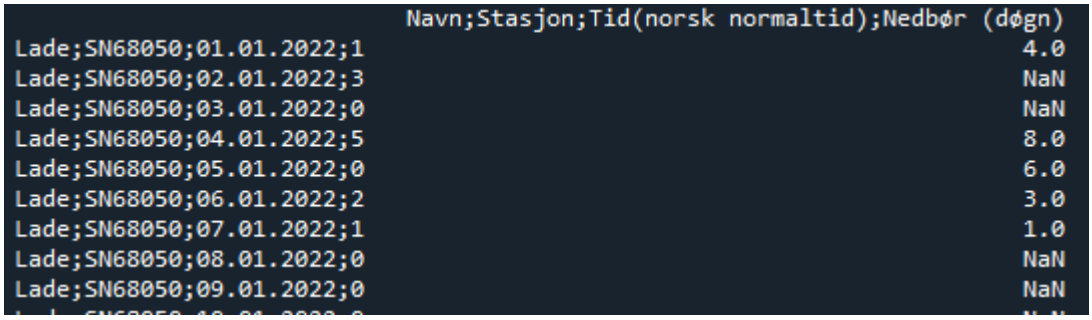
Et utsnitt av CSV-filen ser slik ut:

	A	B	C	D	E
1	Navn	Stasjon	Tid(norsk normaltid)	Nedbør (døgn)	
2	Lade	SN68050	01.01.2022	1,4	
3	Lade	SN68050	02.01.2022	3	
4	Lade	SN68050	03.01.2022	0	
5	Lade	SN68050	04.01.2022	5,8	
6	Lade	SN68050	05.01.2022	0,6	
7	Lade	SN68050	06.01.2022	2,3	
8	Lade	SN68050	07.01.2022	1,1	
9	Lade	SN68050	08.01.2022	0	
10	Lade	SN68050	09.01.2022	0	
11	Lade	SN68050	10.01.2022	0	

Deretter importerer vi filen til Python og skriver ut det som er lastet inn:

```
import pandas as pd

data = pd.read_csv(r"Nedbør Lade januar.csv")

print(data)
```

Et utsnitt av utskriften ser slik ut:



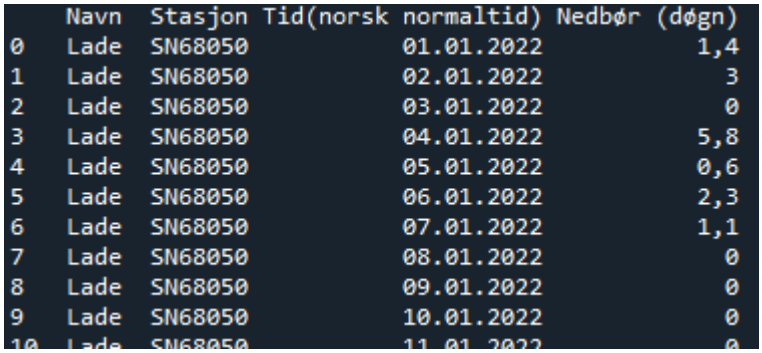
```
                        Navn;Stasjon;Tid(norsk normaltid);Nedbør (døgn)
Lade;SN68050;01.01.2022;1                                           4.0
Lade;SN68050;02.01.2022;3                                           NaN
Lade;SN68050;03.01.2022;0                                           NaN
Lade;SN68050;04.01.2022;5                                           8.0
Lade;SN68050;05.01.2022;0                                           6.0
Lade;SN68050;06.01.2022;2                                           3.0
Lade;SN68050;07.01.2022;1                                           1.0
Lade;SN68050;08.01.2022;0                                           NaN
Lade;SN68050;09.01.2022;0                                           NaN
```

Dette er lastet inn som én enkelt kolonne, noe vi ikke er interessert i. Som skilletegn er det brukt semikolon. Helt til høyre ser dere også at noen verdier er angitt som NaN. Det betyr Not a Number, altså at det ikke er registrert noen verdi.

For å få hentet inn data på ønsket måte, må vi også her angi skilletegn når man laster inn filen. Man kan bruke delimiter som man gjorde med NumPy, men man kan også bruke forkortelsen sep (separator):

```
import pandas as pd

data = pd.read_csv(r"Nedbør Lade januar.csv", sep = ';')

print(data)
```

Utskriften ser nå bedre ut:



	Navn	Stasjon	Tid(norsk normaltid)	Nedbør (døgn)
0	Lade	SN68050	01.01.2022	1,4
1	Lade	SN68050	02.01.2022	3
2	Lade	SN68050	03.01.2022	0
3	Lade	SN68050	04.01.2022	5,8
4	Lade	SN68050	05.01.2022	0,6
5	Lade	SN68050	06.01.2022	2,3
6	Lade	SN68050	07.01.2022	1,1
7	Lade	SN68050	08.01.2022	0
8	Lade	SN68050	09.01.2022	0
9	Lade	SN68050	10.01.2022	0

NaN-verdiene er også erstattet med verdien 0, som er bedre når vi skal bruke kolonnen nedbør til å tegne en graf. I tillegg ser dere at det har dukket opp en indekserings-kolonne helt til venstre. Den starter på 0 fra og med linje 2, så den anser den første kolonnen i filen for å være kolonneoverskrifter. Dersom man ikke har kolonneoverskrifter, må man angi dette når man laster inn filen med header = None. Det ønsker ikke vi i vårt tilfelle.

Det vi derimot ønsker, er å laste inn kun den siste kolonnen. Den heter "Nedbør (døgn)". Dersom vi skriver inn usecols og det nøyaktige navnet til kolonnen i klammeparentesr, får vi lastet inn kun den:

```
import pandas as pd

data = pd.read_csv(r"Nedbør Lade januar.csv", sep = ';', usecols =
["Nedbør (døgn)"])

print(data)
```

```
   Nedbør (døgn)
0            1,4
1              3
2              0
3            5,8
4            0,6
5            2,3
6            1,1
7              0
```

Nedbørsmengden er skrevet inn som desimaltall med komma. I Python må man bruke punktum i desimaltall. Også dette kan angis når man laster inn dataene, ved å bruke ordet decimal:

```
import pandas as pd

data = pd.read_csv(r"Nedbør Lade januar.csv", sep = ';', decimal =
',', usecols = ["Nedbør (døgn)"])

print(data)
```

```
    Nedbør (døgn)
0             1.4
1             3.0
2             0.0
3             5.8
4             0.6
5             2.3
6             1.1
7             0.0
```

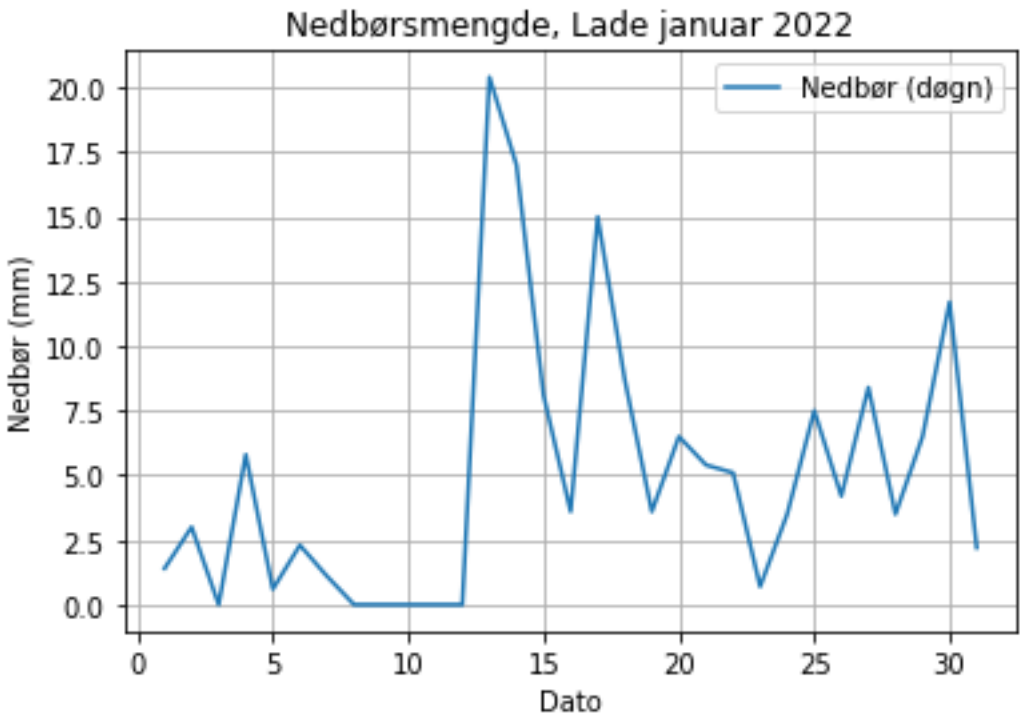
Nå har vi den korrekte nedbørsmengden, og vi har indeksen som starer på null. Dersom indeksen hadde vært ett nummer større, ville det ha tilsvart datoen i måneden. Derfor inkrementerer vi indeksen med 1:

```
data.index += 1
```

Da har vi alt vi trenger for å plotte grafen! Med pandas trenger vi da bare å skrive data.plot(), og en graf. Vi velger også å legge til aksetitler, graftittel og et rutenett:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Laster inn filen
data = pd.read_csv(r"Nedbør Lade januar.csv", sep = ';',
                   decimal = ',', usecols = ["Nedbør (døgn)"])

# Inkrementer indeks med 1
data.index += 1

# Plott grafen
data.plot()

# Style grafen
plt.title('Nedbørsmengde, Lade januar 2022')
plt.xlabel('Dato')
plt.ylabel('Nedbør (mm)')
plt.grid()

# Skriv grafen til skjerm
plt.show()

```

Figur 5-138: Kode for å tegne graf av nedbørsmengde



5.3.7 F-string

f-Strings kalles gjerne en streng-formaterer, og tilbyr en del ekstra funksjonalitet som lar en utforme utskrift av en String i Python på en mer kompleks måte enn ved kun å benytte eksisterende streng-funksjonalitet.

f-String funksjonen tillater for eksempel en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier, slik at koden blir mer leselig og håndterbar for koderen. Ved å skrive inn {code} direkte i selve strengen, får en beregnet, eller hentet resultatet av koden code, som blir plassert i stedet for {code} i tekststrengen når den skal skrives ut. Code fungerer her som en placeholder for kode som har en verdi. Enten fra en variabel, eller som retur fra en funksjon. Så når tekststrengen skal skrives ut, erstattes placeholder code med den aktuelle verdien for uttrykket som code representerer.

Eksempel med variabel:

```
code = 5
print (f'Tallet i dag er {code}!')
```

Print-setningen vil da skrive ut «Tallet i dag er 5!»

Eksempel med funksjon:

```
code = 5
def double_me(tall):
      return 2* tall

print (f'Tallet i dag er {double_me(code)}!')
```

Print-setningen vil da skrive ut «Tallet i dag er 10!»

5.4 Oppgavehefte i Python for VG1

5.4.1 Oppgave 1 - Strålingsintensitet

Inspirasjon til denne oppgaven er hentet fra Naturfag SF (Brandt, Hushovd, & Tellefsen, 2020, ss. 200-201).

5.4.1.1 Kort beskrivelse av tema

For å beskrive energien fra stråling benyttes uttrykket intensitet. Intensiteten til strålingen er et mål på hvor mye energi som passerer et areal per tidsenhet (sekund), eller mer presist, effekt per areal.

Lysintensitet minker med kvadratet av avstanden til kilden. Det betyr at dersom en flate befinner seg én meter fra en lyskilde, og en så flytter flaten enda én meter unna (dobler avstanden) vil lysintensiteten på flaten da være en fjerdedel. Lysintensiteten blir redusert med $1 / r^2$, der r er avstanden fra lyskilden.

5.4.1.2 Oppgave:

Basestasjoner som benyttes av mobiltelefoni fungerer på tilsvarende måte, og følger de samme fysiske lovene. En utendørs basestasjon sender med en effekt på opptil 100 W. Lag en graf som viser strålingsintensiteten ved ulike avstander fra basestasjonen.

5.4.1.3 Problemløsning/Problemløsnings-strategi:

For å tegne en graf som skal illustrere strålingsintensiteten, trengs verdier langs x-aksen og y-aksen. Det er naturlig å velge de ulike avstandene til kilden som x-verdier, og strålingsintensiteten langs y-verdier. X-verdiene kan genereres automatisk ved hjelp av en Python funksjon som linspace() eller range(), mens y-verdien må beregnes basert på de ulike x-verdiene ved hjelp av formelen

$$y = 1 / x^2$$

I denne løsningen benyttes linspace() for å opprette en rekke med x-verdier. linspace() er en funksjon som gjøres tilgjengelig for Python-programmet gjennom at biblioteket pylab importeres.

5.4.1.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere strålingsintensiteten ved ulike avstander til basestasjonen, og i tillegg lar en bruke funksjonen linspace() i koden.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. effekt = 100, for effekten som er oppgitt i W ved 1 m

b. x = linspace(1 , 15), for en liste x over x-verdier fra 1-15

linspace(start, stopp, n)

er en funksjon som tilbys av numpy-biblioteket (som er tilgengelig via pylab). Funksjonen tar et intervall fra start til stopp og deler inn i n like deler. I dette eksempelet opprettes en variabel som heter x, som så får tildelt en liste med ulike verdier nemlig alle verdiene x E [1,15]. Det betyr at variabelen x inneholder en liste med følgende verdier x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

c. y = 1 / x**2 , for samling av alle y-verdiene som blir kalkulert basert på de x-verdiene som nettopp ble opprettet i forrige steg b).

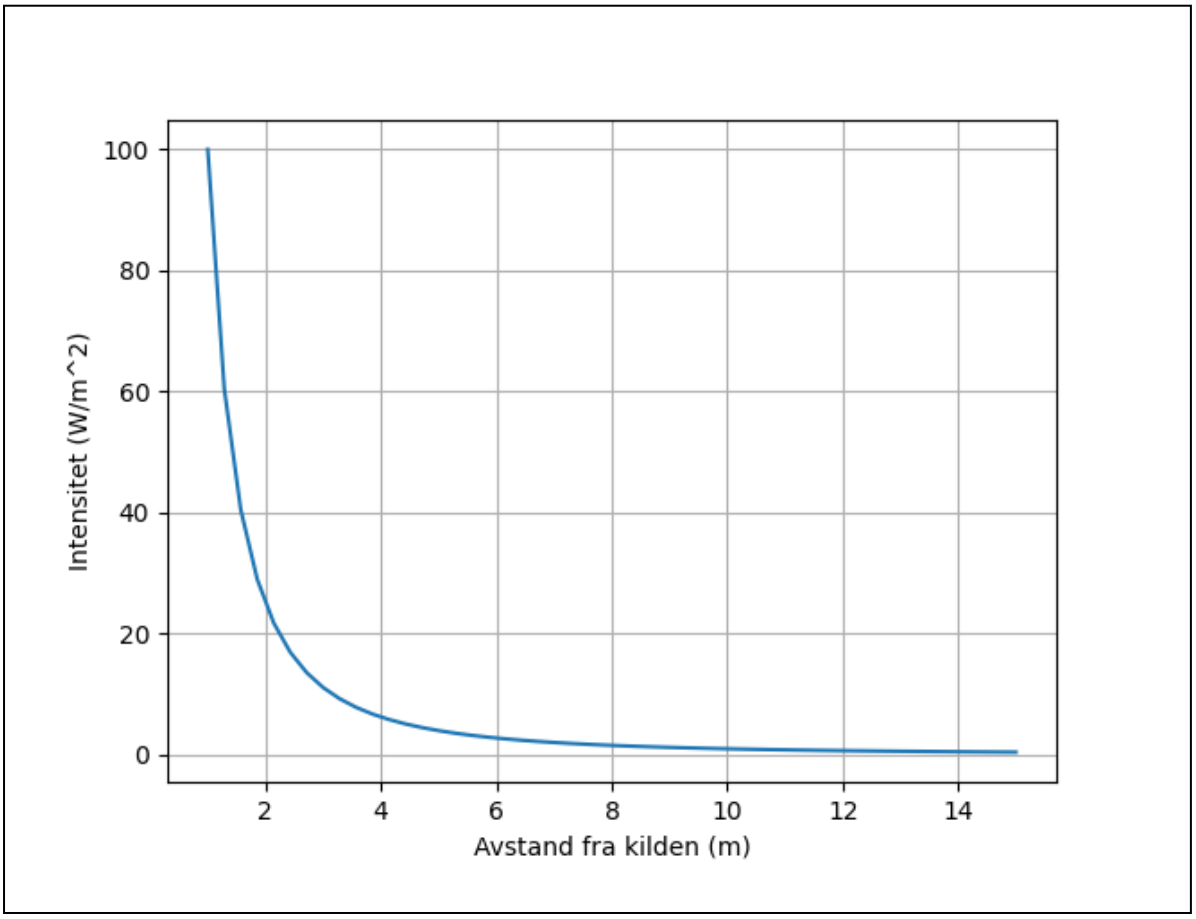
3) Plotter de registrerte verdien i x og y som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pyplot-biblioteket
   a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
   b. xlabel(«Streng»), setter beskrivende tekst til x-aksen
   c. ylabel(«Streng»), setter beskrivende tekst til y-aksen
   d. grid(), tegner inn rutenett på figuren
   e. show(), funksjon som tegner grafen til skjerm

5.4.1.5 Program:

```
# Strålingsintensitet

from pylab import *

effekt = 100                        # effekt i W, 1m fra masten
x = linspace(1, 15)                 # lager x-verdier fra 1-15
y = 1/x**2 * effekt                 # Regner ut y-verdier

plot(x, y)                          # lager grafen
xlabel("Avstand fra kilden (m)")    # tittel langs x-aksen
ylabel("Intensitet (W/m^2)")        # tittel langs y-aksen
grid()                              # tegner på et rutenett
show()                              # viser grafen
```

Figur 5-139: Kode for oppgave 1 - Strålingsintensitet

5.4.1.6 Output:


100
80
60
40
20
0
Intensitet (W/m^2)
2
4
6
8
10
12
14
Avstand fra kilden (m)

5.4.2 Oppgave 2 – Bølgelengde

Inspirasjon til denne oppgaven er hentet fra Naturfag SF (Brandt, Hushovd, & Tellefsen, 2020, s. 302).

5.4.2.1 Kort beskrivelse av tema

Når vi benytter mobiltelefoner eller andre apparater som overfører signaler trådløst, må de elektriske signalene fra apparatet omformes til elektromagnetisk stråling som kan sendes gjennom luften. Signalene blir omgjort til stråling ved hjelp av antenner i sendere og mottakere. Størrelsen på antennene avgjør hvilke frekvenser som sendes ut og kan mottas. I små mobiltelefoner og utstyr som BT (Bluetooth) høretelefoner, er antennene meget små, og frekvensene på strålingen som antennen overfører, tilsvarende høye. BT sender typisk på 2,4 GHz.

Elektromagnetisk stråling beveger seg med lysfarten c, og sammenhengen er

$$c = f * \lambda,$$

der c er den konstante lysfarten ($3 * 10^8$ m/s i vakuum), f er frekvensen, og λ (lambda) er bølgelengden

5.4.2.2 Oppgave:

Du benytter en wifi-ruter på skolen som overfører data trådløst med en frekvens på 2.4GHz. Finn bølgelengden, og kvart-bølgelengden til strålingen.

5.4.2.3 Problemløsning/Problemløsnings-strategi:

For å beregne bølgelengden må formelen for lysfart benyttes. I den inngår tre størrelser, lysfarten c, bølgelengden λ og frekvensen f. To av disse har kjente verdier allerede, og disse lagres som variabler slik at de kan benyttes i beregningen av bølgelengden bl i koden.

5.4.2.4 Algoritme:

1) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. $c = 3 * 10^8$, for lysfarten i m/s i vakuum
    b. $f = 2.4 * 10^9$, for frekvensen
    c. bl = c / f, for beregning av bølgelengden bl.

2) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren.

```
print(«Bølgelengden er», bl, «m»)

print(«Kvartbølgelengden er», bl/4, «m»)
```

5.4.2.5 Program:

```python
# Bølgelengde

c = 3e8          # Lysfarten gitt i m/s
f = 2.4e9        # Frekvens gitt i 1/s (1GHz = 1e9 1/s)

# Utregning av bølgelengde
bl = c / f       # Bølgelengde er lysfart delt på frekvens

# Skriver ut resultatet av utregningen
print("Bølgelengden er", bl, "m")
print("Kvartbølgelengden er", bl/4 * 100, "cm")
```

Figur 5-140: Kode for oppgave 2 – Bølgelengde

5.4.2.6 Output:

```
Bølgelengden er 0.125 m

Kvartbølgelengden er 3.125 m
```

5.4.3 Oppgave 3 – Arv, proteinsyntesen

Inspirasjon til denne oppgaven er hentet fra Naturfag SF (Brandt, Hushovd, & Tellefsen, 2020, s. 282).

5.4.3.1 Kort beskrivelse av tema

Når cellene våre lager nye protein-strukturer, er disse basert på en oppskrift som ligger i genene våre. Proteinene blir satt sammen av ulike organiske syrer, som kalles aminosyrer. For det meste kan kroppen vår produsere de fleste aminosyrene vi trenger, men noen av de må tilføres gjennom kostholdet. I dagligtale kalles disse gjerne essensielle aminosyrer. Cellene i kroppen vår inneholder DNA-molekylene hvor oppskriften på hvordan de ulike proteinene skal lages, er lagret. I løpet av proteinsyntesen blir informasjonen som er lagret i DNA-et skrevet om – transkribert, og en RNA blir laget.

5.4.3.2 Oppgave:

Lag et program som transkriberer (omskriver) en DNA-sekvens i form av en tekststreng, til en RNA-sekvens, og skriver ut resultatet til skjermen.

5.4.3.3 Problemløsning/Problemløsnings-strategi:

Utgangspunktet for oppgaven er en tekststreng (DNA-sekvens) som består av en lang rekke kombinasjoner av bokstavene A, G, C, T, som hver representerer de ulike syrene A, G, C, T, som kan brukes for å bygge ulike protein. Rekkefølgen av disse bokstavene (syrene) bestemmer egenskapene til proteinet som lages basert på oppskriften som DNA-sekvensen følger.

Det opprettes en variabel dna, hvor denne DNA-sekvensen lagres.

For hver eneste bokstav som er lagret i DNA-sekvensen, skal det finnes en tilsvarende RNA-base som det byttes til under byggingen av proteinet. En A i DNA-sekvensen, byttes alltid til en U i RNA-basen. På tilsvarende måte byttes de andre syrene fra DNA sekvensen til følgende baser i RNA-basen etter følgende regel:

Tabell 5-10: DNA og tilsvarende RNA-verdi

DNA	RNA
A	U
T	A
C	G
G	C

Denne gjentakende prosessen utføres av en for-løkke som tar for seg hvert enkelt element (bokstav) i DNA-sekvensen, og behandler den hver for seg.

```
for syre in dna:
```

der syre er en lokal variabel som opprettes kun for å benyttes sammen med for-løkken. Variabelen syre vil få tildelt en verdi for hver gjennomkjøring, som er en bokstav (syre) fra DNA-sekvensen. Når for-løkken har gått igjennom alle elementene i DNA-sekvensen, avsluttes for-løkken, og programmet går videre med resten av koden.

Siden denne utbyttingen repeteres for hver eneste bokstav i tekststrengen dna, definerer vi en funksjon dna_til_rna(dna) som har til oppgave å ta imot en bokstav som står for en syre, og returnere den tilsvarende basen.

```
def dna_til_rna(dna):
```

Funksjonen består av 4 ulike if-setninger, som sjekker om variabelen dna som kommer inn til funksjonen når den blir kalt, er en av 4 muligheter; A, T, C, eller G. For hver av de 4 nevnte mulighetene, returneres den tilhørende basen U, A, G eller C (som beskrevet i tabellen over). Eksempelvis:

```
if dna == 'A': return 'A'
```

Som tilsier at dersom verdien som sendes inn til funksjonen via variabelen dna er 'A', så sendes verdien 'U' tilbake.

Inne i for-løkken opprettes det en lokal variabel rna som tar vare på de alle de nye basene etter hvert som de blir returnert fra funksjonen dna_til_rna

Etter at alle syrene i dna-sekvensen er gjennomgått og transkribert til rna-baser, avsluttes for-løkken.

Til slutt skrives den nylagede rna-sekvensen ut til skjermen ved hjelp av print()-funksjonen

```
print(rna)
```

5.4.3.4 Algoritme:

1) Definerer variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. dna= «AGCCCTCCA … », for en DNA-sekvens
    b. rna = «», for en RNA-sekvens (som er tom)

2) Definerer en funksjon dna_til_rna(dna) som tar imot en verdi dna som representerer en syre, og returnerer en base som korresponderer til den oppgitte verdien av dna

3) Definerer en for-løkke som kjører for alle bokstaver syre som inngår i variabelen dna (Det vil si hele strengen med DNA-sekvensen).

   ```
   for syre in dna:
   ```

   der syre er den lokale variabelen som holder rede på hvor langt en er kommet i gjennomløpet av alle elementene i DNA-sekvensen

    a. Oppretter lokal variabel base som kaller funksjonen dna_til_rna(syre) og sender inn den syren som skal konverters til base. Basen som returneres fra funksjonen lagres så i variabelen base.

b. Variabelen rna oppdateres ved at den nye verdien som ble lagt i variabelen base, nå legges til rna

```
rna = rna + base
```

4) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren.

```
print(rna)
```

5.4.3.5 Program:

```
1  # Proteinsyntese
2
3  # Lager en tekst som inneholder en DNA-sekvens
4  dna = "AGCCCTCCAGGACAGGCTGCATCAGAAGAGGCCATCAAGCAGGTCTGTTCCAAGG"
5
6  # Lager en tom tekstvariabel som kan inneholde RNA-sekvensen
7  rna = ""
8
9  # Lager en funksjon dna_til_rna som gjør om en gitt syre til en base
10 def dna_til_rna(dna):
11     if dna == 'A': return 'U'
12     if dna == 'T': return 'A'
13     if dna == 'C': return 'G'
14     if dna == 'G': return 'C'
15
16
17 # Bruker en løkke for å gjennom hele DNA-sekvensen i variabelen 'dna'
18 # og bytter om hver DNA-syre med den RNA-basen som samsvarer.
19 for syre in dna:
20     # henter basen som korresponderer med syren
21     base = dna_til_rna(syre)
22     # Legger til den nye basen i rna-variabelen
23     rna = rna + base
24
25 # skriver ut den nylagde RNA-sekvensen
26 print(rna)
```

Figur 5-141: Kode for oppgave 3 – Arv, proteinsyntesen

5.4.3.6 Output:

```
UCGGGAGGUCCUGUCCGACGUACUCUUCUCCGGAGUUCGUCCAGACAAGGUUCC
```

5.4.4 Oppgave 4 – Kjønn

Inspirasjon til denne oppgaven er hentet fra Naturfag SF (Brandt, Hushovd, & Tellefsen, 2020, s. 302).

5.4.4.1 Kort beskrivelse av tema

Genene våre regnes gjerne som «en oppskrift på protein». Det er genene som bestemmer hvordan kroppen kommer til å utvikle seg, og hvordan den enkelte bestanddel bygges fra bunnen av. Det er disse proteinene som til slutt gir den enkelte av oss de karakteristika som vi finner igjen som hudfarge, øyefarge, høyde, og en stor mengde andre egenskaper.

De delene som bestemmer hvilket genetisk kjønn et barn har, ligger lagret i det ene av de 23 kromosomparene vi har. Dette kromosomparet kalles gjerne kjønns-kromosomet. Kvinner har to like kromosom-deler i dette kjønnskromosomet som kalles x-kromosom (XX), mens menn har ett x-kromosom og ett y-kromosom (XY). Denne ulikheten er med på å påvirke den statistiske sjansen for at et barn har genetiske markører for mann eller kvinne.

5.4.4.2 Oppgave:

Av 10.000 fødsler vil det i gjennomsnitt bli født 4.854 jenter, og 5.146 gutter. Lag et program som ber brukeren skrive inn et heltall for et antall barn en ønsker å bestemme kjønn på. Disse blir da tilfeldig valgt basert på de statistiske verdiene for å få et jentebarn eller et guttebarn. Hold rede på antallet jenter og gutter som trekkes underveis, og presenter resultatet til skjermen med passende ledetekst.

5.4.4.3 Problemløsning/Problemløsnings-strategi:

For å kunne presentere antallet jenter og gutter som trekkes underveis til brukeren til slutt i programmet, trengs det to variabler som holder rede på henholdsvis antall jenter (ant_jenter) og gutter (ant_gutter). Ved start er disse satt til 0.

Brukeren skal få et spørsmål om hvor mange barn en ønsker å trekke, og dette gjøres i en input() funksjon

```
input(«Hvor mange barn vil du bestille?»)
```

Teksten i input-parentesen blir skrevet ut til skjermen, og programmet venter til brukeren har skrevet inn noe og trykket <Enter>. Svaret som brukeren har skrevet inn blir registrert som en tekststreng (String). Siden vi ønsker å regne med heltall, benytter vi funksjonen int() for å omforme tekststrengen til et heltall (int = integer = heltall).

```
int(input(«Hvor mange barn vil du bestille?»))
```

Nå er svaret fra brukeren gjort om til et heltall som er lettere å benytte i programkoden. For også å kunne bruke tallet brukeren skrev inn, lagrer vi det i en lokal variabel, ant_barn.

Fremgangsmåten for å trekke kjønnet til et enkelt barn er likt for alle barna som skal trekkes, og denne gjentakelsen utnytter vi ved å skrive en for-løkke. Denne for-løkken kjører da den samme koden for hvert enkelt barn brukeren har valgt. For-løkkens oppgave er å foreta en trekning av enten en jente eller en gutt. Vi har fått oppgitt at det av 10.000 fødsler er 4.854 jenter og 5.146 gutter, og vi benytter det som utgangspunkt for vår tilfeldige trekning. Dersom vi trekker et tall mellom 1 og 10.000 må vi finne en måte å avgjøre om det skal regnes som en jente eller gutt. Ved å benytte en tilfeldig-funksjon randint() er det like stor sjanse for å trekke alle de 10.000 ulike numrene ved hver trekning. Da kan vi velge at alle tall fra 1 til 5.146 representerer en gutt, og de resterende (5.147 til 10.000) representerer jenter.

```
randint(fra, til)
```

er en funksjon som da velger et tilfeldig heltall i området [fra, til> .

For hver gjentakelse av for-løkken blir det trukket en tilfeldig verdi, som representerer en jente eller gutt, og for å holde rede på antallet, må vi øke verdien av enten variabelen ant_gutter, eller ant_jenter. For å avgjøre om det tilfeldige tallet som trekkes er en jente eller gutt, brukes vi en valg-setning som kalles if-else.

```
if (betingelse):
      gjør noe i koden
else:
      gjør noe annet
```

Prinsippet er da at om det tilfeldige tallet som trekkes er mindre eller lik 5146, så representerer det en gutt, og variabelen ant_gutter økes med 1. Ellers (det vil si at det tilfeldige tallet er fra 5147 eller høyere), så representerer tallet en jente, og variabelen ant_jenter økes med 1.

Til slutt har for-løkken gjentatt prosessen så mange ganger som brukeren skrev inn, og antallet jenter og gutter er klart og kan presenteres i en print() setning som viser resultatet på skjermen

```
print("Du fikk, ant_jenter, «jenter, og», ant_gutter, «gutter.")
```

5.4.4.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å benytte funksjonen randint() i koden.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. ant_gutter = 0, for antall gutter trukket
   b. ant_jenter = 0, for antall jenter trukket
   c. ant_barn = int(input()), for antallet barn brukeren ønsker å trekke kjønn til

3) Definere for-løkke som kjører for alle definerte verdier av i. (Det vil si for alle de barnene brukeren valgte i sin inntasting). For å beskrive antallet iterasjoner (gjentakelser) av for-løkken benyttes formuleringen

```
for i in range(ant_barn),
```

der i er en lokal variabel som kun opprettes for bruk i for-løkken, og som har som oppgave å holde rede på hvilken iterasjon en er kommet til. Altså hvilket nummer en er kommet til av alle de gangene for-løkken skal gjentas i dette tilfellet, som kommer an på hvilket tall brukeren skrev inn.

Funksjonen range() benyttes for nettopp å beskrive hvor mange ganger løkken skal gjentas. Verdien i range(sluttverdi) gir øvre grense for verdiene som skal benyttes. Ved første gjennomgang blir variabelen i satt til 0, og programkoden inne i for-løkken kjøres helt til i har økt fra i = 0 til i = sluttverdi. Da slutter gjentakelsene av koden inne i for-løkken, og programmet går videre til koden som kommer etter for-løkken.

for-løkken sørger for tilfeldig trekning av kjønn for barn nummer i

```
randint(1, 10001)
```

som gir en verdi fra og med 1 til og med 10.000.

Verdien lagres i en lokal variabel som kun eksisterer inne i for-løkken

```
fødsel = randint(1, 10001)
```

Ved å benytte en if-else blokk, avgjøres det hvilket kjønn tallet som blir lagret i variabelen fødsel representerer

```
if fødsel <= 5146:
      ant_gutter = ant_gutter + 1
else:
      ant_jenter = ant_jenter + 1
```

4) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren.

```
print("Du fikk», ant_jenter, "jenter, og", ant_gutter,
«gutter.")
```

5.4.4.5 Program:

```python
# Kjønn, barnefødsler

from pylab import *

# variabler som holder rede på antall gutter og jenter
ant_gutter = 0
ant_jenter = 0

# Ber brukeren skrive inn et antall barn
ant_barn = int(input("Hvor mange barn vil du bestille? "))

# Bruker en løkke for å tilfeldig bestemme kjønn for hvert barn
for i in range(ant_barn):
    # av 10.000 barn blir statistisk sett 5146 gutter og 4854 jenter
    fødsel = randint(1, 10001)

    if fødsel ⩽ 5146:
        # Det ble en gutt, og ant_gutter økes med 1
        ant_gutter = ant_gutter + 1
    else:
        # det ble en jente, og ant_jenter økes med 1
        ant_jenter = ant_jenter + 1

# skriver ut slutt-resultatet
print(" Du fikk", ant_jenter, "jenter og", ant_gutter, "gutter.")
```

Figur 5-142: Kode for oppgave 4 - Kjønn

5.4.4.6 Output:

```
Hvor mange barn vil du bestille? 10000

 Du fikk 4925 jenter og 5075 gutter
```

5.4.5 Oppgave 5 – Bakterievekst

Inspirasjon til denne oppgaven er hentet fra Naturfag SF (Brandt, Hushovd, & Tellefsen, 2020, s. 372).

5.4.5.1 Kort beskrivelse av tema

De individene av en art som holder til innenfor et avgrenset område kalles gjerne en populasjon, eller en bestand. Antallet individer i en populasjon variere med tiden, det kan skyldes faktorer som naturlige fiender, sykdommer som kan ramme hele eller deler av populasjonen, mangel på mat, eller plass, formeringsforhold, og eventuell netto tilflytting/fraflytting. For bakterier er vekstforholdene ofte gode, noe som kan resultere i en eksplosiv vekst, gjerne kalt ukontrollert vekst, siden den er eksponentiell.

5.4.5.2 Oppgave:

En bakteriekultur øker populasjonen hele tiden. Vekstraten r er på 1,0, som tilsvarer en økning på 100% per døgn. La populasjonen ved start være 1 bakterie. Skriv et program som illustrerer veksten de første 7 timene som en graf.

5.4.5.3 Problemløsning/Problemløsnings-strategi:

For å tegne en graf som skal illustrere veksten i bakteriekulturen på 7 timer, trengs verdier langs x-aksen og y-aksen. Siden vi skal se på endringene per time i 7 timer, oppretter vi en variabel som tar vare på denne verdien

```
ant_timer = 7
```

Det er naturlig å velge de ulike timene som forløper som x-verdier, og antall bakterier som y-verdier. X-verdiene kan genereres automatisk ved hjelp av en Python funksjon som linspace() eller range(), mens y-verdien må beregnes basert på de ulike x-verdiene ved hjelp av formelen

Bakterie nå = bakterie ved forrige måling + økning siste timen

I denne løsningen benyttes linspace() for å opprette en rekke med x-verdier. linspace() er en funksjon som gjøres tilgjengelig for Python-programmet gjennom at biblioteket pylab importeres.

For å ta vare på x-verdiene benyttes en tabell, som vi kaller array(). Denne tabellen opprettes først med nok antall plasser til å romme de 7 timene (ant_timer) vi ønsker målinger.

```
timer = linspace(0, ant_timer, ant_timer+1),
```

linspace oppretter her en tabell som begynner med verdien 0, og slutter med verdien 7 (ant_timer).

For å ta vare på y-verdiene benyttes også en tabell. Denne tabellen opprettes først med nok antall plasser til å romme bakterieantallet i de 7 timene vi ønsker målinger.

```
bakterier = zeros(ant_timer+1),
```

her blir det opprettet en variabel med navnet bakterier. Variabelen settes til å være en tabell av lengden (ant_timer + 1), som har 7 + 1 = 8 plasser tilgjengelig for å skrive verdier av bakterier for de ulike tidspunktene. (1 ved start, og deretter 7 til for hver time som har gått, til sammen 8 plasser). zeros() er en spesiell funksjon som er gjort tilgjengelig via pylab biblioteket. Det eneste den gjør er å fylle alle plassene i tabellen med verdien 0 (zero) når tabellen opprettes. Etter tabellen er opprettet ser den slik ut

```
bakterier = [0, 0, 0, 0, 0, 0, 0, 0]
```

Før vi kan begynne beregningene av bakterieveksten må vi sette en startverdi. Det må være minst 1 bakterie til stede for at formeringen skal kunne foregå

```
bakterier [0] = 1,
```

hvilket betyr at ved starttidspunktet tidspunkt 0, starter vi med 1 bakterie

Da ser tabellen foreløpig slik ut

```
bakterier = [1, 0, 0, 0, 0, 0, 0, 0]
```

For å simulere at timene går, benytter vi en for-løkke, som gjentar samme kode et visst antall ganger. For hver gjennomgang av for-løkken tenker vi oss at det da har gått en time. Inne i for-løkken må selve beregningen av bakterieveksten for den aktuelle timen beregnes og legges på plass på rette stedet i tabellen.

Til slutt, etter at alle de 7 timene har passert, og tabellen til bakterier er fylt av nye verdier, benyttes verdiene til å plotte grafen.

5.4.5.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere bakterieveksten i løpet av de 7 timene, og i tillegg lar en bruke funksjonen linspace() og zeros() i koden.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. ant_timer = 7, for antall timer vi ser på bakterieveksten
   b. r = 1, for vekstraten til bakteriekulturen per dag
   c. timer = linspace(0, ant_timer, ant_timer +1), for en tabell timer med x-verdier fra 0-7

      ```
      linspace(start, stopp, n)
      ```

      er en funksjon som tilbys av numpy-biblioteket (som er tilgengelig via pylab). Funksjonen tar et intervall fra start til stopp og deler inn i n like deler. I dette eksempelet opprettes en variabel som heter timer, som så får tildelt en liste med ulike verdier nemlig alle verdiene x E [0,7]. Det betyr at variabelen timer inneholder en tabell med følgende 8 verdier (ant_timer +1):

```
timer = [0, 1, 2, 3, 4, 5, 6, 7]
```

d. `bakterier = zeros(ant_timer +1)`, for alle y-verdier midlertidig fylt med 0-tall.

```
zeros(n)
```

er en funksjon som tilbys av numpy-biblioteket (som er tilgjengelig via pylab). Funksjonen oppretter en tabell, og fyller alle plassene n, med verdien 0. I dette eksempelet opprettes en variabel som heter bakterier, som så får tildelt en tabell med 8 plasser, alle fylt med verdien 0.

3) Definere for-løkke som kjører for alle definerte verdier av i. (Det vil si for alle de 7 tidspunktene det vil bli utført beregninger i dette tilfellet). For å beskrive antallet iterasjoner (gjentakelser) av for-løkken benyttes formuleringen

```
for i in range(1, ant_timer+1)
```

der i er en lokal variabel som kun opprettes for bruk i for-løkken, og som har som oppgave å holde rede på hvilken iterasjon en er kommet til. Altså hvilket nummer en er kommet til av alle de 7 gangene for-løkken skal gjentas i dette tilfellet.

Funksjonen range() benyttes for nettopp å beskrive hvor mange ganger løkken skal gjentas. Første verdien i range(startverdi, sluttverdi) gir startverdien, slik at ved første gjennomgang, blir variabelen i satt til 1, og programkoden inne i for-løkken kjøres. Den andre verdien i range (startverdi, sluttverdi) sier hva når en skal slutte å gjenta løkken. Så når i har økt fra i=1 til i=sluttverdi, slutter gjentakelsene av koden inne i for-løkken, og programmet går videre.

for-løkken sørger for beregning og registrering av ny verdi for bakterievekst

```
bakterier[i] = bakterier[i-1] + r * bakterier[i-1]
```

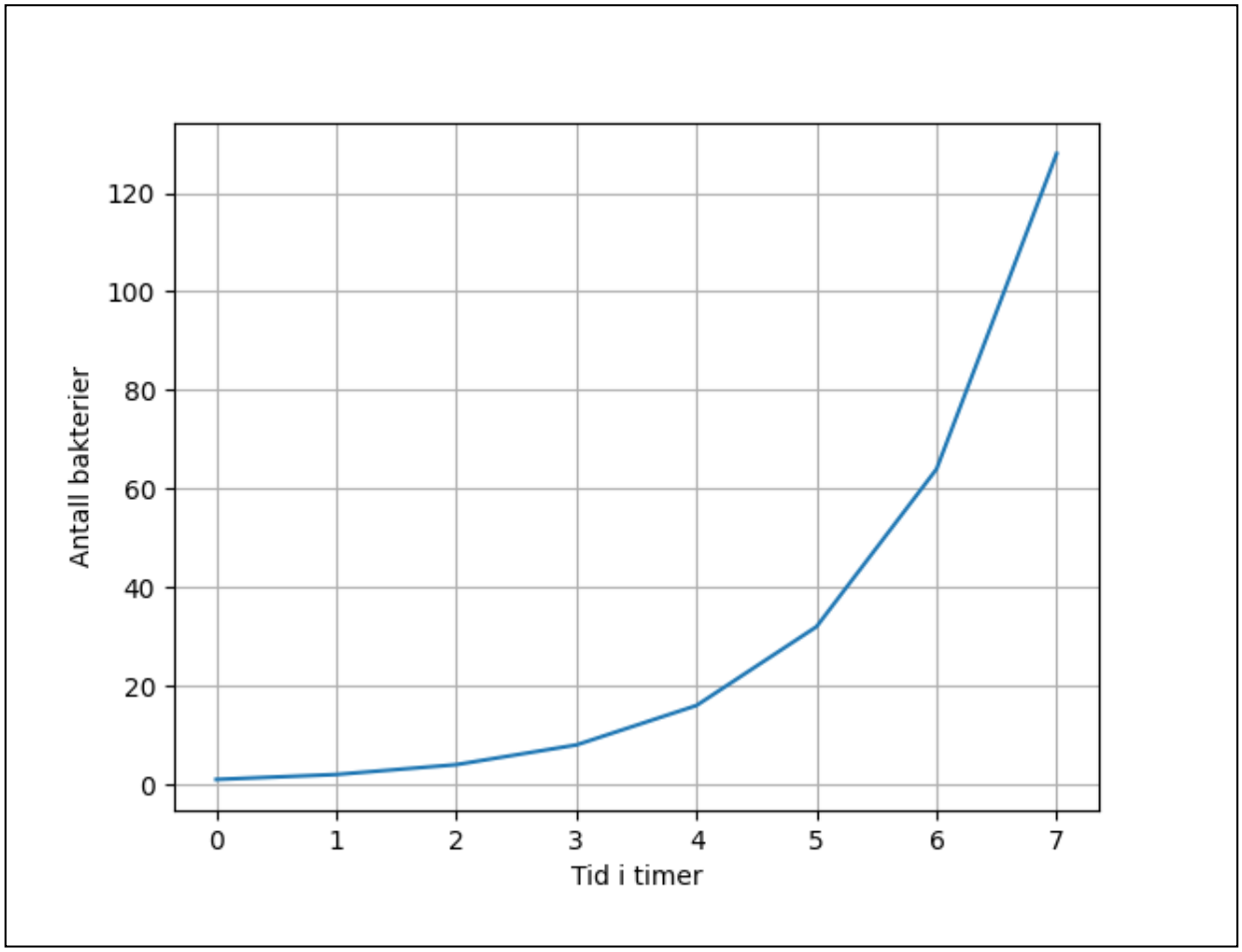
4) Plotter de registrerte verdien i x og y som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pyplot-biblioteket
   a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
   b. xlabel(«Streng»), setter beskrivende tekst til x-aksen
   c. ylabel(«Streng»), setter beskrivende tekst til y-aksen
   d. grid(), tegner inn rutenett på figuren
   e. show(), funksjon som tegner grafen til skjerm

5.4.5.5 Program:

```python
# Bakterievekst

from pylab import *

# Vi vil finne veksten i løpet av 7 timer
ant_timer = 7

# vekstraten (hvor stor andel av populasjonen som kommer til hver dag)
r = 1 # vekstrate på 1 gir 100% vekst per dag

# lager verdier til første-aksen til grafen
timer = linspace(0, ant_timer, ant_timer+1)

# Lager en tom array, og fyller alle elementene med verdien 0 for hver time
bakterier = zeros(ant_timer+1)

bakterier[0] = 1 # setter startverdien ved start til 1 bakterie

# Bruker en løkke for å beregne antall bakterier for hver time
for i in range(1, ant_timer+1):
    bakterier[i] = bakterier[i-1] + r * bakterier[i-1]


plot(timer, bakterier)      # lager grafen
xlabel("Tid i timer")       # tittel langs første-aksen
ylabel("Antall bakterier")  # tittel langs andre-aksen
grid()                      # tegner et rutenett
show()                      # viser grafen
```

Figur 5-143: Kode for oppgave 5 – Bakterievekst

5.4.5.6 Output:



5.4.6 Oppgave 6 – Gjøre oppslag i liste

Inspirasjon til denne oppgaven er hentet fra Element 10 (Arntzen, Bækkedal, Fossestøl, & Fægri, 2022, s. 50)

5.4.6.1 Kort beskrivelse av tema

Alle bølger har en frekvens f som er antall svingninger, og en bølgelengde. Også all elektromagnetisk stråling har en frekvens og en bølgelengde. Bølgelengden er definert som lengden mellom to bølgetopper. Frekvens er et mål på hvor mange bølgelengder som passerer i løpet av ett sekund.

Mennesker kan oppfatte bølger som har en bølgelengde mellom ca. 400 og 750 nanometer. Med andre ord: Vi kan se det lyset dersom bølgelengdene er i det spekteret. Eksempel på lys vi ikke kan se, er ultrafiolett lys. Det har bølgelengde mellom 100 og 400 nanometer.

En nanometer er 0,000 000 0001 meter, eller $1*10^{-9}$ meter.

5.4.6.2 Oppgave:

Vi har en liste b_lengde hvor det er oppgitt eksempler på forskjellige bølgelengder i nanometer. I en annen liste farge har vi navnet som hører til eksemplene i den første listen. Skriv ut navnet på de bølgene vi kan se med tilhørende bølgelengde.

```
b_lengde = [500, 575, 350, 10000000, 690, 800, 420, 570, 425]

farge = ['grønn', 'oransje', 'ultrafiolett', 'radiobølger', 'rød',
'infrarød', 'fiolett', 'gul', 'blå']
```

5.4.6.3 Problemløsning/Problemløsnings-strategi:

Det vesentlige i denne oppgaven er å sammenligne de to listene. Vi skal se på verdiene i den første listen, og lokalisere tilhørende verdi i den andre listen. Til dette må vi finne indeksen i den første, og bruke denne til å slå opp i den andre.

Til dette formålet tenker vi at det er lurt å bruke en for-løkke. Da kan man gå gjennom hele listen. Man må også bruke betingelses-operator for å sjekke om et tall er mellom 400 og 750 nanometer.

5.4.6.4 Algoritme:

1) Legge inn de to listene. Vi velger å kalle de b_lengde og farge.

2) Skrive for-løkken. Vi vil sjekke hvert element i listen b_lengde:

```
for lengde in b_lengde
```

3) Skrive if-setningen. Vi ønsker å sjekke om hvert element i listen er større enn eller lik 400 og mindre enn eller lik 750:

```
if lengde >= lengde and lengde <= 750
```

4) Finne indeksen til elementet som passerer if-setningen. Indeks finner vi ved å slå opp verdien til det gjeldende elementet i listen. Vi oppretter en ny variabel som heter index til dette formålet, og skriver at den variabelen skal inneholde indeks-verdien til elementet:

```
index = b_lengde.index(lengde)
```

5) Vi ønsker å skrive ut den bølgelengden som nettopp ble sjekket, og bruke indeks-verdien til det bølgeelementet for å slå opp i listen farge. Vi ønsker å skrive ut både bølgelengden og fargen, med litt forklarende tekst:

5.4.6.5 Program Python:

```python
# Gjøre oppslag i liste for å finne farge tilhørende lysintensitet

b_lengde = [500, 575, 350, 10000000, 690, 800, 420, 570, 425]       # I nanometer
farge = ['grønn', 'oransje', 'ultrafiolett', 'radiobølger', 'rød', 'infrarød',
         'fiolett', 'gul', 'blå']

for lengde in b_lengde:
    if lengde ⩾ 400 and lengde ⩽ 750:
        index = b_lengde.index(lengde)
        print('Bølgelengden', lengde, 'ses som', farge[index])

```

Figur 5-144: Kode for oppgave 6 – Gjøre oppslag i liste

5.4.6.6 Output:

```
Bølgelengden 500 ses som grønn
Bølgelengden 575 ses som oransje
Bølgelengden 690 ses som rød
Bølgelengden 420 ses som fiolett
Bølgelengden 570 ses som gul
Bølgelengden 425 ses som blå
```

5.5 Oppgavehefte i Python for VG2

5.5.1 Oppgave 1 – Rettlinjet bevegelse

Inspirasjon til denne oppgaven er hentet fra (Callin et al., 2021).

5.5.1.1 Kort beskrivelse av tema

Fart v er definert som v = s / t, der s er strekningen et legeme beveger seg i løpet av en tidsperiode t.

5.5.1.2 Oppgave:

Et legeme holder konstant fart v = 5,0 m/s i et tidsrom på 3,0 s. Beskriv hvordan tilbakelagt strekning s kan beregnes i stegvise inkrement gjennom legemets bevegelsesforløp, ved å beregne tilbakelagt distanse for en rekke antall små tids-økninger (Δt).

5.5.1.3 Problemløsning/Problemløsnings-strategi:

Om en kjenner verdien til to av de tre parameterne i ligningen, kan en finne verdien til den tredje parameteren ved enkel omskriving av ligningen. Vi skriver om ligningen med hensyn på s, og får

s = v * t

I denne oppgaven ønsker vi å beregne tilbakelagt distanse s gjentatte ganger etter hvert som legemet er i bevegelse. For hver ekstra tidsenhet (Δt) legemet er i bevegelse, fra starttidspunktet t_0 = 0 s, til tiden er ute ved t_s = 3.0 s, beregnes det hvor lang distanse legemet har beveget seg, og resultatet skrives ut. Tidsinkrementet Δt som blir benyttet er Δt = 0.1s. Dette tilsvarer da (3 – 0) s / 0.1 s = 30 ulike tidspunkter der tilbakelagt strekning vil bli beregnet.

For hvert tidsinkrement Δt som forløper, vil strekningen legemet har beveget seg i denne lille perioden ha endret seg med en distanse lik (v * Δt). Den nye totalstrekningen legemet har beveget seg frem til det siste tidspunktet vil da være strekningen s ved forrige beregning, pluss den ekstra distansen som er tilbakelagt i løpet av tidsintervallet Δt.

5.5.1.4 Algoritme:

1) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. v som beskriver verdien av legemets konstante fart
    b. t som beskriver startverdien for det totale tidsintervallet
    c. t_slutt som beskriver totallengden av tidsintervallet
    d. dt som beskriver hvor stor hver tidsøkning (Δt) er mellom hver beregning

e. s som beskriver tilbakelagt strekning for legemet

2) Definere formel for beregning av oppdatert strekning for hvert tidsintervall => s = s + v*dt

3) Definere funksjon utskrift() som har som oppgave å skrive ut resultatene til skjerm i et lettleselig format for brukeren.
   a. Benytte round()-funksjonen som er inkludert i Python til å avrunde desimaltall til 2 desimaler

```
round(20.45645646, 2) -> Output: 20.46
```

4) Benytte while-løkke som
   a. oppdaterer verdiene for tilbakelagt strekning s, samt tidspunkt t, etter hvert som tidsøkningen (Δt) går stegvis fra start-tiden 0, til slutt-tiden 3.0s
   b. kaller funksjonen utskrift() som skriver svaret på hver enkelt beregning med egnet ledetekst, til skjerm.

5.5.1.5 Program:

```python
# Rettlinjet bevegelse med konstant fart

# Informasjon om bevegelsen
v = 5.0             # farten til gjenstanden , m/s
s = 0.0             # startposisjon, m
t = 0.0             # starttid, s

# Simuleringsteknisk
dt = 0.1            # lengde på tidssteg, m/s
t_slutt =  3.0      # sluttid for simulering, s

# funksjon som skriver ut verdiene av t, s, og v
def utskrift():
    print(f'Ved tiden t = {round(t, 2)}s, har legemet farten v = {round(v, 2)}m/s, og har tilbakelagt s = {round(s, 2)}m')

# Løkke som beregner ny posisjon for hvert tidssteg dt
while t < t_slutt:
    s = s + v*dt    # regner ut ny posisjon
    t = t + dt      # tid økes med dt
    utskrift()      # skriver ut oppdaterte verdier
```

Figur 5-145: Kode for oppgave 1 – Rettlinjet bevegelse

5.5.1.6 Output:

```
Ved tiden t = 0.1s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 0.5m
Ved tiden t = 0.2s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 1.0m
Ved tiden t = 0.3s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 1.5m
.
.
.
Ved tiden t = 2.8s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 14.0m
Ved tiden t = 2.9s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 14.5m
Ved tiden t = 3.0s, har legemet farten v = 5.0m/s, og har tilbakelagt s = 15.0m
```

5.5.2 Oppgave 2 – Konstant akselerasjon

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al., 2021).

5.5.2.1 Kort beskrivelse av tema

Gjennomsnittakselerasjon i et tidsintervall Δt er gitt ved endringen i fart dividert på endringen i tid (tidsintervallet).

$a = \Delta v / \Delta t$, dette tilsvarer

$a = (v_1 - v_0) / (t_1 - t_0)$

5.5.2.2 Oppgave:

Et legeme holder en fart på 5,0 m/s, og utsettes for en kraft slik at det akselererer med en akselerasjon a= 2 m/s^2 i fartsretningen. Legemet er i bevegelse i 3 sekunder. Beregn verdier for strekning s og fart v i tidsrommet legemet er i bevegelse, og bruk disse verdiene til å tegne en graf som illustrerer legemets bevegelse som tilbakelagt strekning som funksjon av tid. Benytt en Δt på 0.1 s som tidsintervall mellom hver beregning.

5.5.2.3 Problemløsning/Problemløsnings-strategi:

Ettersom legemet utsettes for akselerasjon, må det fortløpende beregnes ny momentantfart ved de ulike tidspunktene, i tillegg til ny posisjon (tilbakelagt distanse), samt oppdatert tidspunkt for hvert nytt tilbakelagt tidsintervall Δt.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a*Δt), ny posisjon er da (s + v*Δt), og nytt tidspunkt (t + Δt).

Verdiene av legemets posisjon s, og tiden t legemet har vært i bevegelse, registreres fortløpende i hver sin liste under forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen skal plottes.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes en while-løkke som løper fra starttiden t = 0, til sluttiden t = 3.0 er oppnådd. Grafen til legemets bevegelse plottes til slutt ved å benytte biblioteket pylab som importeres i Python koden.

5.5.2.4 Algoritme:

4) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere legemets bevegelse i løpet av tidsrommet.

5) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. s = 0, for startposisjonen til legemet
    b. t = 0, for starttidspunktet til legemet
    c. v = 5.0, for startfarten til legemet
    d. a = 2.0, for den konstante akselerasjonen til legemet
    e. t_slutt = 3.0, for sluttiden for beregning
    f. dt = 0.1, for tidsintervall mellom hver beregning
    g. t_verdier = [t], for liste over alle tidspunkt der det blir gjort beregninger

h. s_verdier = [s], for liste over alle posisjoner legemet befinner seg ved de ulike tidspunktene beregningene blir utført.

6) Definere while-løkke som står for
    a. beregning av nye oppdaterte verdier av fart v, posisjon s, og tidspunkt t, for hvert tidssteg dt.
    b. oppdatering av listene t_verdier og s_verdier med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

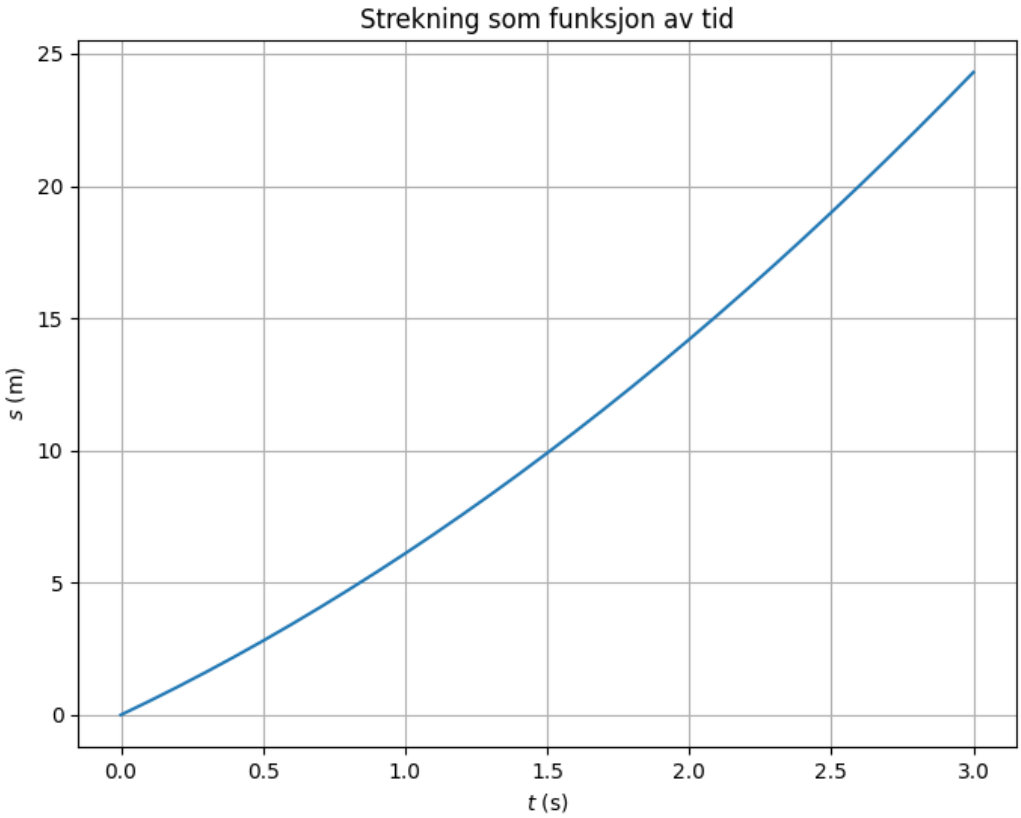
7) Plotter de registrerte verdien i t_verdier og s_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title("Streng"), setter tittel på grafen
    c. xlabel("Streng"), setter beskrivende tekst til x-aksen
    d. ylabel("Streng"), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

5.5.2.5 Program:

```python
# Konstant akselerasjon

from pylab import *

# Informasjon om bevegelsen
s = 0.0                     # startposisjon, m
v = 5.0                     # farten til gjenstanden , m/s
a = 2.0                     # akselerasjonen til gjenstanden, m/s^2
t = 0.0                     # starttid, s

# Simuleringsteknisk
t_slutt = 3.0               # sluttid,  s
dt = 0.1                    # lengde av tidssteg, s
t_verdier = [t]             # liste der verdiene av t blir lagt inn
s_verdier = [s]             # liste der verdiene av s blir lagt inn

# Løkke som regner ut ny posisjon for hvert tidssteg
while t < t_slutt:
    v = v + a*dt            # beregner ny momentanfart
    s = s + v*dt            # beregner ny posisjon
    t = t + dt              # tid økes med dt

    t_verdier.append(t)     # registrerer verdien av t i en tidsliste
    s_verdier.append(s)     # registrerer verdien av s i en posisjonsliste

# Tegner graf av forløpet
plot(t_verdier, s_verdier)              # lager grafen
title("Strekning som funksjon av tid")  # setter tittel på grafen
xlabel("$t$ (s)")                       # x-akse tittel, $ gir kursiv
ylabel("$s$ (m)")                       # y-akse tittel
grid()                                  # legger til rutenett
show()                                  # viser grafen
```

Figur 5-146: Kode for oppgave 2 – Konstant akselerasjon

5.5.2.6 Output:


Strekning som funksjon av tid
25
20
15
10
5
0
s (m)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
t (s)

5.5.3 Oppgave 3 – Ikke-konstant akselerasjon

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al., 2021).

5.5.3.1 Kort beskrivelse av tema

I daglige situasjoner kan akselerasjonen av et legeme endre seg over tid. Eksempelvis kan en se på en idrettsutøver som konkurrerer i 100 m sprint. Ved starten av løpet er akselerasjonen typisk høy, og farten økes kraftig når løperen forlater startblokkene, men allerede etter kort tid er det vanskelig å opprettholde en fartsøkning (akselerasjon). Etter 10-12 meter er det vanskelig å fortsette å øke hastigheten (akselerasjon > 0), og etter 50-60 meter av løpet er unnagjort er det veldig få som klarer å opprettholde konstant fart (akselerasjon <= 0).

5.5.3.2 Oppgave:

Ved starten av et 100-meter løp akselererer en sprinter ut av startblokkene med en akselerasjon på 6.0 m/s^2. Etter få sekunder er det vanskelig å opprettholde akselerasjonen, og løperen når sin toppfart der akselerasjonen da er 0. Etter det punktet er det vanskelig å holde maksimal hastighet, og sprinteren løper typisk gradvis saktere ettersom sprinteren slipper opp for krefter. Da er akselerasjonen negativ.

I dette eksempelet tenker vi oss at sprinterens akselerasjon følger en lineær funksjon

$a(v) = -0{,}5\ s^{-1} * v + 6.0\ m/s^2$

Fremstill en graf som illustrerer sprinterens fart v som funksjon av den tilbakelagte strekningen s. Benytt en Δt på 0.01 s som tidsintervall mellom hver beregning.

5.5.3.3 Problemløsning/Problemløsnings-strategi:

Sprinteren starter fra startblokkene med en startfart v = 0. For hvert tidssteg Δt som går mens sprinteren løper de 100 meterne, beregnes løperens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(v)*dt), ny posisjon er da (s + v*dt).

Verdiene av sprinterens posisjon s, og momentanfarten v sprinteren har ved hvert tidssteg, registreres fortløpende i hver sin liste (s_verdier og v_verdier) under forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen skal plottes.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes en while-løkke som løper fra startposisjonen s = 0, til sluttposisjonen s = 100.0 er oppnådd. Dette tilsvarer distansen sprinteren skal løpe på 100m.

For også å beregne ny akselerasjon for hvert tidssteg, defineres en egen funksjon a(v) som har som oppgave å kalkulere og returnere akselerasjonen ifølge det lineære uttrykket

$a(v) = -0{,}5\ s^{-1} * v + 6.0\ m/s^2$

Funksjonen tar inn farten v som argument, og returnerer akselerasjonen som en desimalverdi.

Grafen til sprinterens bevegelse plottes til slutt ved å benytte biblioteket pylab som importeres i Python koden.

5.5.3.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere sprinterens fart etter hvert som de 100 meterne (distansen) tilbakelegges.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. s = 0, for startposisjonen til sprinteren
    b. v = 0, for startfarten til sprinteren
    c. s_slutt = 100, for distansens lengde
    d. dt = 0.01, for tidsintervall mellom hver beregning
    e. v_verdier = [v], for liste over alle fartene sprinteren holder ved de ulike tidspunktene beregningene blir utført
    f. s_verdier = [s], for liste over alle posisjoner sprinteren befinner seg ved de ulike tidspunktene beregningene blir utført

3) Definere funksjonen a(v) som kalkulerer og returnerer akselerasjonen
    a. Oppretter lokal variabel 'aks' for lagring av beregnet verdi for akselerasjon
    b. Tar inn en verdi v for momentanfarten i øyeblikket
    c. Benytter det lineære uttrykket for beregningen av akselerasjonen -> aks = -0.5 * v + 6
    d. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge posisjonen s < s_slutt. While-løkken står for:
    a. beregning av nye oppdaterte verdier av fart v og posisjon s for hvert tidssteg dt.
    b. oppdatering av listene v_verdier og s_verdier med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

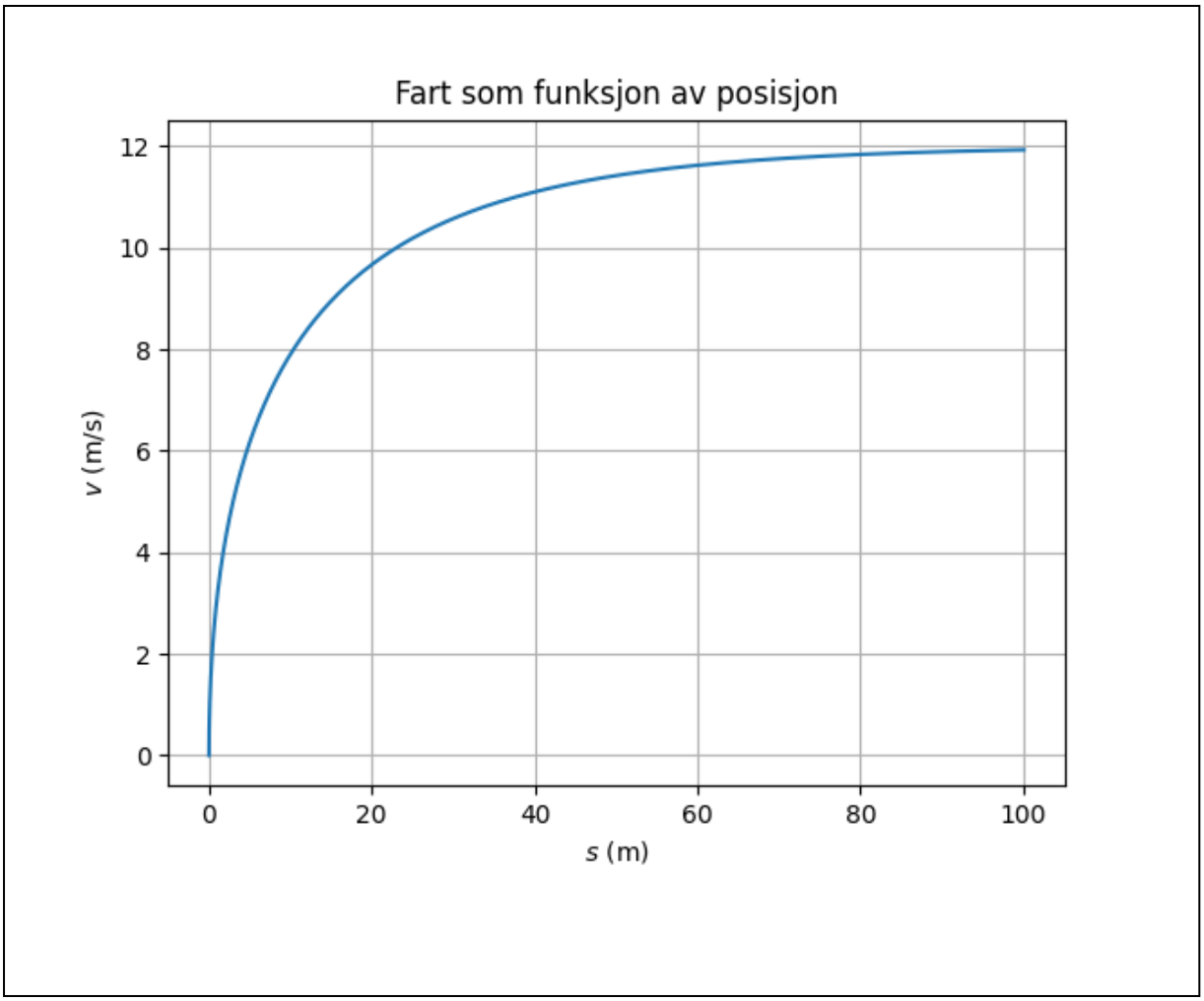
5) Plotter de registrerte verdien i s_verdier og v_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title("Streng"), setter tittel på grafen
    c. xlabel("Streng"), setter beskrivende tekst til x-aksen
    d. ylabel("Streng"), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

5.5.3.5 Program:

```python
# Ikke-konstant akselerasjon

from pylab import *

# Informasjon om bevegelsen
s = 0   # startposisjon, m
v = 0   # startfart, m/s

# Simuleringsteknisk
s_slutt = 100               # sluttposisjon, m
dt = 0.01                   # lengde på tidssteg, s
s_verdier = [s]             # liste med verdier for posisjon
v_verdier = [v]             # liste med verdier for fart

def a(v):                   # akselerasjonen er en funksjon av farten
    aks = -0.5 * v + 6      # regnar ut akselerasjonen
    return aks              # returnerer akselerasjonen

# Løkke som regner ut ny akselerasjon, fart og posisjon
# til løperen for hvert tidssteg
while s < s_slutt:          # etter 100 meter er løpet over
    v = v + a(v)*dt         # regner ut ny fart
    s = s + v * dt          # regner ut ny posisjon
    s_verdier.append(s)     # legger s inn i posisjonslisten
    v_verdier.append(v)     # legger v inn i fartslisten

# Tegning av graf
plot(s_verdier, v_verdier)              # lager grafen
title("Fart som funksjon av posisjon")  # tittel på graf
xlabel("$s$ (m)")                       # x-akse tittel
ylabel("$v$ (m/s)")                     # y-akse tittel
grid()                                  # viser rutenett
show()                                  # viser grafen
```

Figur 5-147: Kode for oppgave 3 – Ikke-konstant akselerasjon

5.5.3.6 Output:


Fart som funksjon av posisjon
v (m/s)
s (m)
0
2
4
6
8
10
12
0
20
40
60
80
100

5.5.4 Oppgave 4 – Ikke-konstant akselerasjon

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al. , 2021).

5.5.4.1 Kort beskrivelse av tema

I daglige situasjoner kan akselerasjonen av et legeme endre seg over tid. Eksempelvis kan en se på en idrettsutøver som konkurrerer i 100 meter sprint. Ved starten av løpet er akselerasjonen typisk høy, og farten økes kraftig når løperen forlater startblokkene, men allerede etter kort tid er det vanskelig å opprettholde en fartsøkning (akselerasjon). Etter 10-12 meter er det vanskelig å fortsette å øke hastigheten (akselerasjon > 0), og etter 50-60 meter av løpet er unnagjort er det veldig få som klarer å opprettholde konstant fart (akselerasjon <= 0).

5.5.4.2 Oppgave:

Ved starten av et 100-meter løp akselererer en sprinter ut av startblokkene med en akselerasjon på 6.0 m/^{2}. Etter få sekunder er det vanskelig å opprettholde akselerasjonen, og løperen når sin toppfart der akselerasjonen da er 0. Etter det punktet er det vanskelig å holde maksimal hastighet, og sprinteren løper typisk gradvis saktere ettersom sprinteren slipper opp for krefter. Da er akselerasjonen negativ.

I dette eksempelet tenker vi oss at sprinterens akselerasjon følger en lineær funksjon

$a(v) = -0{,}5s^{-1} * v + 6.0\ m/s^2$

a) La programmet skrive ut hvor lang tid sprinteren bruker på 100m.

b) Ta utgangspunkt i simuleringen og tegn posisjonen som funksjon av tiden

5.5.4.3 Problemløsning/Problemløsnings-strategi:

En sprinter starter fra startblokkene med en startfart v = 0. For hvert tidssteg Δt som går mens sprinteren løper de 100 meterne, beregnes løperens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(v)*dt), ny posisjon er da (s + v*dt).

Verdiene av sprinterens posisjon s, og momentanfarten v sprinteren har ved hvert tidssteg, registreres fortløpende i hver sin liste (s_verdier og v_verdier) under forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen skal plottes.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes en while-løkke som løper fra startposisjonen s = 0, til sluttposisjonen s = 100.0 er oppnådd. Dette tilsvarer distansen sprinteren skal løpe på 100 m.

For også å beregne ny akselerasjon for hvert tidssteg, defineres en egen funksjon a(v) som har som oppgave å kalkulere og returnere akselerasjonen ifølge det lineære uttrykket

$a(v) = -0{,}5s^{-1} * v + 6.0m/s^2$

Funksjonen tar inn farten v som argument, og returnerer akselerasjonen som en desimalverdi.

a) Utskrift av tiden t sprinteren har brukt på å tilbakelegge de 100 meterne skrives ut til skjerm med passende ledetekst. Verdien av tiden leses direkte fra variabelen t.

b) Grafen til sprinterens posisjon som funksjon av tiden plottes til slutt ved å benytte biblioteket pylab som importeres i Python koden.

5.5.4.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere sprinterens fart etter hvert som de 100 meterne (distansen) tilbakelegges.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. s = 0, for startposisjonen til sprinteren
    b. v = 0, for startfarten til sprinteren
    c. s_slutt = 100, for distansens lengde
    d. dt = 0.01, for tidsintervall mellom hver beregning
    e. v_verdier = [v], for liste over alle fartene sprinteren holder ved de ulike tidspunktene beregningene blir utført
    f. s_verdier = [s], for liste over alle posisjoner sprinteren befinner seg ved de ulike tidspunktene beregningene blir utført.

3) Definere funksjonen a(v) som kalkulerer og returnerer akselerasjonen
    a. Oppretter lokal variabel 'aks' for lagring av beregnet verdi for akselerasjon
    b. Tar inn en verdi v for momentanfarten i øyeblikket
    c. Benytter det lineære uttrykket for beregningen av akselerasjonen -> aks = -0.5 * v + 6
    d. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge posisjonen s < s_slutt. While-løkken står for:
    a. beregning av nye oppdaterte verdier av fart v og posisjon s for hvert tidssteg Δt.
    b. oppdatering av listene v_verdier og s_verdier med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

5) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengen formateres ved hjelp av f-String-funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde tiden til 3 desimaler.

```
print (f'Sprinteren brukte {round(t, 3)} sekunder på 100-
meteren!')
```

6) Plotter de registrerte verdien i s_verdier og t_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere sprinterens posisjon som funksjon av tiden.
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title("Streng"), setter tittel på grafen
    c. xlabel("Streng"), setter beskrivende tekst til x-aksen
    d. ylabel("Streng"), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

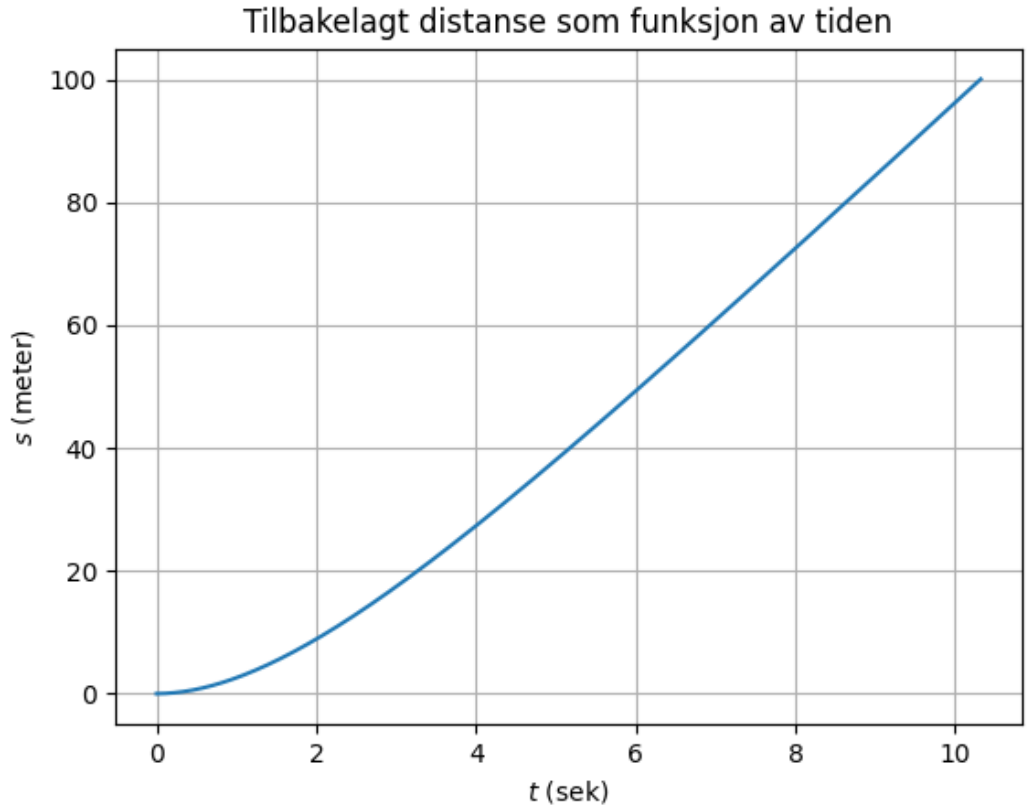
5.5.4.5 Program:

```python
# Ikke-konstant akselerasjon

from pylab import *

# Informasjon om bevegelsen
s = 0   # startposisjon, m
v = 0   # startfart, m/s
t = 0   # starttid, s

# Simuleringsteknisk
s_slutt = 100               # sluttposisjon, m
dt = 0.01                   # lengde på tidssteg, s
s_verdier = [s]             # liste med verdier for posisjon
t_verdier = [t]             # liste med verdier for tid

def a(v):                   # akselerasjonen er en funksjon av farten
    aks = -0.5 * v + 6      # regner ut akselerasjonen
    return aks              # returnerer akselerasjonen

# Løkke som regner ut ny akselerasjon, fart og posisjon
# til løperen for hvert tidssteg
while s < s_slutt:          # etter 100 meter er løpet over
    v = v + a(v)*dt         # regner ut ny fart
    s = s + v * dt          # regner ut ny posisjon
    t = t + dt              # holder rede på tid som er gått
    s_verdier.append(s)     # legger s inn i posisjonslisten
    t_verdier.append(t)     # legger t inn i tidslisten

# Utskrift av sprinterens tid på 100-meteren
print(f'Sprinteren brukte {round(t, 3)} sekunder på 100-meteren!')

# Tegning av graf
plot(t_verdier, s_verdier)                          # lager grafen
title("Tilbakelagt distanse som funksjon av tiden") # tittel på graf
xlabel("$t$ (sek)")                                 # x-akse tittel
ylabel("$s$ (meter)")                               # y-akse tittel
grid()                                              # viser rutenett
show()                                              # viser grafen
```

Figur 5-148: Kode for oppgave 4 – Ikke-konstant akselerasjon

5.5.4.6 Output:

```
Sprinteren brukte 10.32 sekunder på 100-meteren!
```



5.5.5 Oppgave 5 – Ikke-konstant akselerasjon

Inspirasjon til denne oppgaven er hentet fra Ergo 1 Fysikk (Callin et al., 2021)

5.5.5.1 Kort beskrivelse av tema

I daglige situasjoner kan akselerasjonen av et legeme endre seg over tid. Eksempelvis kan en se på en sportsbil som står og venter på å få grønt lys i et lyskryss. Når bilen starter å kjøre er akselerasjonen typisk høy, og farten økes kraftig når bilen settes i bevegelse. Kun de sprekeste bilene med kraftige motorer kan opprettholde akselerasjonen over særlig tid mer enn 10-12 sekunder.

5.5.5.2 Oppgave:

En bil kjører ut av at lyskryss. Akselerasjonen de første 10 sekundene følger funksjonen

$$a(t) = -0.40 m/s^3 * t + 4.0\ m/s^2$$

a) lag en simulering som viser farten til bilen som funksjon av tiden

b) b) hvor langt kommer bilen på 10 sekunder?

5.5.5.3 Problemløsning/Problemløsnings-strategi:

En bil akselererer ut av et lyskryss med en startfart v = 0. For hvert tidssteg Δt som går mens bilen kjører gjennom krysset, beregnes bilens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(t)*dt), ny posisjon er da (s + v*dt).

Oppgaveteksten presiserer at akselerasjons-funksjonen gjelder for de første 10 sekundene. Følgelig begrenses tiden t som E [0,10].

Verdiene av alle tidspunktene t der det foretas beregninger av bilens posisjon s, og momentanfart v bilen har, registreres fortløpende. Tidspunktene t og momentanfarten v til bilen, legges til i liste (t_verdier og v_verdier) under forløpet, mens variabelen s som holder rede på tilbakelagt strekning oppdateres gjennom forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen (fart som funksjon av tiden) skal plottes, og sluttverdien av variabelen s brukes for å uttrykke distansen som er tilbakelagt på de 10 sekundene det foretas beregninger.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes da en while-løkke som løper fra starttidspunktet t = 0, til sluttidspunktet t = 10.0 er oppnådd. Dette tilsvarer de 10 sekundene bilen er i akselerasjon ut av lyskrysset.

For også å beregne ny akselerasjon for hvert tidssteg, benyttes oppgitt funksjon a(t) som har som oppgave å kalkulere og returnere akselerasjonen ifølge det lineære uttrykket

$$a(t) = -0.40 m/s^3 * t + 4.0\ m/s^2$$

5.5.5.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere bilens fart etter hvert som de 10 sekundene (tidsintervallet) tilbakelegges.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. s = 0, for startposisjonen til bilen
   b. v = 0, for startfarten til bilen
   c. t = 0, for startverdi av tidsperioden som skal benyttes i beregning
   d. t_slutt = 10, for tidsperiodens lengde
   e. dt = 0.01, for tidsintervall mellom hver beregning
   f. v_verdier = [v], for liste over alle fartene bilen holder ved de ulike tidspunktene beregningene blir utført
   g. t_verdier = [t], for liste over alle tidspunkt beregningene av bilens attributter blir utført

3) Definere funksjonen a(t) som kalkulerer og returnerer akselerasjonen
   a. Oppretter lokal variabel 'aks' for lagring av beregnet verdi for akselerasjon
   b. Tar inn en verdi t for tidspunktet for beregning av aks
   c. Benytter det lineære uttrykket for beregningen av akselerasjonen -> aks = -0.4 * t + 4
   d. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge tidspunktet t < t_slutt. While-løkken står for:
   a. beregning av nye oppdaterte verdier av fart v, posisjon s og forløpt tid t, for hvert tidssteg dt.
   b. oppdatering av listene v_verdier og t_verdier med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

5) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengen formateres ved hjelp av f-String-funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde tiden til 3 desimaler.

```
print (f'Etter 10 sekunder har bilen kjørt {round(t, 3)}
meter!')
```

6) Plotter de registrerte verdien i t_verdier og v_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere bilens fart som funksjon av tiden.
   a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
   b. title("Streng"), setter tittel på grafen
   c. xlabel("Streng"), setter beskrivende tekst til x-aksen
   d. ylabel("Streng"), setter beskrivende tekst til y-aksen

e. grid(), tegner inn rutenett på figuren
f. show(), funksjon som tegner grafen til skjerm

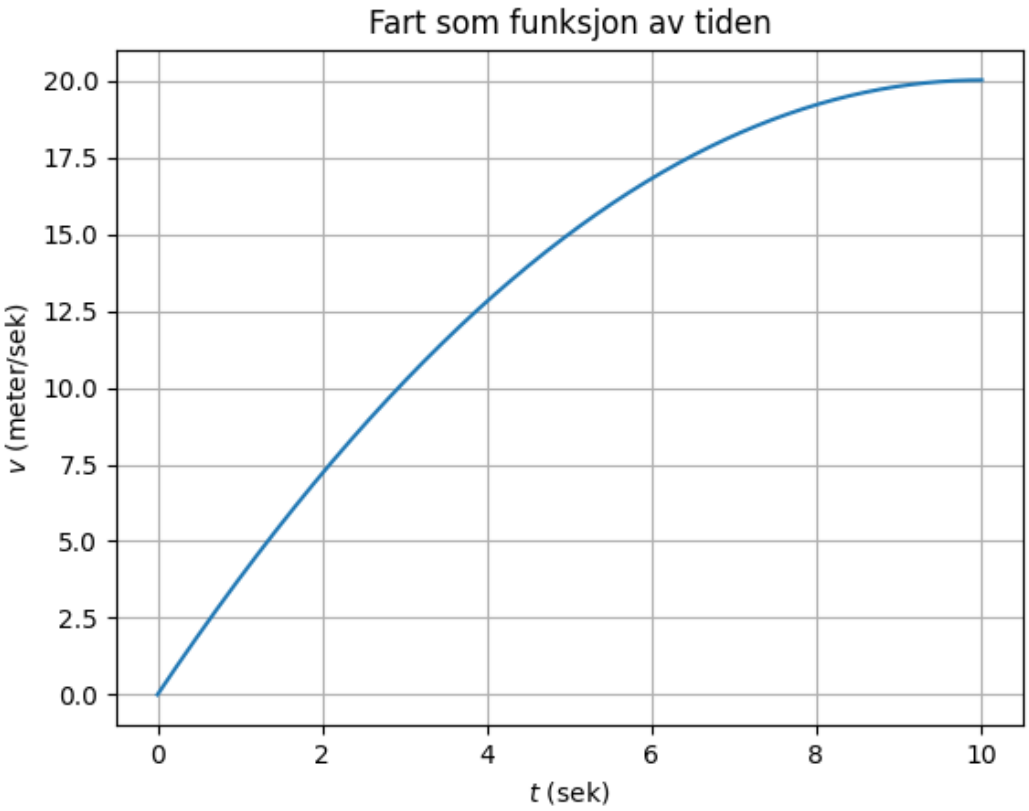
5.5.5.5 Program:

```python
# Ikke-konstant akselerasjon

from pylab import *

# Informasjon om bevegelsen
s = 0   # startposisjon, m
v = 0   # startfart, m/s
t = 0   # starttid, s

# Simuleringsteknisk
t_slutt = 10                # sluttid, s
dt = 0.01                   # lengde på tidssteg, s
v_verdier = [v]             # liste med verdier for fart
t_verdier = [t]             # liste med verdier for tid

def a(t):                   # akselerasjonen som en funksjon av tiden
    aks = -0.4 * t + 4      # regner ut akselerasjonen
    return aks              # returnerer akselerasjonen

# Løkke som regner ut ny akselerasjon, fart og posisjon
# til bilen for hvert tidssteg
while t < t_slutt:          # etter 10 sekunder slutter beregningene
    v = v + a(t)*dt         # regner ut ny fart
    s = s + v * dt          # regner ut ny posisjon
    t = t + dt              # holder rede på tid som er gått
    v_verdier.append(v)     # legger v inn i fartslisten
    t_verdier.append(t)     # legger t inn i tidslisten

# Utskrift av bilens tilbakelagte distanse s etter 10 sekunder
print(f'Etter 10 sekunder har bilen kjørt {round(s, 3)} meter!')

# Tegning av graf
plot(t_verdier, v_verdier)          # lager grafen
title("Fart som funksjon av tiden") # tittel på graf
xlabel("$t$ (sek)")                 # x-akse tittel
ylabel("$v$ (meter/sek)")           # y-akse tittel
grid()                              # viser rutenett
show()                              # viser grafen
```

Figur 5-149: Kode for oppgave 5 – Ikke-konstant akselerasjon

5.5.5.6 Output:

```
Etter 10 sekunder har bilen kjørt 133.734 meter!
```



5.5.6 Oppgave 6 – Konstante krefter

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al., 2021)

5.5.6.1 Kort beskrivelse av tema

Fra kapittelet om rettlinjet bevegelse ble det sett på legemer som beveget seg med konstant eller varierende akselerasjon. Legemets akselerasjon skyldes at legemet er påvirket av ytre krefter som gravitasjon, friksjon, støt fra andre legemer, eller luftmotstand som endrer momentanfarten til legemet i et tidspunkt t.

Når en skal løse oppgaver der nevnte krefter kan påvirke legemet, er prinsippet for beregning det samme – de ulike start-verdiene for legemet registreres, og deretter blir det beregnet hvilke endringer resultant-kraften som påvirker legemet medfører for hver Δt tidsperiode en foretar beregningene.

5.5.6.2 Oppgave:

En trekloss glir horisontalt bortover et trebord. Det er friksjon mellom treklossen og bordet. Treklossens masse er m = 250 gram, og klossens startfart v_0 = 2,3 m/s

a) Plot en graf som beskriver klossens bevegelse over bordplaten.
b) Hvor langt når treklossen før den stopper?
c) Hvor lang tid går det før treklossen stopper?

5.5.6.3 Problemløsning/Problemløsnings-strategi:

Det er oppgitt at friksjonstallet for to legemer av trevirke er μ = 0,3 (μ = my). Ettersom bevegelsen er horisontal, vet vi at normalkraften N er lik gravitasjonskraften G:

$$N = G$$

Friksjonskraften kan beregnes ved formelen

$$R = \mu * N$$

og er motsatt rettet av bevegelsesretningen til legemet.

I tillegg vet vi at kraftsummen (resultantkraften) finnes som

$$\Sigma F = m * a$$

som gir en akselerasjon a på

$$a = \Sigma F / m$$

For hvert tidssteg Δt som går mens treklossen sklir bortover bordflaten, beregnes treklossens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(t)*dt), ny posisjon er da (s + v*dt).

Verdiene av alle tidspunktene t der det foretas beregninger av treklossens posisjon s, og momentanfart v treklossen har, registreres fortløpende. Tidspunktene t og posisjonen s til treklossen, legges til i liste (t_verdier og s_verdier) under forløpet, mens variabelen v som holder rede på momentanfarten oppdateres gjennom forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen (strekning som funksjon av tiden) skal plottes, og sluttverdien av variabelen s og t brukes for å uttrykke distansen som er tilbakelagt og tiden det har tatt til treklossen har stoppet.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes da en while-løkke som løper så lenge momentanfarten v er større enn 0. Dette tilsvarer perioden det tar treklossen å bremse ned (retardere) fra startfarten v = 2,3, til sluttfarten v_slutt = 0.

5.5.6.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere treklossens ferd frem til den stopper helt opp grunnet friksjonskraften.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. m = 0.250, for massen til treklossen i gram
    b. g = 9.81, for tyngdeakselerasjonen, m/s^2
    c. mu = 0.3, for friksjonstallet mellom to flater av trevirke
    d. G = m * g, uttrykk for gravitasjonskraften
    e. N = G, for normalkraften
    f. R = mu * N, uttrykk for friksjonskraften
    g. sum_F = -R, uttrykk for netto kraftsum
    h. a = sum_F / m, uttrykk for treklossens netto akselerasjon
    i. s = 0, for startposisjonen til treklossen
    j. v = 2.3, for startfarten til treklossen
    k. t = 0, for startverdi av tidsperioden som skal benyttes i beregning
    l. v_slutt = 0, for sluttfarten til treklossen når den stopper
    m. dt = 0.01, for tidsintervall mellom hver beregning
    n. s_verdier = [s], for liste over alle posisjonene treklossen befinner seg ved
    o. v_verdier = [v], for liste over alle fartene treklossen holder ved de ulike tidspunktene beregningene blir utført
    p. t_verdier = [t], for liste over alle tidspunkt beregningene av treklossens attributter blir utført.

3) Definere while-løkke som kjører så lenge øyeblikksfarten v > v_slutt. (Det vil si så lenge treklossen er i bevegelse). While-løkken står for:
    a. beregning av nye oppdaterte verdier av fart v, posisjon s og forløpt tid t, for hvert tidssteg dt.
    b. oppdatering av listene t_verdier og s_verdier med de nye verdiene, som skal benyttes til å plotte grafen. I tillegg oppdateres v_verdier med de nye verdiene for fart. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

4) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String-funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde verdiene til 3 desimaler.

    b) print (f'Treklossen skled en distanse på {round(s, 3)} meter, før den stoppet.')

    c) print (f'Treklossen skled i {round(t, 3)} sekunder, før den stoppet.')

5) Plotter de registrerte verdien i t_verdier og s_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere treklossens bevegelse som funksjon av tiden.
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title("Streng"), setter tittel på grafen
    c. xlabel("Streng"), setter beskrivende tekst til x-aksen
    d. ylabel("Streng"), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

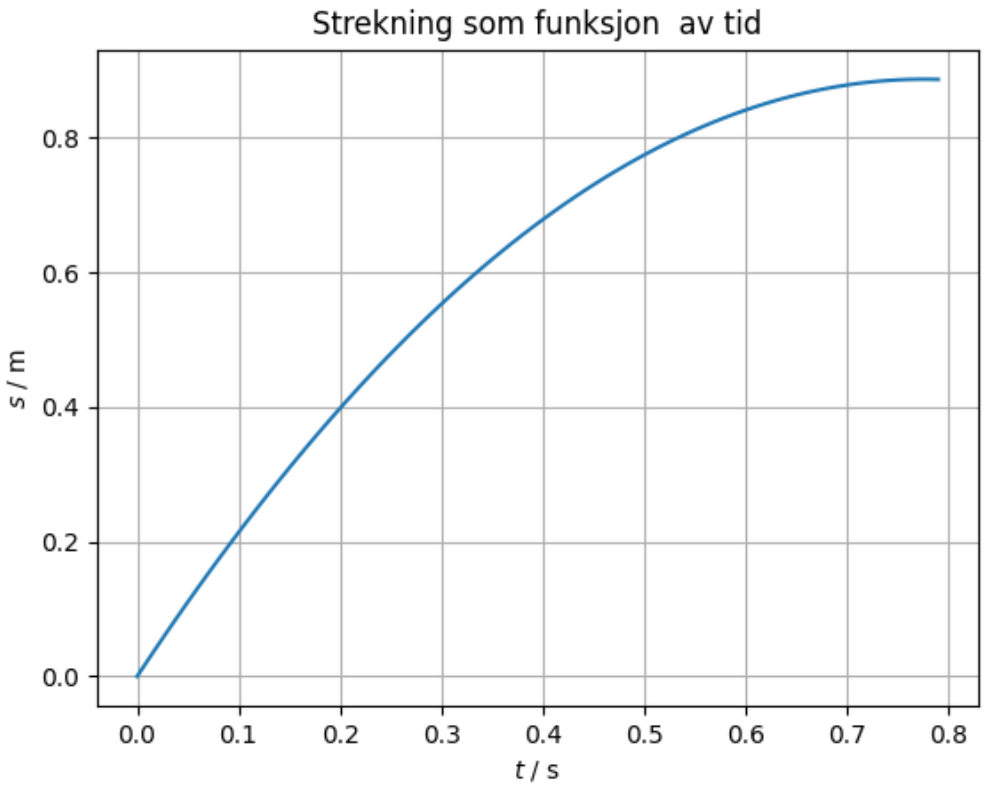
5.5.6.5 Program:

```python
# Konstante krefter

from pylab import *

# Informasjon om konstanter, krefter og gjenstanden
m = 0.250           # massen av gjenstanden, kg
g = 9.81            # tyngdeakselerasjonen, m/s^2
mu = 0.3            # friksjonstallet (tre/tre)
G = m * g           # gravitasjonskraft, m/s^2
N = G               # Normalkraft, N
R = mu * N          # friksjonskraft, N

sum_F = -R          # kraftsum, N
a = sum_F/m         # akselerasjon, m/s^2

# Startverdier for bevegelsen
s = 0               # startposisjon, m
v = 2.3             # startfart, m/s
t = 0               # starttid, s

#Simuleringsteknisk
v_slutt = 0         # sluttid for simuleringen, s
dt = 0.01           # lengden på tidsstegene, s
s_verdier = [s]     # liste for lagring av posisjon
v_verdier = [v]     # liste for lagring av fart
t_verdier = [t]     # liste for lagring av tid

# Løkke som regner ut ny fart, posisjon og tid
while v > v_slutt:  # så lenge legemet ikke har stoppet opp (v = 0)
    v = v + a * dt  # regner ut og oppdaterer ny v
    s = s + v * dt  # regner ut og oppdaterer ny s
    t = t + dt      # regner ut og oppdaterer ny t

    t_verdier.append(t) # legger til ny t-verdi i listen for t
    v_verdier.append(v) # legger til ny v-verdi i listen for v
    s_verdier.append(s) # legger til ny s-verdi i listen for s

# Utskrift av treklossens ferd over bordplaten, distanse og tid
print (f'Treklossen skled en distanse på {round(s, 3)} meter, før den stoppet.')
print (f'Treklossen skled i {round(t, 3)} sekunder, før den stoppet.')


# Tegning av graf
plot(t_verdier, s_verdier)                  # lager grafen
title("Strekning som funksjon av tid")      # tittel på grafen
xlabel("$t$ / s")                           # x-akse tittel
ylabel("$s$ / m")                           # y-akse tittel
grid()                                      # viser rutenett
show()                                      # viser grafen
```

Figur 5-150: Kode for oppgave 6 – Konstante krefter

5.5.6.6 Output:

```
Treklossen skled en distanse på 0.887 meter, før den stoppet.
Treklossen skled i 0.79 sekunder, før den stoppet.
```



5.5.7 Oppgave 7 – Ikke-konstante krefter

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al., 2021).

5.5.7.1 Kort beskrivelse av tema

En kraft, som gravitasjonskraften, påvirker et legeme likt over tid ettersom kraften er proporsjonal med objektets masse – en masse som i alle ordinære sammenhenger holder seg konstant. I motsetning til konstante krefter, vil andre typer krefter som eksempelvis luftmotstanden et objekt blir utsatt for under et fritt fall, endre seg over tid, ettersom kraften er direkte påvirket av objektets fart i et gitt øyeblikk. I slike situasjoner må den nye resultantkraften (nettokraft) beregnes for hver ny fart legemet har, og tilhørende akselerasjon i tidspunktet kan finnes og benyttes i utregningene.

5.5.7.2 Oppgave:

En basehopper kaster seg ut fra taket på et tårn, og faller fritt mot bakken. Startfarten $v_0 = 0$ m/s. Luftmotstandstallet er oppgitt til k = 0,32 kg/m. Basehopperens masse er m = 79 kg.

a) Plot en graf som viser basehopperens fart som funksjon av tiden.
b) Bestem hvor lang tid det går før basehopperen oppnår maksimal fart (terminalfarten).

5.5.7.3 Problemløsning/Problemløsnings-strategi:

Retningen ned mot bakken velges som positiv. Kreftene som virker på basehopperen, er da gravitasjonskraften G og luftmotstanden L:

$$\Sigma F = G - L$$

eller

$$\Sigma F = m*g - k * v^2$$

For hvert tidssteg Δt som går mens basehopperen faller gjennom luften, beregnes basehopperens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(v)*dt), ny posisjon er da (s + v*dt).

Verdiene av alle tidspunktene t der det foretas beregninger av basehopperens posisjon s, og momentanfart v basehopperen har, registreres fortløpende. Tidspunktene t og momentanfarten v til basehopperen, legges til i liste (t_verdier og v_verdier) under forløpet, mens variabelen s som holder rede på strekningen oppdateres gjennom forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen (fart som funksjon av tiden) skal plottes, og sluttverdien av variabelen v og t brukes for å uttrykke terminalfarten som er oppnådd og tiden det har tatt basehopperen å nå terminalfarten.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes da en while-løkke som løper så lenge akselerasjonen til basehopperen er større enn 0, det vil si at basehopperens hastighet øker.

Ettersom luftmotstanden L er avhengig av farten v i et gitt tidspunkt, vil akselerasjonen (som er gitt som a = ΣF/m) også variere når momentanfarten v varierer. For å oppdatere og beregne en fortløpende ny verdi av akselerasjonen a, benyttes en egen funksjon a(v) for å utføre beregningen av akselerasjon basert på momentanfarten v.

$$a(v) = (G - L) / m$$

eller

$$a(v) = (m*g - k * v^2) / m$$

5.5.7.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere basehopperens ferd gjennom luften til han når terminalfarten.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. m = 79, for massen til basehopperen i kg
    b. g = 9.81, for tyngdeakselerasjonen, m/s^2
    c. k = 0.32, for luftmotstandstallet, kg/m
    d. G = m * g, uttrykk for gravitasjonskraften
    e. s = 0, for startposisjonen til basehopperen
    f. v = 0, for startfarten til basehopperen
    g. t = 0, for startverdi av tidsperioden som skal benyttes i beregning
    h. dt = 0.01, for tidsintervall mellom hver beregning
    i. s_verdier = [s], for liste over alle posisjonene basehopperen befinner seg ved
    j. v_verdier = [v], for liste over alle fartene basehopperen holder ved de ulike tidspunktene beregningene blir utført
    k. t_verdier = [t], for liste over alle tidspunkt beregningene av basehopperens attributter blir utført.

3) Definere funksjonen a(v) som kalkulerer og returnerer akselerasjonen, gitt verdien av momentanfarten v sendt inn som argument til funksjonen.
    a. Oppretter lokal variabel L som beregner verdien til luftmotstanden gitt en momentanfart v, og luftmotstandstallet k
        L = k * v^2
    b. Oppretter lokal variabel sum_F som beregner verdien til kraftsummen gitt gravitasjonskraften G, og luftmotstanden L
        sum_F = G - L
    c. Oppretter lokal variabel aks for lagring av beregnet verdi for akselerasjon
        aks = sum_F / m
    d. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge akselerasjonen a > 0. (Det vil si så lenge basehopperens fart øker). While-løkken står for:
    a. beregning av nye oppdaterte verdier av fart v, posisjon s og forløpt tid t, for hvert tidssteg Δt.
    b. oppdatering av listene t_verdier og v_verdier med de nye verdiene, som skal benyttes til å plotte grafen. I tillegg oppdateres s_verdier med de nye verdiene for tilbakelagt strekning. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

5) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde verdiene til 3 desimaler.

    b) print (f'Basehopperen svevde gjennom luften i {round(t, 3)} sekunder før terminalhastigheten {round(v, 3)} m/s ble nådd.')

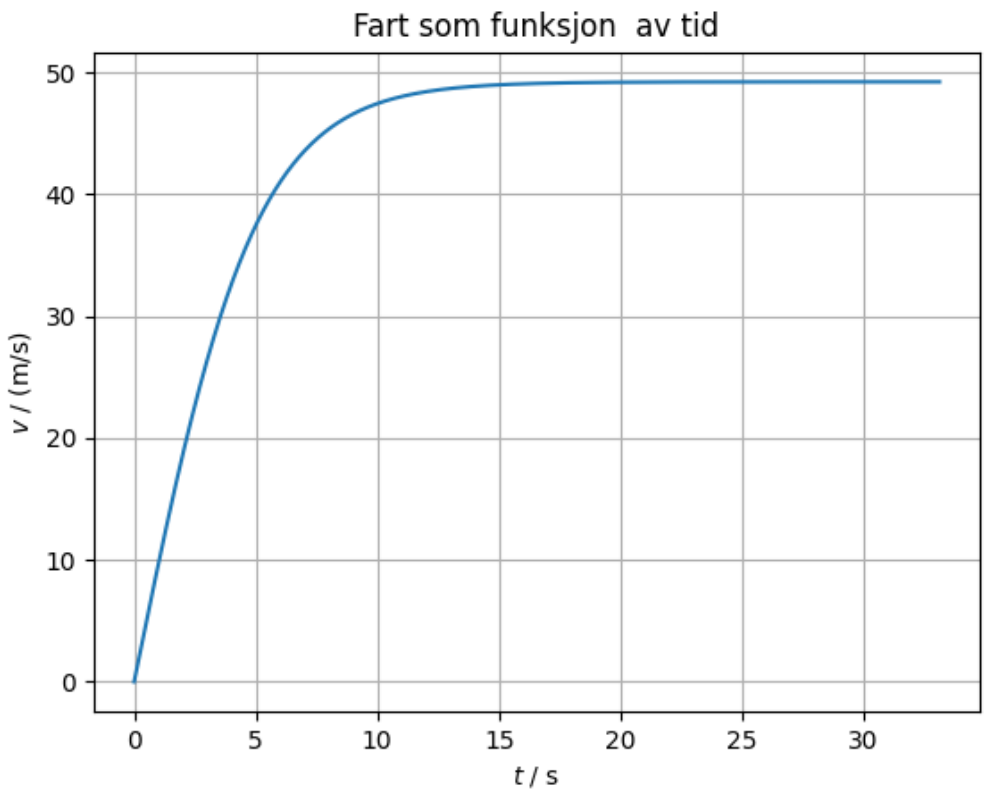
6) Plotter de registrerte verdien i t_verdier og v_verdier som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere basehopperens fart som funksjon av tiden.
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title("Streng"), setter tittel på grafen
    c. xlabel("Streng"), setter beskrivende tekst til x-aksen
    d. ylabel("Streng"), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

5.5.7.5 Program:

```python
# Ikke-konstante krefter

from pylab import *

# Informasjon om konstanter, krefter og gjenstanden
m = 79                      # massen av gjenstanden, kg
g = 9.81                    # tyngdeakselerasjonen, m/s^2
k = 0.32                    # luftmotstandstall, kg/m
G = m * g                   # gravitasjonskraft, N

def a(v):                   # akselerasjonen er en funksjon av farten
    L = k * v**2            # luftmotstand, N
    sum_F = round(G - L, 2) # kraftsum, N
    aks = sum_F/m           # akselerasjon, m/s^2
    return aks

# Startverdier for bevegelsen
s = 0               # startposisjon, m
v = 0               # startfart, m/s
t = 0               # starttid, s

#Simuleringsteknisk
dt = 0.01           # lengde på tidssteg, s
s_verdier = [s]     # liste for lagring av posisjon
v_verdier = [v]     # liste for lagring av fart
t_verdier = [t]     # liste for lagring av tid

# Løkke som regner ut ny fart, posisjon og tid
while a(v) != 0:        # så lenge akselerasjonen er større enn 0 (ikke lik 0)
    v = v + a(v) * dt   # regner ut ny v og oppdaterer
    s = s + v * dt      # regner ut ny s og oppdaterer
    t = t + dt          # regner ut ny t og oppdaterer

    t_verdier.append(t) # legger til ny t-verdi i listen for t
    v_verdier.append(v) # legger til ny v-verdi i listen for v
    s_verdier.append(s) # legger til ny s-verdi i listen for s

# Tegning av graf
plot(t_verdier, v_verdier)                  # lager grafen
title("Fart som funksjon av tid")           # tittel på grafen
xlabel("$t$ / s")                           # x-akse tittel
ylabel("$v$ / (m/s)")                       # y-akse tittel
grid()                                      # viser rutenett
show()                                      # viser grafen

# Utskrift av basehopperens ferd gjennom luften,
# terminalhastighet og tid
print(f'Basehopperens terminalhastighet: {round(v, 3)} m/s')
print(f'Terminalhastigheten ble nådd etter {round(t, 3)} sekunder.')
```

Figur 5-151: Kode for oppgave 7 – Ikke-konstante krefter

5.5.7.6 Output:




```
Basehopperens terminalhastighet: 49.212 m/s
Terminalhastigheten ble nådd etter 33.11 sekunder.
```

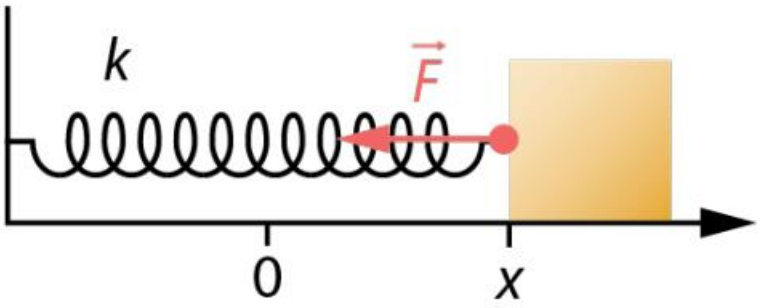
5.5.8 Oppgave 8 – Hookes lov

Inspirasjon og bilde til denne oppgaven er hentet fra nettsiden til Cappelen Damm (Cappelen Damm).

5.5.8.1 Kort beskrivelse av tema

Hookes lov sier at fjærkraften fra en fjær som er strukket en lengde x, er F = kx, der k er fjærstivheten til fjæra målt i N/m.

Vi setter x = 0 der fjæra ikke er strukket. Hvis fjæra er sammenpresset, blir x negativ og F peker i motsatt retning.



5.5.8.2 Oppgave:

En fjær med fjærkonstant k = 2,3 N/m er festet til en vegg. I den andre enden av fjæra er det en kloss med massen m = 0,250 kg. Vi trekker klossen ut til x = 0,20 m og slipper, slik at klossen glir fram og tilbake. Først ser vi bort fra all friksjon.

a) Skriv et program som beregner numerisk hvordan posisjonen varierer de fem første sekundene etter at vi slipper klossen, og lag en posisjonsgraf. Forklar grafen du får.

b) Utvid programmet slik at det også tar hensyn til friksjonen mellom klossen og underlaget. Friksjonstallet er μ = 0,02.

HINT: Friksjonen peker alltid i motsatt retning av fartsretningen. Du kan bruke np.sign() til å finne fortegnet til farten.

5.5.8.3 Problemløsning/Problemløsnings-strategi:

Når klossen slippes fra utgangspunktet x = 0,20m fra likevektspunktet, settes klossen i bevegelse. Ettersom legemet utsettes for akselerasjon gjennom kraften fra fjæren, må det fortløpende beregnes ny momentantfart ved de ulike tidsstegene, i tillegg til ny posisjon, samt oppdatert tidspunkt for hvert nytt tilbakelagt tidssteg Δt. Vi beregner verdier i et tidsrom på 5 sekunder etter at klossen blir sluppet. Nettokraften klossen utsettes for under bevegelsen er gitt som

$$F = k * x[i],$$

der x[i] representerer klossens posisjon ved beregning nummer i (tidssteg i). En beregner fortløpende verdien av F for hver ny posisjon x[i], og avleder akselerasjonen i det gitte punkt x[i] som

$$a[i] = -F / m$$

Ny fart etter at tidssteget i har passert, er da (v[i+1] = v[i] + a[i]*dt), ny posisjon er da

(x[i+1] = x[i] + v[i]*dt), og nytt tidspunkt (t[i+1] = t[i] + dt).

NB! Legg merke til at det er de ulike verdiene tilsvarende element (i+1) i rekken som registreres. Det betyr at en må ta hensyn til øverste grense av arrayet slik at en ikke prøver å legge til en verdi til en celle som ikke eksisterer i arrayet. Antallet beregninger reduseres da til (n-1). (Se punkt 3 under Algoritmer)

Verdiene av legemets posisjon x, og tiden t legemet har vært i bevegelse, registreres fortløpende i hvert sitt array (matrise) under forløpet. Disse arrayene med verdier benyttes som datagrunnlag når punktene som utgjør grafen skal plottes.

For å beregne nye verdier for hvert tidssteg i, benyttes en for-løkke som løper fra tidssteg i = 0, til tidssteg i = (n-1) er oppnådd. Grafen til legemets bevegelse plottes til slutt ved å benytte funksjonen pyplot som importeres via biblioteket matplotlib.

b) Oppgaven utvides til nå også å ta hensyn til klossens friksjon mot underlaget. Klossen påvirkes nå dermed av fjærkraften F som tidligere, samt friksjonskraften R, gitt som

$$R = my * m * g,$$

der my er friksjonstallet $\mu = 0.02$

Friksjonskraften vil alltid virke i motsatt retning av klossens bevegelse, så en benytter momentanfartens verdi v[i] til å bestemme hvilken verdi som skal beregnes for friksjonskraften R.

Uttrykket for akselerasjonen for hvert tidssteg, når en tar hensyn til både fjærkraften og friksjonskraften, kan da uttrykkes som

$$a[i] = (-F - R) / m, \quad \text{for } v[i] >= 0$$

$$a[i] = (-F + R) / m, \quad \text{for } v[i] < 0$$

5.5.8.4 Algoritme:

Oppgave a)

1) Importere matplotlib, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere legemets bevegelse i løpet av tidsrommet. I tillegg importeres biblioteket NumPy som brukes for å arbeide med arrayer og numeriske beregninger.

   Det opprettes 2 lokale variabler np og plt i forbindelse med import av nevnte bibliotek. Disse variablene kalles aliaser og brukes som forkortelser for å slippe å skrive hele navnet til biblioteket hver gang en ønsker å benytte det i koden. En kan navngi disse aliasene hva en ønsker (av gyldige uttrykk i Python), men det er kutyme å benytte standardnavnene np og plt slik at andre programmere lett kjenner igjen bruken og betydningen av disse.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. T = 5, for tiden i sekunder en ønsker å foreta målinger
   b. n = 10000, for antall tidssteg en ønsker for beregninger
   c. dt = T / n, for tidsintervall mellom hver beregning
   d. t = np.linspace(0, T, n), for liste over alle tidspunkt der det blir gjort beregninger

      ```
      linspace(start, stopp, n)
      ```

      er en funksjon som tilbys av numpy-biblioteket. Funksjonen tar et intervall fra start til stopp og deler inn i n like deler.

   e. x = np.zeros(n) , for liste over alle posisjoner legemet befinner seg ved de ulike tidspunktene beregningene blir utført.

      ```
      zeros(n)
      ```

      er en funksjon som tilbys av numpy-biblioteket og gjøres tilgjengelig gjennom aliaset (variabelen) np. Funksjonen oppretter et array av lengde n, og fyller hvert element med verdien 0.

   f. v = np.zeros(n) , for liste over alle momentanfartene legemet har ved de ulike tidspunktene beregningene blir utført.
   g. a = np.zeros(n-1) , for liste over alle akselerasjonsverdiene legemet har ved de ulike tidspunktene beregningene blir utført.
   h. m = 0.250, for massen i kg
   i. g = 9.81, for gravitasjonskonstanten
   j. k = 2.3, for fjærens stivhet i N/m
   k. x[0] = 0.20, for startverdi for klossens startposisjon

3) Definere for-løkke som kjører for alle definerte verdier av n. (Det vil si for alle de 10.000 tidsintervallene det vil bli utført beregninger i dette tilfellet). For å beskrive antallet iterasjoner (gjentakelser) av for-løkken benyttes formuleringen

   for i in range(n-1),

der i er en lokal variabel som kun opprettes for bruk i for-løkken, og som har som oppgave å holde rede på hvilken iterasjon en er kommet til. Altså hvilket nummer en er kommet til av alle de 10.000 gangene for-løkken skal gjentas i dette tilfellet.

Funksjonen range() benyttes for nettopp å beskrive hvor mange ganger løkken skal gjentas. Ved første gjennomgang, blir variabelen i satt til 0, og programkoden inne i for-løkken kjøres. Verdien (argumentet) inne i range (sluttverdi) sier når en skal slutte å gjenta løkken. Så når i har økt fra 0 til sluttverdien (n-1), slutter gjentakelsene av koden inne i for-løkken, og programmet går videre.

Ettersom beregningen av de ulike verdiene for hvert tidssteg i legges til i celle (i+1) vil det å benytte verdien n (som er 10.000) resultere i at programmet prøver å legge siste verdi i celle 9.999 + 1, som ikke eksisterer i arrayet. (Arrayet går fra [0, 9.999]). Derfor settes rekkevidden til funksjonen range til (n-1), og ikke n.

for-løkken sørger for oppdatering av arrayene ved

a. beregning av verdi for kraften F, ved hjelp av formelen F = k * x[i]
b. beregning av verdi for akselerasjonen a[i], ved hjelp av formelen a[i] = -F/m
c. beregning av verdi for neste momentanfart v[i+1], ved hjelp av formelen v[i] + a[i]*dt
d. beregning av verdi for neste posisjon x[i+1], ved hjelp av formelen x[i] + v[i]*dt
e. beregning av verdi for neste tidsintervall t[i+1], ved hjelp av formelen t[i] + dt

4) Plotter de registrerte verdien i t og x som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pyplot-biblioteket

a. plt.plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
b. plt.title("Streng"), setter tittel på grafen
c. plt.xlabel("Streng"), setter beskrivende tekst til x-aksen
d. plt.ylabel("Streng"), setter beskrivende tekst til y-aksen
e. plt.grid(), tegner inn rutenett på figuren
f. plt.show(), funksjon som tegner grafen til skjerm

Tillegg oppgave b)
Algoritmen følger i stor grad løsningsforslaget under a). Kun tilleggene/endringene listes opp her.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   l. my = 0.02, for friksjonstallet
   m. R = my * m * g, for beregning av friksjonskraften

3) For-løkken som kjører for alle definerte verdier av n.
   b. Beregning av oppdatert verdi for akselerasjon a[i] som også tar hensyn til friksjonskraften R i beregningen, ved hjelp av formelen

a[i] = (-F -R * np.sign(v[i])) / m

`np.sign(val)`

er en funksjon som tilbys av numpy-biblioteket og gjøres tilgjengelig gjennom aliaset (variabelen) np. Funksjonen tar inn en verdi val som argument og returnerer -1 om val <0, 0 om val = 0, og +1 om val > 0.

5.5.8.5 Program:
Oppgave a)

```python
# Hookes lov - Fjærkraft

# Importerer tilleggsbibliotek.
import numpy as np
import matplotlib.pyplot as plt

# Hvor lenge vi skal simulere.
T = 5 #s

# Antall tidssteg, må være mange nok til å gi en god nok beregning.
n = 10000

# Lengden av hvert tidssteg.
dt = T / n

# Lager en liste med tidsverdier.
t = np.linspace(0, T, n)

# Lager tomme lister for posisjon, fart og akselerasjon.
x = np.zeros(n)
v = np.zeros(n)
a = np.zeros(n-1)

# Definerer konstanter og startverdier.
m = 0.250 #kg
g = 9.81 #m/s^2
k = 2.3 #N/m
x[0] = 0.20 #m

# Simulerer bevegelsen.
for i in range(n-1):
    F = k*x[i]
    a[i] = -F/m
    v[i+1] = v[i] + a[i]*dt
    x[i+1] = x[i] + v[i]*dt
    t[i+1] = t[i] + dt

#Lager og tegner posisjonsgrafen.
plt.plot(t, x)
plt.title("Kloss i fjær")
plt.xlabel("$t$ / s")
plt.ylabel("$x$ / m")
plt.grid()
plt.show()
```

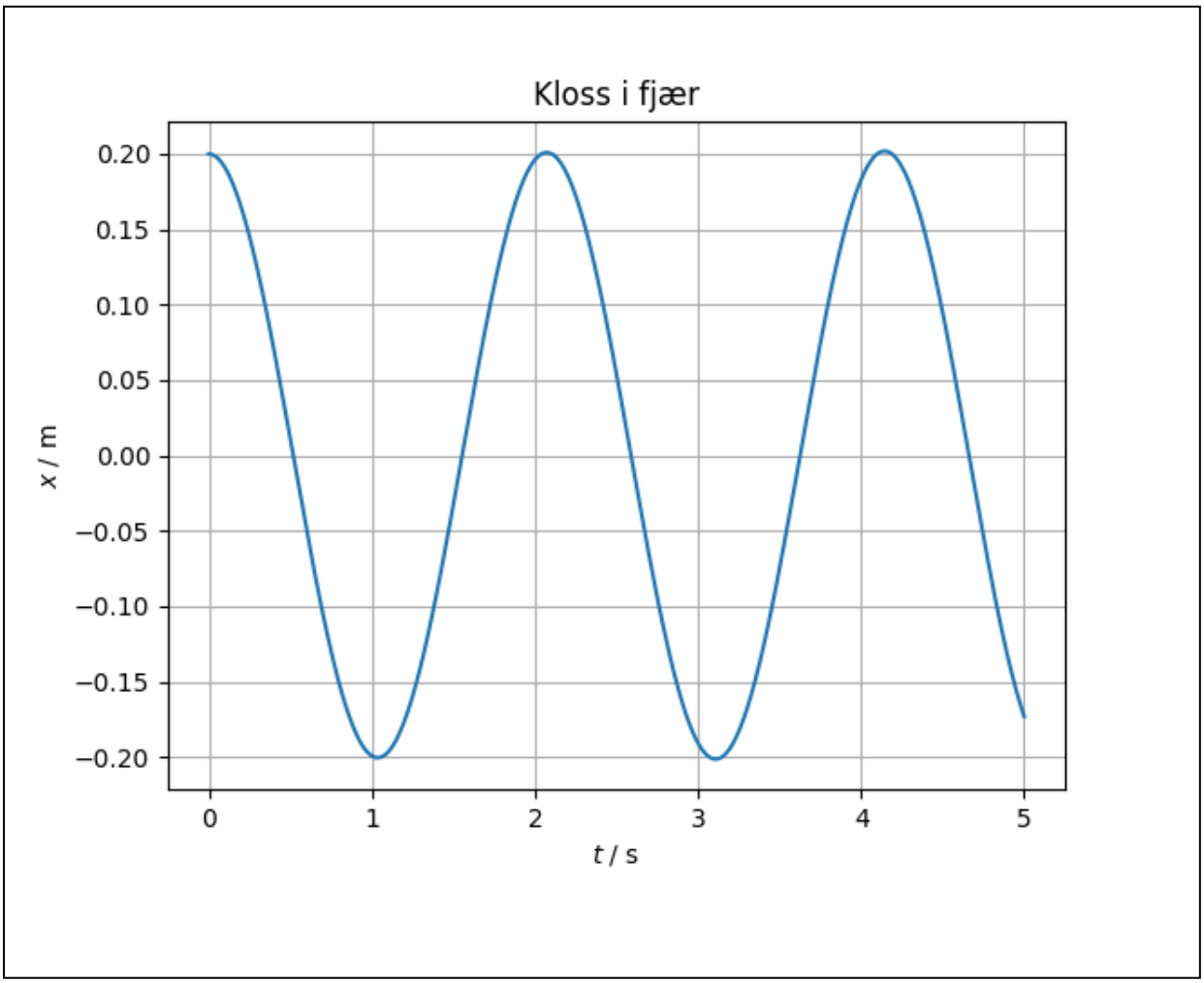
Figur 5-152: Kode for oppgave 8 a – Hookes lov

Oppgave b)

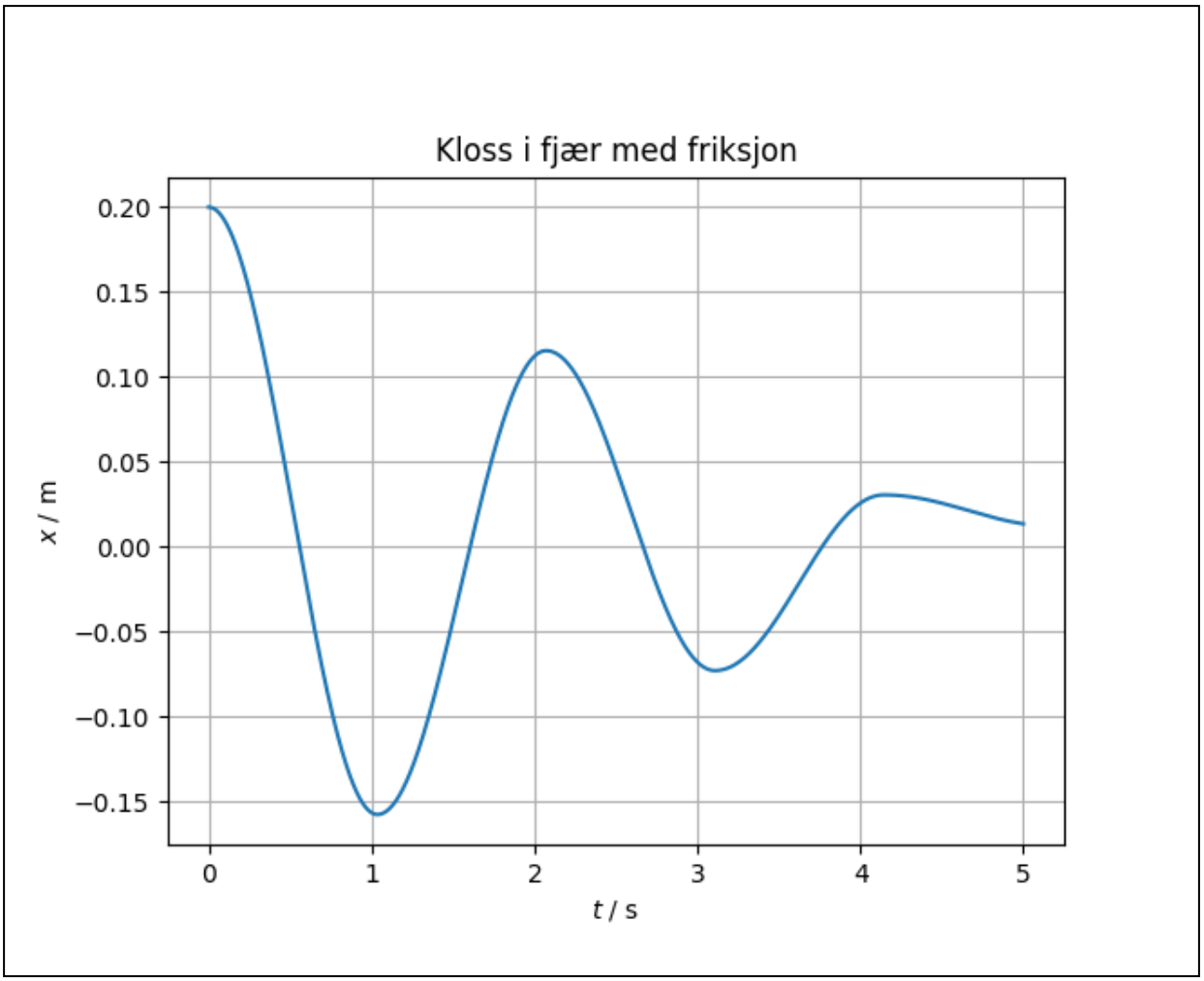
```python
# # Hookes lov - Fjærkraft

#Importerer tilleggsbibliotek.
import numpy as np
import matplotlib.pyplot as plt

#Hvor lenge vi skal simulere.
T = 5 #s

#Antall tidssteg, må være mange nok til å gi en god nok beregning.
n = 10000

#Lengden av hvert tidssteg.
dt = T / n

#Lager en liste med tidsverdier.
t = np.linspace(0, T, n)

#Lager tomme lister for posisjon, fart og akselerasjon.
x = np.zeros(n)
v = np.zeros(n)
a = np.zeros(n-1)

#Definerer konstanter og startverdier.
m = 0.250 #kg
g = 9.81 #m/s^2
k = 2.3 #N/m
x[0] = 0.20 #m
my = 0.02 # friksjonstallet

R = my * m * g # Friksjonskraften

#Simulerer bevegelsen.
for i in range(n-1):
    F = k*x[i]
    a[i] = ( -F - R * np.sign(v[i]) ) / m
    v[i+1] = v[i] + a[i]*dt
    x[i+1] = x[i] + v[i]*dt
    t[i+1] = t[i] + dt


#Lager og tegner posisjonsgrafen.
plt.plot(t, x)
plt.title("Kloss i fjær med friksjon")
plt.xlabel("$t$ / s")
plt.ylabel("$x$ / m")
plt.grid()
plt.show()
```

Figur 5-153: : Kode for oppgave 8 b – Hookes lov

5.5.8.6 Output:
Oppgave a)



Oppgave b)


Kloss i fjær med friksjon
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
x / m
0
1
2
3
4
5
t / s

5.5.9 Oppgave 9 – Bohrs atommodell

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 1 (Callin et al., 2021, s. 326).

5.5.9.1 Kort beskrivelse av tema

Bohrs postulater gir viktig informasjon om et atoms energitilstand ettersom et elektron i atomet kan endre nivå eller "bane". Et atom kan få tilført energi (absorpsjon), eller avgi energi (emisjon). En slik endring i atomets energitilstand kommer til uttrykk gjennom at atomets elektron går fra et energinivå til et annet. Modellen sier at energien som tilføres må være stor nok til at elektronet kan skifte nivå, det vil si at kvantumet av energi må være tilstrekkelig for å nå det neste nivået. De ulike nivåene ("banene") et elektron kan inneha tilsvarer da en endring av konkrete, faste energinivå – kvanter.

Bohrs første postulat sier

> "Et atom kan være i en av mange ulike energitilstander uten å miste energi. I hver tilstand har atomet en helt bestemt energi,
>
> $$E_1, E_2, \dots, E_n \text{ der } n \in \{1, 2, 3, \dots\}$$"

(Callin, Dokka, Hellesøy, Seland, & Skåland, 2021, s. 326)

Bohrs andre postulat sier

> "Et atom kan gå fra én tilstand med energien En til en annen tilstand med lavere energi E_m. Ved overgangen blir energidifferansen sendt ut som et foton med energien $E = hf$, der
>
> $$hf = E_n - E_m \text{ , } n > m$$
>
> hvor h er Plancks konstant og f er strålingsfrekvensen.
>
> Tilsvarende kan et atom i en lav energitilstand E_m gå til en høyere energitilstand E_n, gjennom å absorbere energimengden som tilsvarer differansen $E_n - E_m$"

(Callin, Dokka, Hellesøy, Seland, & Skåland, 2021, s. 326)

5.5.9.2 Oppgave:

Lag et program som beregner og skriver ut de laveste energinivåene hydrogenatomet kan oppnå.

5.5.9.3 Problemløsning/Problemløsnings-strategi:

Basert på de generelle postulatene for atomer laget Bohr også en spesiell modell for energinivåene i hydrogenatomet spesifikt.

> "Energinivåene i hydrogenatomet er gitt ved
>
> $$En = -B / n^2 \text{ , } n \in \{1, 2, 3\dots\}$$
>
> hvor B er Bohrs konstant, $B = 2{,}18 * 10^{-18}$ J = 2.18aJ."

Det er den samme formelen som benyttes for beregning av energi ved de ulike energinivåene (banene), og som formelen for energinivåene i hydrogenatomet over viser, refereres nivåene n til som heltall fra 1 og oppover, n E {1, 2, 3…}. Det vil følgelig være naturlig å foreta beregningene i en for-løkke som gjentar beregningen av det aktuelle nivået n, skriver ut dette resultatet, for deretter å avansere til neste verdi av n, så gjentas prosessen til alle nivå n er beregnet og skrevet ut.

5.5.9.4 Algoritme:

1) Benytter funksjonen print() som er innebygget i Python til å presentere en overskrift til brukeren.
2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregningen av hydrogenatomets energinivåer.
    a. B = 2.18, for Bohrs konstant oppgitt i aJ
3) Definere for-løkke som kjører for alle definerte verdier av n. (Det vil si for energinivå 1-6 i dette tilfellet). For å beskrive antallet iterasjoner (gjentakelser) av for-løkken benyttes formuleringen

    for n in range(1, 7),

    der n er en lokal variabel som kun opprettes for bruk i for-løkken, og som har som oppgave å holde rede på hvilken iterasjon en er kommet til. Altså hvilket nummer en er kommet til av alle gangene for-løkken skal gjentas.

    Funksjonen range() benyttes for nettopp å beskrive hvor mange ganger løkken skal gjentas. Første verdien i range(startverdi, sluttverdi) gir startverdien, slik at ved første gjennomgang, blir variabelen n satt til 1, og programkoden inne i for-løkken kjøres. Den andre verdien i range (startverdi, sluttverdi) sier hva når en skal slutte å gjenta løkken. Så når n har økt fra n=1 til n=sluttverdi, slutter gjentakelsene av koden inne i for-løkken, og programmet går videre.

    For-løkken sørger for:

    a. beregning av verdi for Energinivået E, ved hjelp av formelen E = -B /(n**2)
    b. den beregnede verdien E avrundes til 3 desimaler ved bruk av round()-funksjonen
    c. den beregnede verdien E skrives deretter ut til skjerm ved bruk av print()-funksjonen

        print("E_", n, "=", E, "aJ")

5.5.9.5 Program:

```python
# Bohrs atommodell

print("Energinivået i hydrogenatomet: ")

B = 2.18 #Bohrs konstant i aJ

for n in range(1, 7):       # løkke der n går fra 1 til 6
    E = -B / (n**2)         # regner ut energinivået fra Bohrs formel
    E = round(E, 3)         # runder av til 3 desimalar

    # Skriver ut resultatet
    print(" E_", n, "=", E, "aJ")
```

Figur 5-154: Kode for oppgave 9 – Bohrs atommodell

5.5.9.6 Output:

```
Energinivået i hydrogenatomet:
E_ 1 = -2.18 aJ
E_ 2 = -0.545 aJ
E_ 3 = -0.242 aJ
E_ 4 = -0.136 aJ
E_ 5 = -0.087 aJ
E_ 6 = -0.061 aJ
```

5.6 Oppgavehefte i Python for VG3

5.6.1 Oppgave 1 – Skråplan med friksjon

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin et al., Ergo Fysikk 2, 2022).

5.6.1.1 Kort beskrivelse av tema

Når et objekt beveger seg langs et skråplan påvirkes det av flere ulike krefter. Det kan være krefter som gravitasjon, friksjon mot underlaget, og luftmotstand. Friksjon og luftmotstand regnes som motsatt rettet av fartsretningen til objektet, mens tyngdekraften virker normalt (det vil si 90 grader) ned mot jordens sentrum. Det betyr at gravitasjonskraften må dekomponeres slik at en finner den komponenten som virker langs objektets fartsretning.

5.6.1.2 Oppgave:

En kloss med massen 400 g beveger seg oppover et langt skråplan uten luftmotstand. Startfarten til klossen en 3,20 m/s oppover skråplanet, helningsvinkelen til skråplanet $\theta = 30$ grader (θ = theta), og friksjonstallet mellom klossen og underlaget er $\mu = 0{,}35$ (μ = mu).

Lag et program som tegner fartsgrafen til klossen inntil den kommer tilbake til utgangspunktet.

5.6.1.3 Problemløsning/Problemløsnings-strategi:

Klossen starter sin ferd oppover skråplanet fra posisjon s med en startfart v = 3,20 m/s. For hvert tidssteg Δt som går mens klossen sklir oppover skråplanet før den snur og sklir ned igjen mot startposisjonen, beregnes klossens fart v, posisjon s og akselerasjon a på det enkelte tidspunktet.

Ny fart etter at tidsintervallet Δt har passert, er da (v + a(v)*dt), ny posisjon er da (s + v*dt).

Verdiene av klossens posisjon s, og momentanfarten v klossen har ved hvert tidssteg, registreres fortløpende i hver sin liste (s_verdier og v_verdier) under forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen skal plottes.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes en while-løkke som løper så lenge klossens posisjon er større enn klossens startposisjon, det vil si s >= 0. Når klossen igjen er ved posisjonen s=0 har den returnert til utgangspunktet.

Det må også beregnes ny akselerasjon for hvert tidssteg, og til å gjøre det defineres en egen funksjon a(v) som har som oppgave å kalkulere og returnere akselerasjonen som

påvirker klossen ved de ulike tidsstegene underveis i forløpet. Funksjonen beregner først friksjonskraften R klossen utsettes for

$$R = - (v * my * N), \quad \text{for } v >= 0$$

$$R = (v * my * N), \quad \text{for } v < 0$$

I tillegg innvirker gravitasjonskomponenten Gx i fartsretningen også på klossen. Denne er gitt ved formelen

$$Gx = m * g * \sin(theta)$$

Deretter beregnes netto kraftsum sum_F som påvirker klossen

$$sum\_F = -Gx + R$$

som igjen benyttes til å utlede akselerasjonen i hvert tidssteg.

$$aks = sum\_F / m$$

5.6.1.4 Algoritme:

6) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere klossens fart etter hvert som klossen beveger seg oppover skråplanet, før den snur og sklir tilbake ned til startpunktet.

7) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. m = 0.400, for massen til klossen
   b. theta = radians(30), for skråplanets vinkel
      Benytter funksjonen radians(grad) for å regne om grader til radianer som Python utfører sine beregninger med. radians() funksjonen følger med Math biblioteket som er inkludert både i NumPy og pylab.
   c. my = 0.35, for friksjonstallet mellom klossen og underlaget
   d. g = 9.81, for tyngdeakselerasjonen
   e. Gx = m * g * sin(theta), for beregning av tyngdekraftens komponent langs skråplanet
   f. Gy = m * g * cos(theta), for beregning av tyngdekraftens normalkomponent på skråplanet
   g. N = Gy, som tilsvarer normalkraften på klossen
   h. S = 0, for startposisjonen til klossen
   i. v = 3.20, for startfarten til klossen
   j. t = 0, fort starttiden
   k. dt = 0.001, for tidsintervall mellom hver beregning
   l. v_verdier = [v], for liste over alle momentanfartene klossen holder ved de ulike tidspunktene beregningene blir utført
   m. s_verdier = [s], for liste over alle posisjoner s klossen befinner seg ved de ulike tidspunktene beregningene blir utført.
   n. t_verdier = [t], for liste over alle tidspunktene t de ulike beregningene blir utført.

8) Definere funksjonen a(v) som kalkulerer og returnerer akselerasjonen

a. Tar inn en verdi v for momentanfarten i øyeblikket
b. Oppretter lokal variabel R som beregner friksjonskraften ved hjelp av uttrykket

```
R = -sign(v) * my * N
```

der sign() er en funksjon som tilbys av numpy biblioteket og gjøres tilgjengelig gjennom import av pylab. Funksjonen tar inn en verdi val som argument og returnerer -1 om val <0, 0 om val = 0, og +1 om val > 0.

c. Oppretter lokal variabel sum_F som beregner netto kraftsum ved hjelp av uttrykket

```
sum_F = -Gx + R
```

d. Oppretter lokal variabel 'aks' for lagring av beregnet verdi for akselerasjon
e. Benytter uttrykket

```
aks = sum_F / m
```

for beregning av akselerasjonen

f. Returnerer verdien av variabelen aks

9) Definere while-løkke som kjører så lenge posisjonen s >= 0. Det vil si helt til klossen returnerer til startposisjonen. While-løkken står for
   a. beregning av nye oppdaterte verdier av fart v, posisjon s, og tidspunkt t for hvert tidssteg Δt.
   b. oppdatering av listene t_verdier og v_verdier med de nye verdiene, som skal benyttes til å plotte grafen. I tillegg registreres s_verdier på tilsvarende måte for å kortslutte while-løkken når s = 0. Verdiene legges til de eksisterende listene ved å benytte funksjonen append() som utvider den eksisterende listen, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. append() er en funksjon som alle lister i Python har innebygget.

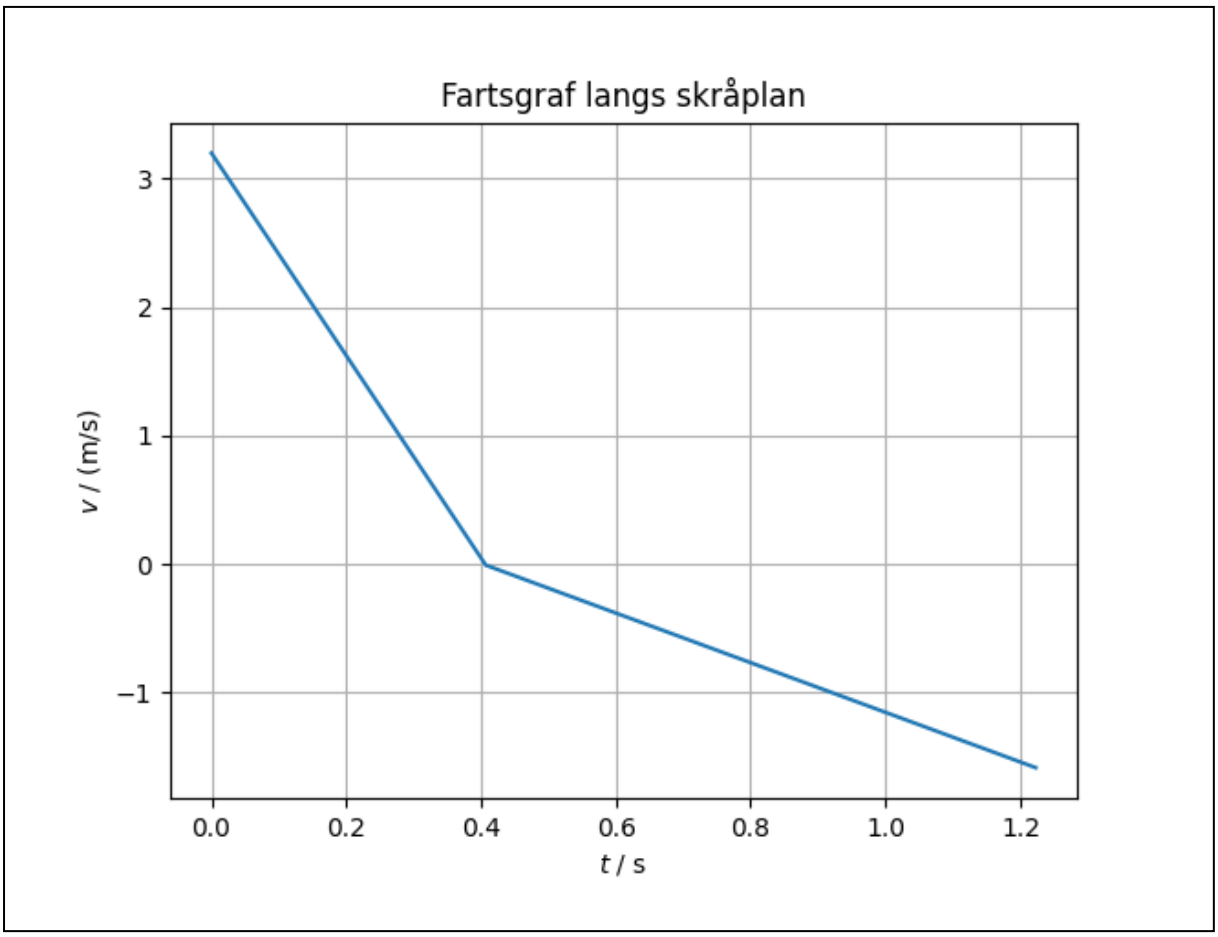
10) Plotter de registrerte verdien i t_verdier og v_verdier som punkter på en graf som skal fremstille farten til klossen som funksjon av tiden t, ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket
   a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
   b. title(«Streng»), setter tittel på grafen
   c. xlabel(«Streng»), setter beskrivende tekst til x-aksen
   d. ylabel(«Streng»), setter beskrivende tekst til y-aksen
   e. grid(), tegner inn rutenett på figuren
   f. show(), funksjon som tegner grafen til skjerm

5.6.1.5 Program:

```python
# Skråplan med friksjon

from pylab import *

# Konstanter
m = 0.400               # massen til gjenstanden, kg
theta = radians(30)     # vinkel for skråplanet
my = 0.35               # friksjonstall
g = 9.81                # tyngdeakselerasjon, m/s^2

# Konstante krefter
Gx = m*g*sin(theta)     # tyngdekraft langs skråplanet, N
Gy = m*g*cos(theta)     # tyngdekraft normalt på skråplanet, N
N = Gy                  # normalkraft på gjenstanden, N

# Variabe krefter, utregning av kraftsum og akselerasjon
def a(v):
    R = -sign(v)*my*N   # friksjonskraft, motsatt retning av fart
    sum_F = -Gx + R     # kraftsum, N
    aks = sum_F/m       # akselerasjon, m/s^2
    return aks

# Startverdier
s = 0                   # startposisjon, m
v = 3.20                # startfart, m/s
t = 0                   # starttid, s

# Lister for lagring av data
s_verdier = [s]
v_verdier = [v]
t_verdier = [t]

# Simulering av bevegelse
dt = 0.001              # tidssteg i simulering, s

while s >= 0:           # krav for stopp av simulering
    v = v + a(v)*dt     # regner ut ny fart
    s = s +v*dt         # regner ut ny posisjon
    t = t + dt          # går til neste tidspunkt

    # Lagring av verdier
    v_verdier.append(v)
    s_verdier.append(s)
    t_verdier.append(t)

# Tegning av graf
plot(t_verdier, v_verdier)          # lager grafen
xlabel("$t$ / s")                   # x-akse tittel
ylabel("$v$ / (m/s)")               # y-akse tittel
title("Fartsgraf langs skråplan")   # tittel på grafen
grid()                              # legger til rutenett
show()                              # viser grafen
```

Figur 5-155: Kode for oppgave 1 – Skråplan med friksjon

5.6.1.6 Output:



5.6.2 Oppgave 2 - Skrått kast med luftmotstand

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin et al., Ergo Fysikk 2, 2022).

5.6.2.1 Kort beskrivelse av tema

En kraft, som gravitasjonskraften, påvirker et legeme likt over tid ettersom kraften er proporsjonal med objektets masse – en masse som i alle ordinære sammenhenger holder seg konstant. I motsetning til konstante krefter, vil andre typer krefter som eksempelvis luftmotstanden et objekt blir utsatt for under en bevegelse, endre seg over tid, ettersom kraften er direkte påvirket av objektets fart i et gitt øyeblikk. I slike situasjoner må den nye resultantkraften (nettokraft) beregnes for hver ny fart legemet har, og tilhørende akselerasjon i tidspunktet kan finnes og benyttes i utregningene. Under et skrått kast er det da to krefter som påvirker baseballen – tyngdekraften og luftmotstanden. Tyngdekraften virker normalt (det vil si 90 grader) ned mot jorden, og luftmotstanden virker i motsatt retning av ballens fartsretning.

5.6.2.2 Oppgave:

En baseballspiller slår en ball som får en utgangsfart på 42,7 m/s i en vinkel 40 grader opp fra bakken. Baseball-ballen har luftmotstandstallet k = 1.31 * 10^{-3} kg/m og massen til ballen er m = 145 g. Ballen er 1,20 m over bakken når den treffes av balltreet.

a) Tegn en graf som illustrerer ballens bevegelse, etter den forlater balltreet. Plot ballens posisjon i x-retning langs x-aksen, og posisjonen i y-retning langs y-aksen av koordinatsystemet.

b) Finn den maksimale høyden ballen får over bakken i løpet av ferden.

c) En av utespillerne fanger ballen i en høyde 2,10 meter over bakken. Hvor langt i horisontal retning har ballen kommet fra startposisjonen da?

5.6.2.3 Problemløsning/Problemløsnings-strategi:

Oppgave a)

For å beskrive både ballens posisjon, fart og krefter i et gitt tidssteg, benyttes vektorer.

Eksempelvis kan en registrere vektoren $\vec{v}$ = [1,3] som et egen lite array, med de verdiene en trenger for det spesifikke øyeblikksbildet. Eksempelvis v = array([1,3]).

I Python kan en da benytte funksjonen norm() for å bestemme lengden av vektoren som er registrert i arrayet. l = norm(v)

Luftmotstanden peker alltid i motsatt retning av fartsretningen til ballen i øyeblikket. En enhetsvektor er en vektor som har lengde 1, og som har samme retning som den opprinnelige vektoren

Vektor $\vec{e}$ = vektor $\vec{v}$ / v

Dette brukes da for å finne retningen til vektoren $\vec{v}$ i det enkelte tidssteg det utføres ny beregning. For hvert tidssteg Δt som går mens baseball-ballen flyver gjennom luften, beregnes ballens fartvektor $\vec{v}$, posisjonsvektor $\vec{r}$, og akselerasjon $\vec{a}$ på det enkelte tidspunktet.

Ny fartsvektor etter at tidsintervallet Δt har passert, er da (v + a(v)*dt), ny posisjonsvektor er da (r + v*dt).

For alle tidspunktene t der det foretas beregninger av ballens posisjonsvektor $\vec{r}$, og fartsvektor $\vec{v}$ ballen har, registreres verdiene fortløpende. Posisjonsvektoren $\vec{r}$ og fartsvektoren $\vec{v}$ til ballen, legges til i listene r_liste og v_liste under forløpet. Disse listene med verdier benyttes som datagrunnlag når punktene som utgjør grafen (ballens fart i x-retning og y-retning) skal plottes.

For å beregne nye verdier for hvert tidsintervall Δt, benyttes da en while-løkke som løper så lenge posisjonsvektoren $\vec{r}$ (i y-retning) er større enn 0, altså frem til ballen har nådd bakkenivå igjen.

Ettersom luftmotstanden L er avhengig av farten v i et gitt tidspunkt, vil akselerasjonen (akselerasjonsvektoren) gitt som $\vec{a}$ = ΣF / m også variere.

For å oppdatere og beregne en fortløpende ny verdi av akselerasjonen a, benyttes en egen funksjon a(v) for å utføre beregningen av akselerasjon basert på fartsvektoren $\vec{v}$.

Akselerasjonsvektoren beregnes ved

$$\vec{a} = \Sigma F / a$$

der ΣF er gitt som vektorsummen (nettokraft) av alle kreftene som virker på ballen,

$$\Sigma F = G + L ,$$

der $\vec{G}$ er tyngdekraftsvektoren, og $\vec{L}$ er luftmotstandsvektoren.

Oppgave b) Finn den maksimale høyden ballen får over bakken i løpet av ferden.

Den maksimale høyden y over bakkenivå, kan finnes når farten i y-retning er 0. Det vil si vy = 0. Vy finnes i v-arrayet som andre indeks av hvert registrert element. Arrayet er bygget opp på følgende mal v = [(x-komponent, y-komponent)], og verdien av y-komponenten hentes ut ved hjelp av v[1]. (Indeksene på de ulike elementene i arrayet, starter på 0, og øker med 1. Slik at x-komponenten er å finne i v[0], og y-komponenten i v[1]).

Å bestemme når vy = 0, kan gjøres på flere ulike måter, men den mest trivielle fremgangsmåten er å stoppe gjennomkjøringen av while løkken, når ballen har nådd ballbanens topp,

```
while v[1] >= 0
```

denne løkken vil kjøre helt til vy-verdien = 0, og en kan lese høyden direkte ut av posisjonsvektor array-et r som høyde = r[1] som vil være den siste posisjonsvektoren som ble beregnet.

Oppgave c) Hvor langt fra startposisjonen har ballen kommet i x-retning når den fanges 2,10m over bakken (y-retning)?

En vet at ballen vil stige opp fra startposisjonen 1,20 m over bakken til makshøyden for deretter å dale ned mot bakken igjen. Det betyr at ballen vil nå høyden 2,10 m over bakken 2 ganger – først på vei oppover (situasjon 1), så på vei ned igjen (situasjon 2). Antar at det her er snakk om å finne den horisontale avstanden til ballen når den er på vei ned igjen (situasjon 2).

I den situasjonen vet en først og fremst at y-komponenten til posisjonen r skal være 2.10:

$$r[1] >= 2.1$$

Om en benytter dette uttrykket i while-løkken

```
while r[1] >= 2.1 :
```

vil ikke beregningen engang starte, siden startverdien til r[1] er 1.20 m, som er under grensen som er satt i while begrensningen > 2.1

Her trengs mer informasjon, og en benytter den ekstrainformasjonen at fartsvektoren v har ulikt fortegn ved de to situasjonene: På vei oppover har ballen en positiv y-komponent av fartsvektoren, mens på vei ned igjen etter å ha nådd toppunktet, har den en negativ y-komponent av fartsvektoren. Dermed kan en sette dette som tilleggskrav til å beskrive hvilken situasjon en ønsker å se på.

- Høyden skal være minst 2.1 over bakken ELLER farten i y-retning skal være positiv (for at while-løkken skal starte i det hele tatt)

Dette gir følgende betingelse som må til for at while-løkken skal kjøre helt til ballen er på vei ned igjen og når høyden 2.1m over bakken (situasjon 2)

```
while v[1] > 0 or r[1] >= 2.1 :
```

Nå vil while-løkken kjøre så lenge ballen er på vei oppover og har en positiv fartsvektor-komponent i y-retningen. Etter at ballen har nådd toppen av ballbanen, er fartsvektoren snudd til negativ og vilkår 1 i while-løkken gjelder ikke lenger, men da gjelder vilkår 2, siden r[1] er over 2.1 mens ballen daler nedover. Når ballen så kommer ned til høyden 2.1 vil while-løkken stoppe, i situasjon 2.

5.6.2.4 Algoritme
Oppgave a)

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere baseball-ballens ferd gjennom luften til den når bakken igjen.
   Importerer metoden concat fra biblioteket operator, som brukes for å sammenslå (concatenate) to lister til én.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. m = 0.145, for massen til ballen i kg
   b. g = 9.81, for tyngdeakselerasjonen, m/s^2
   c. k = 1.31*10^(-3), for luftmotstandstallet, kg/m
   d. v0 = 42.7, for startfart, m/s
   e. y0 = 1.2, for starthøyde, m
   g. theta = radians(40), for skråplanets vinkel i radianer
      Benytter funksjonen radians(grad) for å regne om grader til radianer som Python utfører sine beregninger med. radians() funksjonen følger med Math biblioteket som er inkludert både i NumPy og pylab.
   f. G = array([0, -m*g]), uttrykk for tyngden i N
      Benytter funksjonen array() som er gjort tilgjengelig gjennom biblioteket pylab, for å opprette et nytt array, og legger til verdiene (0, -m*g) som er x-komponent og y-komponent av tyngdekraften
   g. r = array([0, y0]), for startposisjonen til ballen i x- og y-retning
   h. v = array([v0 * cos(theta), v0 * sin (theta)]), for startfart i x- og y-komponent
   i. t = 0, for startverdi av tidsperioden som skal benyttes i beregning
   j. r_liste = [r], liste for lagring av de ulike posisjonsvektorene
   k. v_liste = [v], liste for lagring av de ulike fartsvektorene
   l. dt = 0.001, for tidssteg mellom hver beregning

3) Definere funksjonen a(v) som kalkulerer og returnerer akselerasjonsvektoren, gitt verdien av momentanfarten v sendt inn som argument til funksjonen.
   a. Oppretter lokal variabel for enhetsvektoren til farten v som mottas som input til funksjonen

      e_v = v / norm(v)

      der norm() er en funksjon som gjøres tilgjengelig via biblioteket numPy. (som er inkludert i pylab biblioteket).

   b. Oppretter lokal variabel for luftmotstandsvektoren L som beregner verdien til luftmotstanden gitt en fartsvektor v, og luftmotstandsvektoren k, samt enhetsvektoren e_v

      L = -k * norm(v)^2 * e_v

   c. Oppretter lokal variabel sum_F som beregner verdien til kraftsummen gitt gravitasjonskraften G, og luftmotstanden L

      sum_F = G + L

   d. Oppretter lokal variabel aks for lagring av beregnet verdi for akselerasjonsvektoren

      aks = sum_F / m

   e. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge posisjonsvektoren i y-retning r[1] >= 0. (Det vil si så lenge ballen er over bakkenivå). While-løkken står for
    a. beregning av nye oppdaterte verdier av fartsvektor v, posisjonsvektor r og forløpt tid t, for hvert tidssteg Δt.
    b. oppdatering av listene r_liste og v_liste med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listene ved å benytte funksjonen concatenate() som utvider den eksisterende listen som inneholder arrayer, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. Concatenate er en funksjon som er gjort tilgjengelig via biblioteket numpy (som er inkludert i pylab biblioteket).

5) Plotter de registrerte verdien i r_liste[:, 0] og r_liste[:, 1] som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere baseball-ballens bevegelse i x-retning og y-retning etter at den er truffet av balltreet.
    a. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
    b. title(«Streng»), setter tittel på grafen
    c. xlabel(«Streng»), setter beskrivende tekst til x-aksen
    d. ylabel(«Streng»), setter beskrivende tekst til y-aksen
    e. grid(), tegner inn rutenett på figuren
    f. show(), funksjon som tegner grafen til skjerm

Algoritme oppgave b: (viser kun endringer i algoritmen)

4) Definere while-løkke som kjører frem til fartsvektoren i y-retning v[1] >= 0. (Det vil si frem til ballen ikke har fart oppover lenger).

6) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String-funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde verdiene til 2 desimaler.

```
print (f'Ballen når maksimalhøyde på {round(r[1], 2)} meter.')
```

Algoritme oppgave c: (viser kun endringer i algoritmen)

4) Definere while-løkke som kjører frem til posisjonsvektoren i y-retning r[1] >= 2.1 (Det vil si frem til ballen har en høyde lik 2.1 m over bakken), samtidig som en kun ser på situasjon 2 – dvs når v[1] > 0, som gjelder frem til ballen når sitt høyeste punkt.

6) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-

verdier. I tillegg benyttes round()-funksjonen for å avrunde verdiene til 2 desimaler.

```
print (f'Ballen har beveget seg {round(r[0], 2)} meter i horisontal retning.')
```

5.6.2.5 Program
Oppgave a)

```
# Skrått kast med luftmotstand

from operator import concat
from pylab import *

# Konstanter
m = 0.145               # massen av gjenstanden, kg
k = 1.31*10**(-3)       # luftmotstandstallet, kg/m
g = 9.81                # tyngdeakselerasjon, m/s^2
v0 = 42.7               # startfart, m/s
y0 = 1.20               # starthøyde, m
theta = radians(40)     # konverterer vinkel i grader til radianer

# Konstante krefter
G = array([0, -m*g])    # tyngden i N

# Variable krefter, utregning av kraftsum og akselerasjon
def a(v):                           # akselerasjonsfunksjonen
    e_v = v/norm(v)                 # enhetsvektor for farten
    L = -k*norm(v)**2 * e_v         # luftmotstandsvektor, N
    sum_F = G + L                   # vektorsummen av kreftene, N
    aks = sum_F/m                   # akselerasjonsvektoren, m/s^2
    return aks

# Startverdier for bevegelsen
r = array([0, y0])                          # startposisjon, m
v = array([v0*cos(theta), v0*sin(theta)])   # startfart, m/s
t = 0                                       # starttid, s

# Lister for lagring av verdier
r_liste = [r]
v_liste = [v]

# Simulering av bevegelsen
dt = 0.001                  # tidssteg i simuleringen, s

while r[1] ⩾ 0:             # stopper når y = 0
    v = v + a(v)*dt         # regner ut neste fartsvektor
    r = r + v*dt            # regner ut neste posisjonsvektor
    t = t + dt              # går til neste tidspunkt

    # Lagring av 2D-verdier i lister
    r_liste = concatenate([r_liste, [r]])
    v_liste = concatenate([v_liste, [v]])

# Tegning av graf
plot(r_liste[:,0], r_liste[:, 1])       # lager grafen
title("Skrått kast med luftmotstand")   # tittel på grafen
xlabel("$x$ / m")                       # x-akse tittel
ylabel("$y$  / m")                      # y-akse tittel
grid()                                  # legger til rutenett
show()                                  # viser grafen
```

Figur 5-156: Kode for oppgave 2 a - Skrått kast med luftmotstand

Oppgave b)

```python
# Skrått kast med luftmotstand b)

from operator import concat
from pylab import *

# Konstanter
m = 0.145               # massen av gjenstanden, kg
k = 1.31*10**(-3)       # luftmotstandstallet, kg/m
g = 9.81                # tyngdeakselerasjon, m/s^2
v0 = 42.7               # startfart, m/s
y0 = 1.20               # starthøyde, m
theta = radians(40)     # konverterer vinkel i grader til radianer

# Konstante krefter
G = array([0, -m*g])    # tyngden i N

# Variable krefter, utregning av kraftsum og akselerasjon
def a(v):                           # akselerasjonsfunksjonen
    e_v = v/norm(v)                 # enhetsvektor for farten
    L = -k*norm(v)**2 * e_v         # luftmotstandsvektor, N
    sum_F = G + L                   # vektorsummen av kreftene, N
    aks = sum_F/m                   # akselerasjonsvektoren, m/s^2
    return aks

# Startverdier for bevegelsen
r = array([0, y0])                          # startposisjon, m
v = array([v0*cos(theta), v0*sin(theta)])   # startfart, m/s
t = 0                                       # starttid, s

# Lister for lagring av verdier
r_liste = [r]
v_liste = [v]

# Simulering av bevegelsen
dt = 0.001                  # tidssteg i simuleringen, s

while v[1] >= 0:            # stopper når farten i y-retning er 0
    v = v + a(v)*dt         # regner ut neste fartsvektor
    r = r + v*dt            # regner ut neste posisjonsvektor
    t = t + dt              # går til neste tidspunkt

    # Lagring av 2D-verdier i lister
    r_liste = concatenate([r_liste, [r]])
    v_liste = concatenate([v_liste, [v]])

#Skriver maksimal høyde til skjerm
print(f'Ballen når maks høyde på {round(r[1], 2)} meter')

# Tegning av graf
plot(r_liste[:,0], r_liste[:, 1])       # lager grafen
title("Skrått kast med luftmotstand")   # tittel på grafen
xlabel("$x$ / m")                       # x-akse tittel
ylabel("$y$  / m")                      # y-akse tittel
grid()                                  # legger til rutenett
show()                                  # viser grafen
```

Figur 5-157: Kode for oppgave 2 b - Skrått kast med luftmotstand

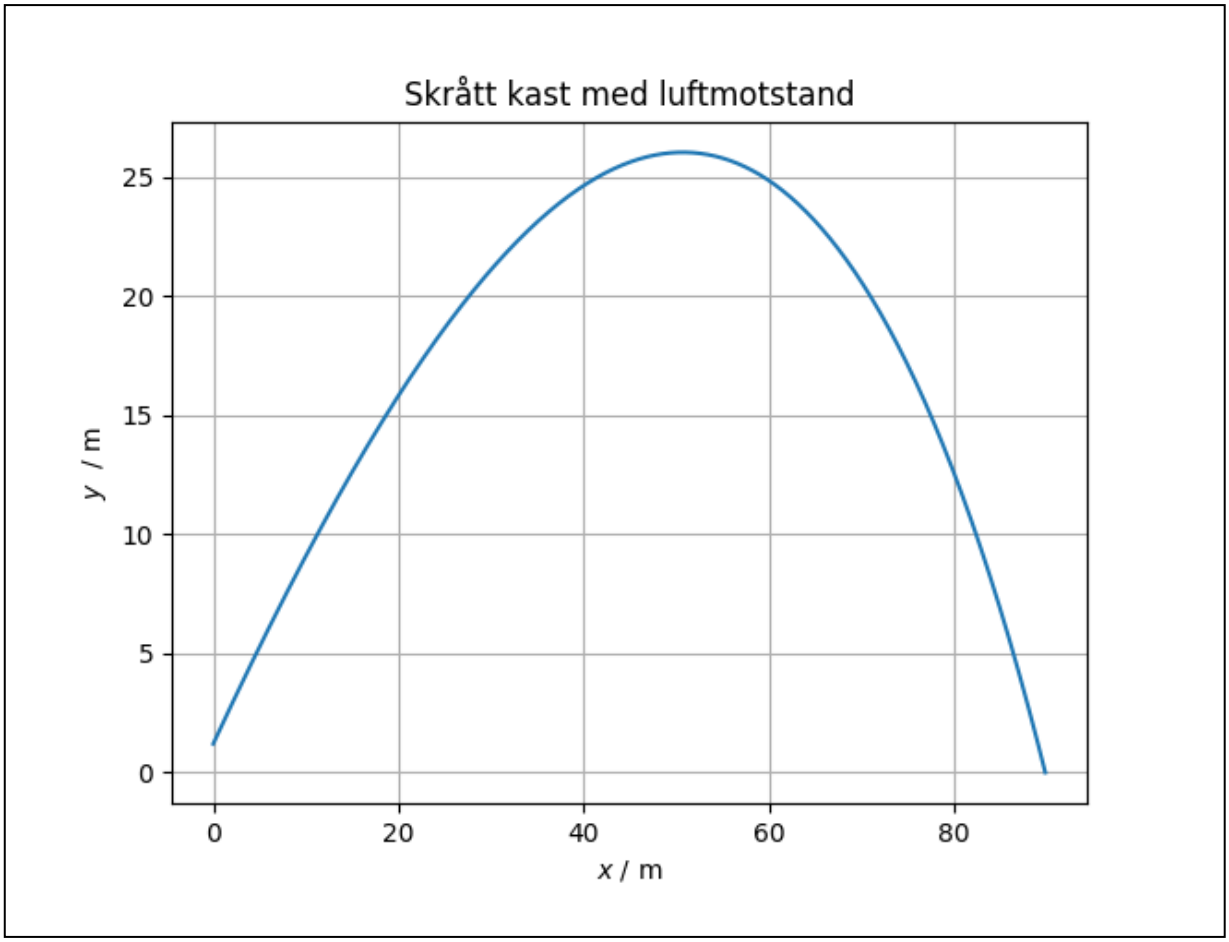
Oppgave c)

```python
# Skrått kast med luftmotstand c)

from operator import concat
from pylab import *

# Konstanter
m = 0.145               # massen av gjenstanden, kg
k = 1.31*10**(-3)       # luftmotstandstallet, kg/m
g = 9.81                # tyngdeakselerasjon, m/s^2
v0 = 42.7               # startfart, m/s
y0 = 1.20               # starthøyde, m
theta = radians(40)     # konverterer vinkel i grader til radianer

# Konstante krefter
G = array([0, -m*g])    # tyngden i N

# Variable krefter, utregning av kraftsum og akselerasjon
def a(v):                           # akselerasjonsfunksjonen
    e_v = v/norm(v)                 # enhetsvektor for farten
    L = -k*norm(v)**2 * e_v         # luftmotstandsvektor, N
    sum_F = G + L                   # vektorsummen av kreftene, N
    aks = sum_F/m                   # akselerasjonsvektoren, m/s^2
    return aks

# Startverdier for bevegelsen
r = array([0, y0])                          # startposisjon, m
v = array([v0*cos(theta), v0*sin(theta)])   # startfart, m/s
t = 0                                       # starttid, s

# Lister for lagring av verdier
r_liste = [r]
v_liste = [v]

# Simulering av bevegelsen
dt = 0.001                      # tidssteg i simuleringen, s

while v[1] > 0 or r[1] ⩾ 2.1:  # stopper når ballen når høyden 2.1, på vei
ned
    v = v + a(v)*dt             # regner ut neste fartsvektor
    r = r + v*dt                # regner ut neste posisjonsvektor
    t = t + dt                  # går til neste tidspunkt

    # Lagring av 2D-verdier i lister
    r_liste = concatenate([r_liste, [r]])
    v_liste = concatenate([v_liste, [v]])

#Skriver horisontal strekning til skjerm
print(f'Ballen har beveget seg {round(r[0], 2)} meter i horisontal retning.')

# Tegning av graf
plot(r_liste[:,0], r_liste[:, 1])       # lager grafen
title("Skrått kast med luftmotstand")   # tittel på grafen
xlabel("$x$ / m")                       # x-akse tittel
ylabel("$y$  / m")                      # y-akse tittel
grid()                                  # legger til rutenett
show()                                  # viser grafen
```

Figur 5-158: Kode for oppgave 2 c - Skrått kast med luftmotstand

5.6.2.6 Output

Oppgave a)



Oppgave b)

```
Ballen når maksimalhøyde på 26.04 meter
```

Oppgave c)

```
Ballen har beveget seg 88.43 meter i horisontal retning.
```

5.6.3 Oppgave 3 – Beregning av satellittbane

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin et al., Ergo Fysikk 2, 2022).

5.6.3.1 Kort beskrivelse av tema

I dagligtale tenker en på en satellitt som en kommunikasjonsenhet som går i bane rundt jorden. Et fellestrekk for satellitter er at de er objekter som går i fast bane rundt et himmellegeme. De satellittene vi har sendt opp i rommet for å gå i bane kalles kunstige satellitter.

5.6.3.2 Oppgave:

En satellitt med massen m = 1000 kg, går i bane rundt jorden. På et tidspunkt er avstanden fra sentrum av jorden til satellitten $4{,}0 * 10^7$m. Satellitten har da en fart på $2.4 * 10^3$ m/s vinkelrett på jordoverflaten.

Lag et program som simulerer satellittens bane rundt jorden og presenter en grafisk beskrivelse.

5.6.3.3 Problemløsning/Problemløsnings-strategi:

Utgangspunktet for løsning av oppgaven er at når satellitten går i bane rundt jorden, påvirkes den av gravitasjonskraften fra jorden. Denne kraften har da retning fra satellitten ned mot jordens midtpunkt (ideelt sett jordens sentrum).

Satellitten beveger seg i en tilnærmet sirkuler bane rundt jorden. Det betyr at den må være påvirket av en nettokraft i hele banen, siden den hele tiden endrer retning. (Den beveger seg ikke i en rett linje). For å kunne tegne en figur som illustrerer satellittens bane, må satellittens posisjon r i mange ulike tidspunkter noteres, og brukes som punkter som kan plottes på grafen.

Det regnes ikke med noe luftmotstand i dette eksempelet, slik at nettokraften som påvirker satellitten består av akselerasjonen aks som skyldes gravitasjonskraften G.

$$G = (\gamma * M * m) / r^2$$

der M er massen til jordkloden, m er massen til satellitten, γ er gravitasjonskonstanten og r er posisjonen til satellitten over jordens sentrum.

Utregningen i ett enkelt punkt ved hjelp av formelen er relativt enkel, men det kreves en rekke påfølgende tilsvarende utregninger for å kunne plotte en fullstendig satellittbane grafisk. Her kommer bruken av gjentakelser (iterasjoner) ved å benytte while- eller for-løkker i et programmeringsspråk til sin rett.

For hvert tidssteg dt kalkuleres akselerasjonen i det øyeblikket, og basert på akselerasjonen beregnes ny posisjon r til satellitten. Det benyttes en while-løkke for å foreta mange gjentatte beregninger av satellittens posisjon r. Satellitten vil naturlig nok fortsette å gå i bane «i det uendelige», så for å sørge for å terminere while-løkken, settes to vilkår:

$$t < 1 * 10^5$$

som gir en begrensning på hvor mange sekunder t en ønsker å plotte ferden, og

$$r > 6{,}371 * 10^6$$

som tilsier at satellittens avstand til jordens midtpunkt må være større enn jordens omtrentlige radius, slik at satellitten i det minste ikke krasjer med jorden.

For å oppdatere og beregne en fortløpende ny verdi av akselerasjonen aks, benyttes en egen funksjon akselerasjon(r) for å utføre beregningen av akselerasjon basert på satellittens posisjon r i øyeblikket.

Akselerasjonen aks beregnes ved

$$aks = G / m$$

der G er gitt som gravitasjonskraften (nettokraft). Å beregne G i et programmeringsspråk som Python, krever noen ekstra steg (disse er beskrevet i nærmere detalj i algoritme-delen)

- Først bestemmes G_abs som er absoluttverdien av gravitasjonskraften
- Deretter bestemmes e_r som er enhetsvektoren med retning mot sentrum av jorden. Denne benyttes for å bestemme G gitt som:

$$G = G\_abs * e\_r ,$$

der G_abs er absoluttverdien av tyngdekraftsvektoren, og e_r er enhetsvektoren som gir korrekt retning mot jordens sentrum.

5.6.3.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å tegne grafen som skal illustrere satellittens bane rundt jorden.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. m = 100, for massen til satellitten i kg
   b. M = 5.972 * 10^24, for massen til jorden
   c. Gamma = 6.67 * 10^(-11)
   d. r = array([4*10^7, 0]), for startposisjonen for satellitten
   e. v = array([0, 2.4 * 10^3]), for satellittens startfart i x- og y-komponent
   f. t = 0, for startverdi av tidsperioden som skal benyttes i beregning
   g. r_liste = [r], liste for lagring av de ulike posisjonene

h. dt = 10, for tidssteg mellom hver beregning

3) Definere funksjonen akselerasjon(r) som kalkulerer og returnerer satellittens akselerasjonen i posisjon r, gitt verdien av r sendt inn som argument til funksjonen.
   a. Oppretter lokal variable G_abs som beregner absoluttverdien av gravitasjonen gitt gamma, de to objektenes masser m og M, samt posisjonen r

      ```
      G_abs = (gamma * m * M) / norm(r)^2
      ```

      der norm() er en funksjon som gjøres tilgjengelig via biblioteket NumPy. (som er inkludert i pylab biblioteket).

   b. Oppretter lokal variabel for enhetsvektoren til posisjonen r som mottas som input til funksjonen. Dette er en enhetsvektor som har som mål å gi rette retning (+/-) til gravitasjonskraften som det regnes med.

      ```
      e_r = -r / norm(r)
      ```

      der norm() er en funksjon som gjøres tilgjengelig via biblioteket NumPy. (som er inkludert i pylab biblioteket).

   c. Oppretter lokal variable G for beregning av gravitasjonskraften, med korrigert retning

      ```
      G = G_abs * e_r
      ```

   d. Oppretter lokal variabel aks for lagring av beregnet verdi for akselerasjonen

      ```
      aks = G / m
      ```

   e. Returnerer verdien av variabelen aks

4) Definere while-løkke som kjører så lenge posisjonen r til satellitten er større enn r > 6.371*10^6, og tidsperioden t det beregnes nye verdier t < 1 *1 0^5. (Det vil si så lenge satellitten er over bakkenivå). While-løkken står for
   a. Beregning av ny akselerasjon ved hjelp av av nevnte funksjon akselerasjon(r)
   b. beregning av nye oppdaterte verdier av fart v, posisjon r og forløpt tid t, for hvert tidssteg Δt.
   c. oppdatering av listen r_liste med de nye verdiene, som skal benyttes til å plotte grafen. Verdiene legges til de eksisterende listen ved å benytte funksjonen concatenate() som utvider den eksisterende listen som inneholder arrayer, og legger den nye verdien til i listen som et nytt element, på siste plass i listen. Concatenate er en funksjon som er gjort tilgjengelig via biblioteket numpy (som er inkludert i pylab biblioteket).

5) Plotter de registrerte verdien i r_liste[:, 0] og r_liste[:, 1] som punkter på en graf ved hjelp av følgende funksjoner som er innebygget i pylab-biblioteket for å illustrere satellittens bane rundt jorden.
   a. axis(«equal»), bruker samme mål langs begge akser

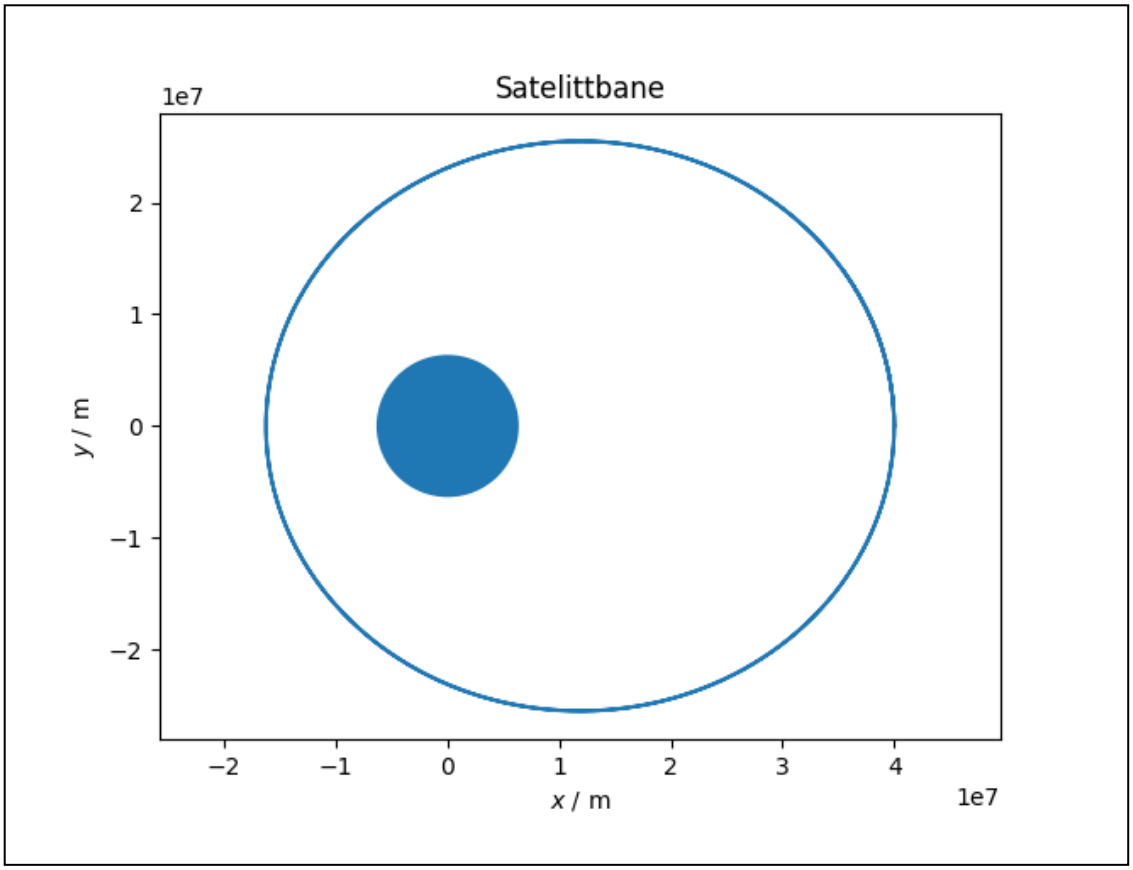
b. plot(x, y) , konstruerer grafen basert på en liste over x-verdier og y-verdier
c. title(«Streng»), setter tittel på grafen
d. xlabel(«Streng»), setter beskrivende tekst til x-aksen
e. ylabel(«Streng»), setter beskrivende tekst til y-aksen
f. gca().add_artist(Circle((0,0), 6.37 * 10^6)), som benytter funksjonen add_artist() for å tegne en sirkel i sentrum (koordinatene 0.0) med radius tilsvarende jordkloden. gca er inkludert i pylab biblioteket.
g. show(), funksjon som tegner grafen til skjerm

5.6.3.5 Program:

```python
# Beregning av satelittbane

from pylab import *

# Konstanter
m = 100                 # masse av satelitt, kg
M = 5.972e24            # masse av jorden, kg
gamma = 6.67e-11        # gravitasjonskonstanten, Nm^2/kg^2

# Startverdier
r = array([4e7, 0])     # posisjon til satelitten, m
v = array([0, 2.4e3])   # farten til satelitten, m/s
t = 0                   # tid, s

# Liste for lagring av verdier
r_liste = [r]           # liste med posisjoner

# Variable krefter, beregning av kraftsum og akselerasjon
def akselerasjon(r):
    G_abs = gamma*m*M/norm(r)**2    # absoluttverdi gravitasjon, N
    e_r = -r/norm(r)                # enhetsvektor mot sentrum av jorden
    G = G_abs*e_r                   # gravitasjonskraft med riktig retning
    aks = G/m                       # akselerasjon, m/s^2
    return aks

# Simulerer bevegelsen så lenge det ikke er gått 1*10^5 s
# og banen er over jordoverflaten.
dt = 10                 # tidssteg, s

while t < 1e5 and norm(r) > 6.371e6:
    a = akselerasjon(r)             # beregner akselerasjon
    v = v + a*dt                    # beregner ny fart
    r = r + v*dt                    # beregner ny posisjon
    t = t + dt                      # ny tid

    # Lagring av 2D-verdier
    r_liste = concatenate([r_liste, [r]])

# Tegner graf
axis("equal")                               # setter akser like
title("Satelittbane")                       # tittel på grafen
xlabel("$x$ / m")                           # navn på x-aksen
ylabel("$y$ / m")                           # navn på y-aksen
gca().add_artist(Circle((0,0), 6.37e6))     # tegner sirkel som viser jorden
plot(r_liste[:,0], r_liste[:,1])            # plotter posisjonene
show()                                      # viser grafen
```

Figur 5-159: Kode for oppgave 3 – Beregning av satellittbane

5.6.3.6 Output:



5.6.4 Oppgave 4 – Gravitasjon mellom to objekter

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin et al., Ergo Fysikk 2, 2022).

5.6.4.1 Kort beskrivelse av tema

Når en i dagligtale snakker om gravitasjonskrefter eller tyngdekraften, tenker en vanligvis på kraften et legeme med enorm masse har på et legeme med liten masse, som et eple som faller mot jordkloden, eller en astronaut som hopper rundt på månen. Det er likevel de samme reglene som gjelder når massen til 2 legemer er av mer jevnbyrdige størrelser.

5.6.4.2 Oppgave:

To personer med samme masse m1 = m2 = 60 kg er plassert i verdensrommet i en avstand av r = 1m. Til å begynne med er begge personene i ro, men etter hvert trekker gravitasjonskraften de to personen mot hverandre. Hvor lang tid vil det ta før de møtes?

5.6.4.3 Problemløsning/Problemløsnings-strategi:

Det regnes ikke med noe luftmotstand i dette eksempelet, slik at nettokraften som påvirker de to objektene (personene) er gravitasjonskraften G, gitt som

$$G = (\gamma * m_1 * m_2) / r^2 ,$$

Hvor γ = gamma. Siden m1 = m2 kan dette igjen skrives som

$$G = (\gamma * m_2) / r^2$$

Akselerasjonen gitt som a = ΣF / m vil også være like stor for de to personene, men motsatt rettet. Dette kan en ta høyde for ved å beregne farten v til de to personene som

```
v1 + a(r) * dt
```

for person 1, og

```
v2 – a(r) * dt,
```

for person 2

Det opprettes en funksjon a(r) som beregner akselerasjonen basert på gravitasjonskraften som virker på hver av de to personene i en gitt posisjon r. Akselerasjonen for hver person er gitt som

```
aks = G / m
```

For hvert tidssteg dt kalkuleres akselerasjonen i det øyeblikket, og det benyttes en while-løkke for å foreta mange gjentatte beregninger av avstanden r mellom de personene. While-løkken vil fortsette å kjøre frem til avstanden mellom de to personene r = r2 - r1 har blitt 0

```
while r > 0:
```

5.6.4.4 Algoritme:

1) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. m = 60, for massen til hver av de to personene i kg
    b. gamma = 6.67 * 10^(-11)
    c. r1 = 0, for startposisjonen til person1 (Origo)
    d. v1 = 0, for startfarten til person 1
    e. r2 = 1.0, for startposisjonen til person 2
    f. v2 = 0, for startfarten til person 2
    g. r = r2 - r1, for gjenværende avstand mellom person1 og person 2
    h. t = 0, for starttidspunktet
    i. dt = 0.01, for tidssteg mellom hver beregning

2) Definere funksjonen a(r) som kalkulerer kraftsum for hver person, og returnerer personens akselerasjon aks i posisjon r, gitt verdien av r sendt inn som argument til funksjonen.
    a. Oppretter lokal variable G som beregner gravitasjonen gitt gamma, de to objektenes masser 2*m, samt posisjonen r

        ```
        G = (gamma * m * m) / r^2
        ```

    b. Oppretter lokal variabel aks for lagring av beregnet verdi for akselerasjonen

        ```
        aks = G / m
        ```

    c. Returnerer verdien av variabelen aks

3) Definere while-løkke som kjører så lenge avstanden r mellom de to personene er: r > 0 (Slik at løkken avsluttes når avstanden mellom peronene er 0, og de møtes).

    While-løkken står for beregning av nye verdier fo hvert tidssteg Δt

    a. Beregning av ny fart for person 1 ved hjelp av nevnte funksjon a(r)

        ```
        v1 + a(r) * dt
        ```

    b. Beregning av ny fart for person 2 ved hjelp av nevnte funksjon a(r)

        ```
        v2 – a(r) * dt
        ```

Merk at farten v2 til person 2 er motsatt rettet farten v1 til person 1, slik at fartsendringen kommer til fratrekk i denne beregningen.

c. beregning av nye oppdaterte verdier av posisjon r1 og r2 for de to personene utføres,

`r1 + v1 * dt`, for posisjonen til person 1

`r2 + v2 * dt`, for posisjonen til person 2

d. verdier oppdateres og forløpt tid t, for hvert tidssteg Δt registreres.

4) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier. I tillegg benyttes round()-funksjonen for å avrunde verdiene til 3 desimaler.

```
print (f'Tid før kollisjon mellom person 1 og person 2:
{round(t/3600, 3)}')
```

5.6.4.5 Program:

```python
# Gravitasjon mellom to objekter

# Konstanter
m = 60                    # massen av hver elev
gamma = 6.67*10**(-11)  # gravitasjonskonstanten, Nm^2/kg^2

# Variable krefter, utregning av kraftsum og akselerasjon
def a(r):
    G = gamma*m**2/r**2     # gravitasjonskraft for hver elev, N
    sum_F = G               # kraftsum for hver elev, N
    aks = sum_F/m           # akselerasjon for hver elev, m/s^2
    return aks

# Startverdier for bevegelsen
r1 = 0                      # elev 1 starter i origo
v1 = 0                      # elev 1 starter i ro
r2 = 1.0                    # elev 2 starter 1.0m fra origo
v2 = 0                      # elev 2 starter i ro
r = r2 - r1                 # avstand mellom de to elevene
t = 0                       # starttid

# Simulering av bevegelsen
dt = 0.01                   # tidssteg i simuleringen
while r > 0:                # stopper kjøring av simulering når elevene møtes
    v1 = v1 + a(r)*dt       # regner ut neste fart for elev 1
    v2 = v2 - a(r)*dt       # regner ut neste fart for elev 2 (motsatt retning)
    r1 = r1 + v1 * dt       # regner ut neste posisjon for elev 1
    r2 = r2 + v2 * dt       # regner ut neste posisjon for elev 2
    r = r2 - r1             # ny avstand mellom elevene
    t = t + dt              # avanserer til neste tidspunkt

print(f"Tid før kollisjon mellom person 1 og person 2: {round(t/3600, 3)} h")
```

Figur 5-160: Kode for oppgave 4 – Gravitasjon mellom to objekter

Output:

```
Tid før kollisjon mellom person 1 og person 2: 3.449 h
```

5.6.5 Oppgave 5 – Elektriske krefter og felt

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin, Dokka, Hellesøy, Seland, & Skåland, Ergo Fysikk 2, 2022).

5.6.5.1 Kort beskrivelse av tema

Elektrisk strøm er ladninger som er i bevegelse, og ladningene påvirkes av elektriske krefter i omgivelsene, som får ladningene til å bevege seg. Coulombs lov beskriver den elektriske kraften som virker mellom to punktformede gjenstander som har ladningen q1 og q2, og er i en avstand r fra hverandre.

$$Fe = ke\ (|q1| * |q2|) / r^2$$

der ke er en konstant med verdien $8{,}99 * 10^9\ Nm^2/C^2$ i vakuum. Dersom ladningene q1 og q2 har motsatt fortegn, er kraften som virker mellom dem tiltrekkende, og dersom ladningene har samme fortegn er kraften som virker mellom dem, frastøtende.

5.6.5.2 Oppgave:

(Bygger på Ernest Rutherfords «gullfolie-eksperiment»)

En alfa-partikkel beveger seg rett mot sentrum av kjernen til et tungt gull-atom. Alfa-partikkelen har ladningen $q_1 = 2e = 3{,}20 * 10^{-19}$ C, og massen $m = 4u = 6{,}64 * 10^{-27}$ kg.

Gull-atomet har ladningen $q_2 = 79e = 1{,}26 * 10^{-17}$ C.

Hvor nær gullkjernen klarer alfa-partikkelen å komme før den blir frastøtt?

5.6.5.3 Problemløsning/Problemløsnings-strategi:

Det antas at den mye større massen til gullatomet gjør at det blir liggende i ro, mens alfa-partikkelen beveger seg mot gull-atomet med sin initial-fart. Gullkjernen ligger da på posisjonen r = 0, og alfapartikkelen starter da ved sin posisjon $ra = -1.0 * 10^{-10}$ unna.

På alfapartikkelen virker det da en kraft

$$F = k\ (q_1 * q_2) / r^2$$

Av denne kan akselerasjonen til alfapartikkelen finnes ved

$$a = \Sigma F / m,\ \text{ som gir } a = -\ k\ (q_1 * q_2) / (m * r^2)$$

5.6.5.4 Algoritme:

1) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
    a. m = 6.64 * 10^(-27), for massen til alfapartikkelen i kg
    b. q1 = 3.20 * 10^(-19), for ladningen til alfapartikkelen
    c. q2 = 1.26 * 10 ^(-17), for ladningen til gullkjernen
    d. k = 8.99 * 10^9, for Coulombs konstant
    e. r = -1.0 * 10^(-10), for startavstanden mellom alfapartikkelen og gullkjernen
    f. v = 1.57 * 10^7, for startfarten til alfapartikkelen i retning gullkjernen
    g. t = 0, for starttidspunktet
    h. dt = 1.0 * 10^(-22), for tidssteg mellom hver beregning

2) Definere while-løkke som kjører så lenge farten v til alfa-partikkelen er positiv. v > 0 (Det vil si frem til partikkelen snur etter å ha blitt frastøtt av gullkjernen).

   While-løkken står for beregning av nye verdier for hvert tidssteg Δt

    a. Beregning av akselerasjonen a i øyeblikket, gitt som

```
a = (-k * q1 * q2) / (m * r ^2)
```

    b. Beregning av ny fart for alfapartikkelen

```
v + a * dt
```

    c. Beregning av ny posisjon r for alfapartikkelen

```
r + v * dt
```

    d. verdier oppdateres og forløpt tid t, for hvert tidssteg Δt registreres.

3) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren. Resultat-strengene formateres ved hjelp av f-String-funksjonen som tillater en kombinasjon av statiske strenger, og dynamisk oppdaterte variabel-verdier.

```
print (f'Alfapartikkelen snur når den er r = {r} meter unna
gullkjernen')
```

5.6.5.5 Program:

```python
# Elektriske krefter og felt

# Konstanter
m = 6.64e-27        # massen til alfapartikkelen, kg
q1 = 3.20e-19       # ladning til alfapartikkelen, C
q2 = 1.26e-17       # ladning til gullkjernen, C
k = 8.99e9          # konstant i Coulombs lov

# Startverdier
r = -1.0e-10        # startavstand fra kjernen, m
v = 1.57e7          # startfart, m/s
t = 0               # starttid, s

# Simulering av bevegelsen
dt = 1.0e-22        # tidssteg, s

while v > 0 :                   # gjenta så lenge farten er positiv
    a = -k*q1*q2 / (m*r**2)     # beregner akselerasjon, m/s^2
    v = v + a*dt                # beregner ny fart
    r = r + v*dt                # beregner ny posisjon
    t = t + dt                  # beregner nytt tidspunkt

print(f'Alfapartikkelen snur når den er r = {r} meter unna gullkjernen.')
```

Figur 5-161: Kode for oppgave 5 – Elektriske krefter og felt

5.6.5.6 Output:

```
Alfapartikkelen snur når den er r = -4.427694628489823e-14 meter unna
gullkjernen.
```

5.6.6 Oppgave 6 – Inhomogene elektriske felt

Inspirasjon til denne oppgaven er hentet fra Ergo Fysikk 2 (Callin et al., Ergo Fysikk 2, 2022).

5.6.6.1 Kort beskrivelse av tema

Styrke og retning på elektriske felt kan variere, ettersom objekters ladning kan være positiv og negativ, og plassering (posisjon) kan variere. I en dipolar sammenheng, er området rundt de to polene påvirket av de 2 like store ladningene, som har motsatt fortegn (positiv/negativ) og resultatet blir at eventuelle partikler som befinner seg i ulike avstander unna de to polene, vil oppleve ulike nettokraft også kalt feltstyrke basert på ladning og avstand.

5.6.6.2 Oppgave:

En positivt og en negativt ladet kule er 10 cm fra hverandre. Begge kulene har ladningen 2,00 nC i absoluttverdi. På et midtpunkt på den tenkte rette linjen mellom de 2 kulene, finnes normalen til et punkt P som ligger 10 cm over linjen mellom de to kulene.

Finn styrken og retningen til det elektriske feltet i punktet p.

5.6.6.3 Problemløsning/Problemløsnings-strategi:

Velger først midtpunktet mellom de to ladningene som origo (0,0). Da ligger P 10 cm over dette punktet, og modellen er helt symmetrisk.

Da ligger ladning q_1 på punktet p_1(-0.05, 0), og ladning q_2 på punktet p_2(0.05, 0). De to ladningene er dipolare, så en er negativ og en er positiv. Setter q_1 til å være negativ og q_2 til positiv.

Punktet P opplever en netto-kraft fra de to ladningene q_1 og q_2 som er summen av de to kreftene. Beregner hver av de 2 enkeltkreftene som utgjør feltet i P.

Hver av de to kulene resulterer i en feltstyrke E bestemt av

$$E = kc\,|Q| / r^2,$$

der r er avstanden til punktet en ønsker å måle feltet.

For å finne nettofeltstyrken i punktet P, beregnes derfor de 2 kulenes tilskudd, og dette legges sammen.

For å beskrive både kulenes posisjon, kreftenes retning, benyttes vektorer.

Å beregne E-feltet i et programmeringsspråk som Python, krever noen ekstra steg (disse er beskrevet i nærmere detalj i algoritme-delen) .

Det defineres en funksjon E-felt(r) som har til oppgave å beregne og returnere netto elektrisk felt i punktet P:

- Først bestemmes $\overrightarrow{r1}$ som er en vektor som representerer avstanden (og retning) fra p1 til punktet P.
- Deretter bestemmes $\overrightarrow{r1\_enhet}$ som er enhetsvektoren med retningen til r_1 . Denne benyttes for å bestemme det elektriske felt-komponenten E_1 fra kule 1 i punktet P.
- Dette gjentas for $\overrightarrow{r2\_enhet}$ som da er enhetsvektoren med retningen til $\overrightarrow{r2}$, som igjen benyttes for å beregne det elektriske felt-komponenten E_2 fra kule 2 i punktet P.
- Til slutt summeres disse til $E = E_1 + E_2$ som returneres av funksjonen.

5.6.6.4 Algoritme:

1) Importere pylab, som er et Python-bibliotek som gir funksjonalitet som trengs for å utføre en del av beregningene.

2) Definere variabler som trengs for lagring av mellomverdier til bruk i beregning.
   a. q1 = 2.00 * 10^(-9), for ladningen til kule 1
   b. p1 = array([-0.05, 0]), for posisjonen til kule 1
   c. q2 = -2.00 * 10^(-9), for ladningen til kule 2
   d. p2 = array([0.05, 0]), for posisjonen til kule 2
   e. k = 8.99 * 10^9, for konstant i Coulombs lov
   f. punkt = array([0, 0.10]), for posisjonen til punktet P hvor elektrisk felt skal beregnes
   g. E for lagring av verdien av det beregnede E-feltet

3) Definere funksjonen E_felt(r) som kalkulerer og returnerer nettosummen av det elektriske feltet i punktet r, gitt verdien av r sendt inn som argument til funksjonen.
   a. Oppretter lokal variabel r1 for vektoren som går fra punktet p1 til r.

```
r1 = r - p1
```

   b. Oppretter lokal variabel r1_enhet for enhetsvektor ut fra p1

```
r1_enhet = r1/norm(r1)
```

   c. Beregner lokal variabel E1, elektrisk felt komponent fra kule 1

```
E1 = (k * q1) / (norm(r1)^2 * r1_enhet)
```

   d. Oppretter lokal variabel r2 for vektoren som går fra punktet p2 til r.

```
r2 = r - p2
```

e. Oppretter lokal variabel r2_enhet for enhetsvektor ut fra p2

```
r2_enhet = r2/norm(r2)
```

f. Beregner lokal variabel E2, elektrisk felt komponent fra kule 2

```
E2 = (k * q2) / (norm(r2)^2 * r2_enhet)
```

g. Oppretter lokal variabel E for lagring summen av de to elektriske kraftkomponentene

```
E = E1 +E2
```

h. Returnerer verdien av variabelen E.

4) Benytter funksjonen E_felt(r) med argument punkt og lagrer resultatet i en global variabel E
5) Benytter funksjonen print() som er innebygget i Python til å presentere resultatet for brukeren.

print(«Det elektriske feltet i punktet», punkt, «er» , E, «.»)

print(«Det har en absoluttverdi på», norm(E), «V/m»)

print(«Vinkelen mellom feltretningen og x-aksen er», arctan(E[1] / E[0], «radianer»)

5.6.6.5 Program:

```python
# Inhomogene elektriske felt

from pylab import *

# Konstanter
q1 = 2.00e-9                # ladning til kule 1, C
P1 = array([-0.05, 0])      # posisjon til kule 1, m
q2 = -2.00e-9               # ladning til kule 2, C
P2 = array([0.05, 0])       # posisjon til kule 2, m
k = 8.99e9                  # konstanten i Coulombs lov

# Funksjon som beregner det elektriske feltet i posisjonen r
def E_felt(r):
    r1 = r - P1                         # vektor fra P1 til r
    r1_enhet = r1/norm(r1)              # enhetsvektor ut fra P1
    E1 = k*q1/norm(r1)**2 * r1_enhet    # elektrisk felt fra kule 1
    r2 = r - P2                         # vektor fra P2 til r
    r2_enhet = r2/norm(r2)              # enhetsvektor ut fra P2
    E2 = k*q2/norm(r2)**2 * r2_enhet    # elektrisk felt fra kule 2
    E = E1 + E2                         # summerer feltene fra de to kulene
    return E

# Beregning
punkt = array([0, 0.10])                # punktet hvor feltet skal beregnes
E = E_felt(punkt)                       # beregner felt vha funksjon

# Skriver ut resultatet
print("Det elektriske feltet i punktet", punkt, "er", E, ".")
print("Det har en absoluttverdi på", norm(E), "V/m")
print("Vinkelen mellom feltretningen og x-aksen er", arctan(E[1]/E[0]), "radianer.")
```

Figur 5-162: Kode for oppgave 6 – Inhomogene elektriske felt

5.6.6.6 Output:

```
Det elektriske feltet i punktet [0.    0.1] er [1286.54407153     0.     ] .
Det har en absoluttverdi på 1286.5440715342788 V/m,
Vinkelen mellom feltretningen og x-aksen er 0.0 radianer.
```

6 Konklusjon

Som del av bestillingen fra oppdragsgiver, ble det ved starten av prosjektet beskrevet en del krav og forventninger til sluttproduktet. Det ble satt som mål å:

1. Utvikle grundige innføringshefter i programmering for lærere som skal undervise i naturfag og/eller fysikk på ungdomstrinnet, eller i den videregående skolen.

2. Utvikle gode oppgavehefter i programmering for lærere som de kan benytte i sin undervisning på de ulike alderstrinnene. U8-10, VG1, VG2 og VG3

I tillegg til disse hovedmålene, har det hele veien gjennom prosessen med utvikling av oppgaveheftene, vært viktig å fokusere på å presentere og ferdigstille oppgavene på en slik måte at de oppfordrer elevene til refleksjon, problem-løsning og algoritmisk tenking, gjennom steg-for-steg overgang fra en kontekstuell skriftlig oppgave som skal løses, via informasjonssamling og modellering, til utarbeidelse av klare algoritmer som til slutt kulminerer i programkode som kan eksekveres.

På samme tid har det vært et poeng å lage oppgaver av ulik vanskelighetsgrad, for å prøve å motivere og utfordre den som skal løse oppgavene, men samtidig prøve å sørge for at de er interessant nok til også å skape interesse for temaet som omhandles, og fysikk og naturfag som fag.

Rammeplanene for de ulike alderstrinnene har også lagt en del føringer for hvilke tema innen fysikk og naturfag som er egnet for bruk av programmering. Vi har i all vesentlig grad fulgt dette, men har i en del eksempler i innføringsheftene også benyttet enkelte litt mer dagligdagse tema, for å både vise muligheter for bruk også utenfor de typiske STEM-fagene, og gjøre enkelte ting mer håndgripelig og gjenkjennbart.

Vi håper innføringsheftene og oppgaveheftene vi har produsert som sluttprodukt samlet sett kan hjelpe lærere og lærerassistenter med den store, spennende og sikkert lett skremmende oppgaven det må være å ta fatt på en ny hverdag i yrket som motivator, inspirator, og underviser i programmering.

Referanser

Arntzen, M., Bækkedal, K. S., Fossestøl, K. O., & Fægri, K. (2022). *Element 10.* Gyldendal Norsk Forlag.

Brandt, H., Hushovd, O. T., & Tellefsen, C. W. (2020). *Naturfag SF* (2. utg.). H. Aschehoug & Co.

Brandt, H., Hushovd, O. T., & Tellefsen, C. W. (u.d.). *Naturfag Påbygging.* Aschehoug Norsk Forlag.

Callin, P., Dokka, I. H., Hellesøy, A., Seland, A., & Skåland, E. K. (2021). *ERGO Fysikk 1* (3. utg.). Aschehoug Undervisning.

Callin, P., Dokka, I. H., Hellesøy, A., Seland, A., & Skåland, E. K. (2022). *Ergo Fysikk 2* (3. utg.). Aschehoug Norsk Forlag.

Cappelen Damm. (u.d.). *Cappelen Damm.* Henta frå https://kraft.cappelendamm.no/: https://kraft.cappelendamm.no/binaryfile.php?tid=t-2991205

Cappelen Damm. (u.d.). *Naturfag 8-10 fra Cappelen Damm*. Henta frå Cappelen Damm: https://www.cappelendammundervisning.no/verk/Naturfag%208-10%20fra%20Cappelen%20Damm-153458

Dellnes, H. R., Vinjusveen, H., Fossum, J.-C., Sandstad, M., & Bergsli, E. (2021). *Kraft 1 Fysikk 1.* Cappelen Damm.

Gyldendal. (u.d.). *Element Naturfag 8-10*. Henta frå Gyldendal: https://www.gyldendal.no/grs/element/c-486691/

Lær Kidsa Koding. (u.d.). *Lær Kidsa Koding*. Henta frå https://www.kidsakoder.no: https://www.kidsakoder.no/om-lkk/

Polya, G. (1973). *A New Aspect of Mathematical Method* (2. utg.). Princeton Science Library.

Sevik, K., & m.fl. (2016, 11). *Notat om programmering i skolen*. Henta frå Utdanningsdirektoratet: https://www.udir.no/kvalitet-og-kompetanse/profesjonsfaglig-digital-kompetanse/notat-om-programmering-i-skolen/

Statped. (2021, 03 01). *Programmering*. Henta frå Statped: https://www.statped.no/laringsressurser/teknologitema/programmering-for-barn-med-saerskilte-behov/programmering/algoritme-og-algoritmisk-tenkning/)

Steineger, E., Wahl, A., & Holstad, H. H. (2021). *Naturfag 10.* Cappelen Damm.

Utdanningsdirektoratet. (2019, 03 27). *Utdanningsdirektoratet*. Henta frå https://www.udir.no/: https://www.udir.no/kvalitet-og-kompetanse/profesjonsfaglig-digital-kompetanse/algoritmisk-tenkning/

Utdanningsdirektoratet. (u.d.a). *Fysikk (FYS01-02): Kjerneelementer*. Henta frå Utdanningsdirektoratet: https://www.udir.no/lk20/fys01-02/om-faget/kjerneelementer?TilknyttedeKompetansemaal=true&anchorId=KE612

Utdanningsdirektoratet. (u.d.b). *Naturfag (NAT01-04): Kjerneelementer*. Henta frå Utdanningsdirektoratet: https://www.udir.no/lk20/nat01-04/om-faget/kjerneelementer?lang=nob

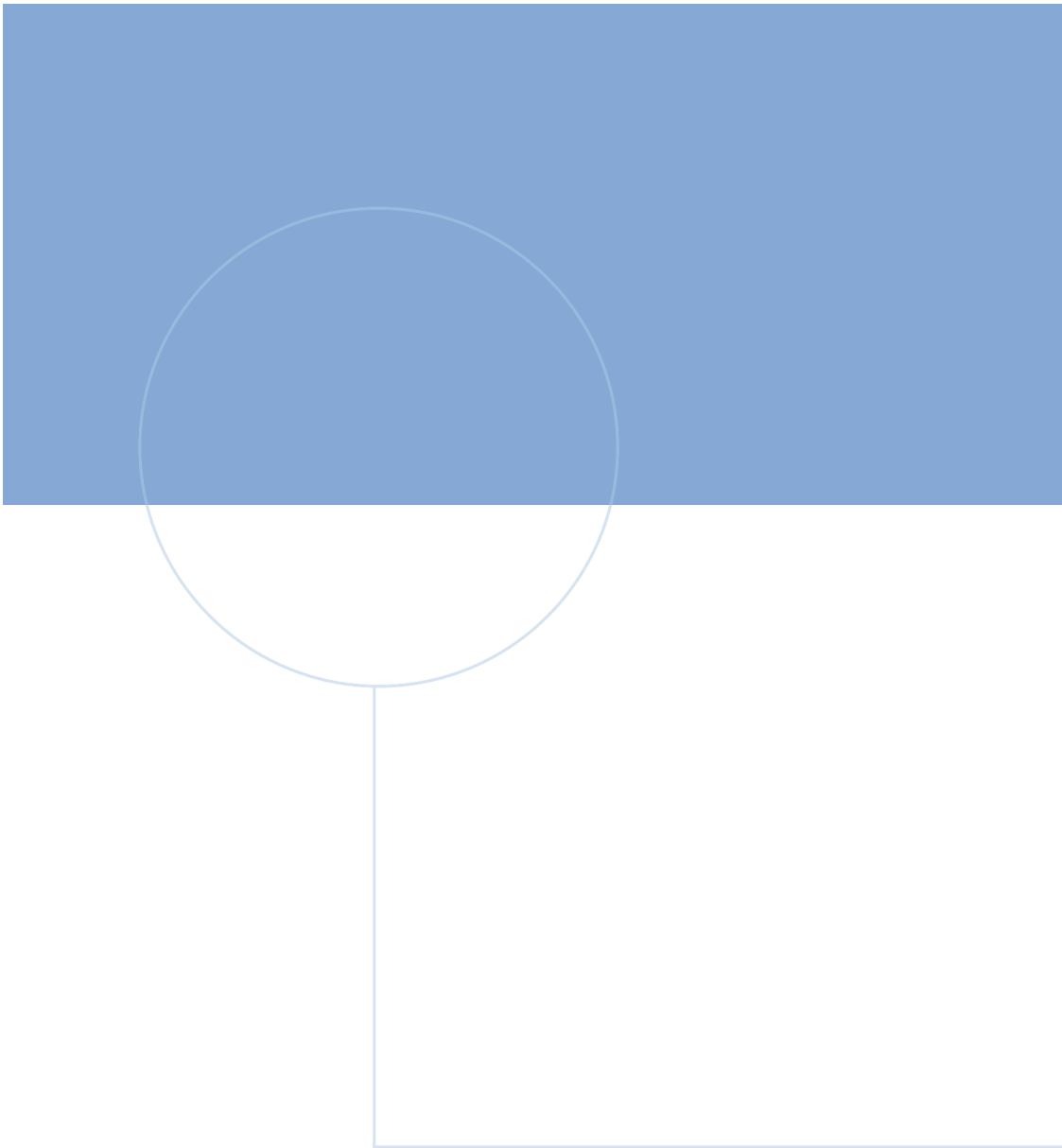
Utdanningsdirektoratet. (u.d.c.). *Naturfag (NAT01-04): Kompetansemål og vurdering*. Henta frå Utdanningsdirektoratet: https://www.udir.no/lk20/nat01-04/kompetansemaal-og-vurdering/kv79?lang=nob

Utdanningsdirektoratet. (u.d.d.). *Naturfag (NAT01-04): Kompetansemål og vurdering*. Henta frå Utdanningsdirektoratet: https://www.udir.no/lk20/nat01-04/kompetansemaal-og-vurdering/kv78?lang=nob

Vickers, P. (u.d.). *How To Think Like a Programmer.* Henta frå Cengage: http://cws.cengage.co.uk/vickers2/students/httlaptable.pdf

Vogt, Y. (2021, 03 23). *Digi.no*. Henta frå https://www.digi.no/: https://www.digi.no/artikler/programmering-blir-allemannseie-i-skolen/508362

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering.* Springer.



NTNU

Kunnskap for en bedre verden