Analyse av studenters eksamenskode: typiske feil og muligheter for autoretting

Guttorm Sindre
Excited SFU og Institutt for datateknologi og informatikk (IDI), NTNU

Sammendrag

Selv om programkode egner seg for automatisk prosessering, har den typisk vært vurdert manuelt, også etter innføring av digital eksamen. Vanlige verktøy for digital eksamen har ikke tilbudt oppgavetyper hvor studentene kan teste koden underveis, ei heller at sensor kjører koden under retting. Slik funksjonalitet vil bli tilgjengelig i nær framtid, og det er interessant å se hvilke implikasjoner dette kan ha for eksamen. Denne artikkelen analyserer studentkode på noen forholdsvis enkle oppgaver i Python-programmering fra en eksamen i høst 2019. Få studenter hadde kode som ville ha kjørt feilfritt, men vesentlig flere hadde kode som de kunne ha klart å rette til feilfri kjøring i løpet av kort tid gitt testmulighet. Analysen avdekker også typiske feil som gikk igjen i mange studenters kode, og diskuterer hvordan en eksamen med helt eller delvis automatisk retting av koden vil slå ut.

Innledning

Eksamen i programmering kan ha ulike oppgavesjangre, som konseptuelle faktaspørsmål, kodeforståelse, feilfinning/-retting, og komplettering av kode hvor fragmenter er utelatt, men det mest dominerende er skriving av programkode [1-3]. Selv med digital eksamen er det vanlig at slike oppgaver vurderes manuelt under sensur. Det fins e-læringsverktøy hvor kode kan kjøres mot testsuiter og scores automatisk basert på antall passerte tester – slik som Moodle CodeRunner[1]. Verktøy som Inspera Assessment eller WISEflow har ikke hatt slik funksjonalitet, verken student eller sensor, men dette er i ferd med å endre seg. Inspera har betafunksjonalitet for oppgavetypen «Kompilering»[2] hvor studenter kan teste koden underveis, og sensor kan gi poeng helt eller delvis automatisk, basert på antall passerte tester. Særlig i introkurs med mange studenter kan mulighet for auto-retting gi ytterligere fordeler for digital eksamen versus papireksamen [4-6].

Vil det være en fordel for studentene å kunne teste koden på eksamen? For de som får koden til å virke, vil det gi en ekstra trygghet å se at den fungerer – samt at små glipper blir synliggjort og kan fikses før levering. For svakere studenter kan derimot høy tidsbruk på å fikse feil i tidlige deloppgaver vil gi tidsnød på senere deloppgaver. Hvis man slett ikke får koden til å virke, ville man med manuell retting fortsatt få noen poeng her og der, mens poenggivning kun basert på testtilfeller vil gi null uttelling.

Denne artikkelen presenterer resultatene fra en analyse av studenters kode på deloppgaver med programmering i emnet TDT4127 ved NTNU høsten 2019, med digital eksamen i Inspera. Eksamenssettet besto av 70% autorettede oppgaver (flervalg, paring, dra og slipp, nedtrekk, fyll inn tekst) som dels fokuserte på kodeforståelse og dels på kodekomplettering (innfylling av manglende fragmenter i delvis ferdig kode). De autorettede oppgavene var gjenstand for en dypere analyse i artikkelen [6], som fant at disse korrelerte bra med programmeringsoppgavene. Her vil vi se nærmere på de 30% av oppgavesettet som besto av manuelt rettede programmeringsoppgaver, med tanke på å besvare følgende forskningsspørsmål:

- FS1: Hvordan fordelte studenters besvarelser seg mellom de som ville ha kjørt feilfritt mot testtilfeller, og de som hadde ulike feil?

[1] https://moodle.org/plugins/qtype_coderunner
[2] https://support.inspera.com/hc/no/articles/360039319172-Oppgavetype-Kompilering

This paper was presented at the UDIT 2020 conference. For more information see http://www.udit.no/

- FS2: Hvor mange studenter fikk full pott fra sensor selv om de hadde kode som ikke ville ha fungert? Hvor langt unna fungerende kode var disse?
- FS3: Hva var de mest utbredte feilene som studentene gjorde på programmeringsoppgavene?

Emnet hadde 252 oppmeldte studenter, hvorav 236 møtte til eksamen, og 210 hadde skrevet kode på de undersøkte deloppgavene. At det kun dreier seg om en enkelt eksamen, er en svakhet ved artikkelen, det ville vært bedre å ha data fra mange eksamener i ulike programmeringsemner ved ulike universiteter. Funn fra en enkelt eksamen kan likevel gi grunnlag for større sammenligninger senere.

Resten av artikkelen er strukturert som følger: Seksjon 2 gir en gjennomgang av forskningslitteratur om nybegynnerfeil i programmering. Seksjon 3 gjør rede for konteksten for det aktuelle emnet og hvordan eksamen var bygget opp. Seksjon 4 forklarer forskningsmetoden som ble brukt i analyse av dataene. Seksjon 5 presenterer og analyserer resultatene, og seksjon 6 inneholder en avsluttende diskusjon og konklusjon.

Relatert arbeid

Det har vært forsket mye på vanskene studenter har med å lære seg programmering, en større gjennomgang er gjort av Qian og Lehman [7]. Kaczmarczyk et al. [8] intervjuet førsteårs IT-studenter om hva de slet mest med å forstå. Hristova et al. [9] laget en liste over typiske nybegynnerfeil i Java. Brown og Altadmri [10] analyserte faglæreres oppfatninger av typiske nybegynnerfeil i Java-programmering versus faktiske feil fra en database med studentkode, og fant relativt begrenset samsvar både faglærere imellom og vis a vis faktiske feil. McCall og Kölling [11] kategoriserte ulike feil som studentene gjorde i Java ved automatisk innhenting av programkode som feilkompilerte. Kennedy og Kraemer [12] undersøkte studenters tankegang bak feil svar på oppgaver i C. Spesifikt relatert til nybegynnerfeil i Python, har Kohn [13] og Kallia og Sentance [14] undersøkt feil og misforståelser hos elever fra videregående skoler. Rørnes et al. [15] analyserte misforståelser hos førsteårs universitetsstudenter med en kombinasjon av eksperiment og intervju, spesielt relatert til bruk av variabelreferanser. Johnson et al. [16] diskuterte konsepter studenter sliter med å forstå i Python, særlig knyttet til lister og muterbarhet. Barstad et al. [17] brukte blant annet statisk analyse for å bedømme kvalitet av kode. Størst likheter med forskningen vår artikkel har kanskje Veerasamy et al. [18] som også analyserte studenters svar på en avsluttende eksamen i Python. Veersamy et al. brukte en kombinasjon av kvalitativ og kvantitativ analyse, med mye manuell inspeksjon, og analyserte alle deloppgavene men bare for 39 av studentene. I vår artikkel ser vi derimot kun på et utvalg av oppgaver som utgjorde 30% av eksamen, men for alle de 210 studentene som besvarte dem. Vi bruker primært automatisk analyse, med kjøring mot testsuiter og telling av forekomster av forventede uttrykk i koden – samt noe manuell inspeksjon av utvalgte besvarelser.

Kontekst og eksamensdesign

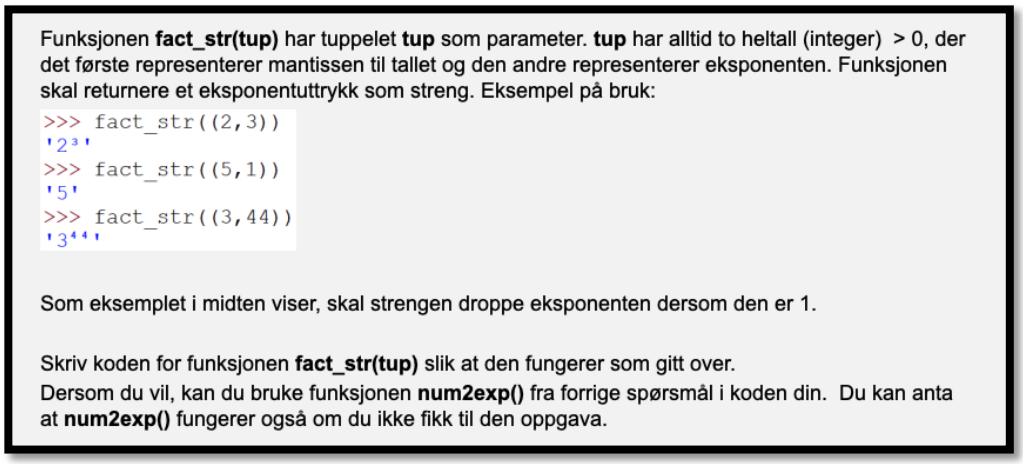
Emnet TDT4127 Programmering og Numerikk ble gitt første gang høsten 2018. Målgruppen er masterstudenter som blir tatt opp i 4.årskurs på ulike siv.ing.studier ved NTNU, etter tidligere å ha gått 3 år på ingeniørhøgskole, men med for lite IT-fag fra tidligere studier i forhold til kravene for siv.ing.grad. De fleste kan ikke programmere fra før (en uformell Kahoot! i første forelesning indikerte at bare 20% hadde noe erfaring med programmering fra tidligere), så emnet starter med helt grunnleggende konsepter á la et innføringskurs i programmering på bachelornivå.

Karakter var basert på en avsluttende digital eksamen. Eksamen hadde 19 oppgaver totalt, som vist i *Tabell 1*. De manuelt rettede oppgavene med skriving av kode, som er fokus for analysen i denne artikkelen, er merket med gult – altså oppgave 4, 5, 10, 13 og 18 i settet. Disse utgjorde tilsammen 30% av karaktervekten.

Tabell 1. Eksamensoppgaver i rekkefølge gitt i settet

Nr	Vekt%	Sjanger	Pensumdel	Inspera-Type	Retting
1	4	Forstå kode	Intro Prog	Paring	Auto
2	4	Forstå kode	Intro Prog	Paring	Auto
3	5	Fullfør kode	Intro Prog	Plasser i tekst	Auto
4	5	Skriv kode	Intro Prog	Grunnl. Prog.	Manuell
5	5	Skriv kode	Intro Prog	Programmering	Manuell
6	6	Fullfør kode	Intro Prog	Nedtrekk	Auto
7	5	Fullfør kode	Intro Prog	Dra og slipp	Auto
8	5	Fullfør kode	Intro Prog	Fyll inn tekst	Auto
9	5	Fullfør kode	Intro Prog	Dra og slipp	Auto
10	7	Skriv kode	Intro Prog	Programmering	Manuell
11	5	Fullfør kode	Intro Prog	Fyll inn tekst	Auto
12	5	Fullfør kode	Intro Prog	Nedtrekk	Auto
13	7	Skriv kode	Intro Prog	Programmering	Manuell
14	10	Forstå konsept	Numerikk	Flervalg	Auto
15	4	Fullfør kode	Numerikk	Nedtrekk	Auto
16	5	Fullfør kode	Numerikk	Dra og slipp	Auto
17	4	Feilretting	Numerikk	Flervalg	Auto
18	6	Skriv kode	Numerikk	Programmering	Manuell
19	3	Feilretting	Numerikk	Flervalg	Auto

Figur 1 viser oppgave 4, med faglærers forslag til løsning vist i Figur 2. Den foregående oppgave 3 hadde allerede presentert i en funksjon **num2exp()** som mottar et tall og returnerer tilsvarende superskript-streng (f.eks. `3` → `"³"`), så ved å kalle denne, unngår kandidaten utfordringer med å få til superskriptet. Oppgave 4 var ment å være enkel, men har likevel en del utfordringer for svakere studenter: (i) aksessere tallene i tuppelet, (ii) konvertere mantissen til en streng, (iii) teste om potens > 1, (iv) kalle num2exp() på eksponentdelen, (v) konkatenere de to delene av strengen, (vi) returnere resultatet.



Funksjonen **fact_str(tup)** har tuppelet **tup** som parameter. **tup** har alltid to heltall (integer) > 0, der det første representerer mantissen til tallet og den andre representerer eksponenten. Funksjonen skal returnere et eksponentuttrykk som streng. Eksempel på bruk:

```
>>> fact_str((2,3))
'2³'
>>> fact_str((5,1))
'5'
>>> fact_str((3,44))
'3⁴⁴'
```

Som eksemplet i midten viser, skal strengen droppe eksponenten dersom den er 1.

Skriv koden for funksjonen **fact_str(tup)** slik at den fungerer som gitt over.
Dersom du vil, kan du bruke funksjonen **num2exp()** fra forrige spørsmål i koden din. Du kan anta at **num2exp()** fungerer også om du ikke fikk til den oppgava.

Figur 1. Oppgave 4 i eksamenssettet

```
1  def fact_str(tup):
2      result = str(tup[0])
3      if tup[1] > 1:
4          result += num2exp(tup[1])
5      return result
```

Figur 2. Løsningsforslag oppgave 4

Oppgave 5 bygger videre på oppgave 4. Her skal det lages en funksjon **expr()** som får inn en liste av tupler og returnerer en streng av potensuttrykk forbundet med gangetegn. Av plasshensyn vises ikke hele oppgaveteksten, bare illustrasjonen på eksempel på bruk, jfr. Figur 3.

```
>>> expr( [] )
''
>>> expr( [(7,2)] )
'7²'
>>> expr( [(2,2), (3,2), (11,1)] )
'2²·3²·11'
>>> expr( [(2,13), (5,1), (7,2)] )
'2¹³·5·7²'
```

Figur 3. Utdrag fra oppgavetekst, oppgave 5 (bare eksempel på bruk)

En mulig løsning til denne oppgaven er vist i Figur 4. Oppgaven kan løses på flere måter, dels med mye kortere kode enn dette. Aller kortest, en funksjonskropp med kun én kodelinje, `return '·'.join([fact_str(t) for t in lst])`, men ingen kandidater på eksamen hadde noe så kompakt. Løsningen i figuren ligner på mange av de vellykkede løsningene, en annen typisk løsning var å ikke ha if-setning for tom liste men heller ha if-setning inni løkka for å unngå å sette gangetegn først (eller sist). Enkelte løste også oppgaven uten if-setning, ved å legge til gangetegnet i hver runde av løkka og deretter fjerne det overflødige tegnet etter løkka, f.eks. `return result.strip('·')`.

```
def expr(lst):
    if lst == []:
        return ''
    else:
        result = fact_str(lst[0])
        for i in range(1,len(lst)):
            result += '·' + fact_str(lst[i])
        return result
```

Figur 4. En mulig løsning til oppgave 5

De tre øvrige programmeringsoppgavene var mer komplekse, løsning vises derfor ikke av plasshensyn. Oppgave 10 gikk ut på å lese data fra en tekstfil, splitte hver tekstlinje i dataelementer, hvorav noen skulle konverteres fra tekst til tall, og samle alle dataene i en liste av lister som skulle returneres. Funksjonen skulle også håndtere eventuelle unntak, som at filen ikke kunne åpnes eller at data var i feil format. Faglærers løsningsforslag var på 22 linjer kode, hvorav 9 linjer for unntaksbehandling. Oppgave 13 gikk ut på å lage en funksjon som kunne ta inn en matrise med heltall som argument (gitt som en liste av lister) og returnere en liste med tuppel (radnr, kolonnenr) hvor det var

tilfelle at summen av tallene i raden var lik summen av tallene i kolonnen. Løsningsforslaget var 13 linjer langt, men algoritmisk mer komplekst enn oppgave 10, med to doble løkker og to lokale listevariable. Oppgave 18 var å skrive en funksjon for numerisk integrasjon etter Simpsons 3/8-metode, med parametre (f, a, b, n) hvor f er funksjonen som skal integreres, a og b start- og sluttpunktet, og n antall intervaller som skal benyttes. Løsningsforslaget var 11 linjer og ikke spesielt komplisert, men studenter som manglet forståelse for numerisk integrasjon, vil ha slitt med denne.

Forskningsmetode

Forskningen i artikkelen bygger primært på automatisk analyse av besvarelsene. FS1, i hvilken grad studentene skrev kode som ville ha kjørt korrekt, ble undersøkt ved å skrive et testskript i Python som iterativt importerte hver students løsning som en modul og kalte studentens funksjon for å se om den returnerte forventet resultat. En del kode ga unntak allerede ved import og lot seg ikke kjøre, på grunn av syntaksfeil. For kode som det lyktes å importere, kunne denne for hvert testtilfelle enten gi korrekt svar, galt svar, eller resultere i kjøretidsfeil.

Med hensyn på FS2, relatert til besvarelser som hadde fått full pott ved manuell sensur selv om koden ikke ville ha passert alle testene, ble testresultater sammenlignet med poenggivning fra manuell sensur. For to av deloppgavene ble dessuten besvarelser som hadde fått full pott selv om koden ikke ville ha kjørt feilfritt, inspisert manuelt for å se hva slags feil koden inneholdt og hvor mye retting som ville ha vært nødvendig. FS3, om hvilke typer feil studentene gjorde, baserer seg dels på automatisk telling av forekomster av forventete uttrykk i koden til alle studentene på den enkleste deloppgaven, samt manuell inspeksjon .

Resultater

Overordnet for de fem programmeringsoppgavene

Tabell 2 viser oversikt over resultater for de 5 programmeringsoppgavene. Kolonnen LOC viser antall kodelinjer i faglærers løsning av oppgaven. Kolonnen p-verdi – NB: her p-verdi (*p-value*) slik begrepet brukes i *item response theory*, **ikke** relatert til statistisk hypotesetesting – viser snittscore på hver oppgave justert til intervallet [0,1]. Snittscore for eksamen totalt sett var 62%, så oppgave 10, 13 og 18 ligger betydelig lavere enn dette, og var blant de vanskeligste oppgavene i eksamenssettet. Oppgave 4 var ment å være den letteste av de fem programmeringsoppgavene, men hadde forholdsvis dårlig resultat (0,61), svakere enn den påfølgende Oppgave 5 (0,65). Kolonnen **Korrelasjon** viser såkalt «item-rest correlation» for hver oppgave (dvs. score på denne oppgaven versus eksamenssettet forøvrig). Her er verdier over 0,6 ansett for å være meget bra. Kolonnen D-verdi viser oppgavens diskrimineringsindeks, dvs. i hvilken grad den skilte mellom sterke og svake studenter. D over 0,4 er ansett som meget bra. Tid+/- viser tidsforbruk på deloppgaven i forhold til oppgavens vekt. F.eks. hadde oppgave 4 en vekt på 5%, mens studentene i snitt brukte 4,2% på denne oppgaven, derfor -0,8 her. På Oppgave 10, som hadde en vekt på 7%, brukte studentene i snitt 9,3% av tiden sin, derfor +2,3 her. Kolonnen **Test** viser testscore på oppgaven, dvs. hva ville studenten ha scoret hvis den var blitt rettet kun automatisk basert på testtilfeller, med proporsjonal score for hvert enkelt testtilfelle (antall testtilfeller i parentes). En tendens her er at jo lenger kode (jfr. kolonnen LOC), jo lavere testscore. Oppgave 10, som krevde den lengste koden, hadde **ingen** korrekt kjørende besvarelser og dermed testscore 0,0 – det er utfordrende for studenter å skrive fungerende kode på 20+ linjer, særlig når de ikke har mulighet til å

teste koden. Kolonnen C (sensor-test) viser korrelasjonen mellom testscore og sensorscore. Denne er ganske høy for noen deloppgaver, lavere for andre – og forståelig nok 0 for Oppgave 10 hvor ingen studenter hadde korrekt kode.

Tabell 2. Kodeoppgavenes snittscore, rest-korrelasjon, diskriminering og tidsbruk

Oppgave	LOC	p-verdi	C (item-rest)	D-verdi	Tid +/-	Test	C (sensor-test)
4	5	0,61	0,62	0,53	-0,8	0,30 (2)	0,73
5	8	0,65	0,74	0,64	+0,4	0,27 (4)	0,60
10	22	0,53	0,70	0,55	+2,3	0,00 (3)	0,00
13	13	0,41	0,69	0,56	+1,1	0,04 (3)	0,35
18	11	0,49	0,63	0,71	-0,2	0,17 (3)	0,52

Oppgave 4: Konvertere tuppel til streng

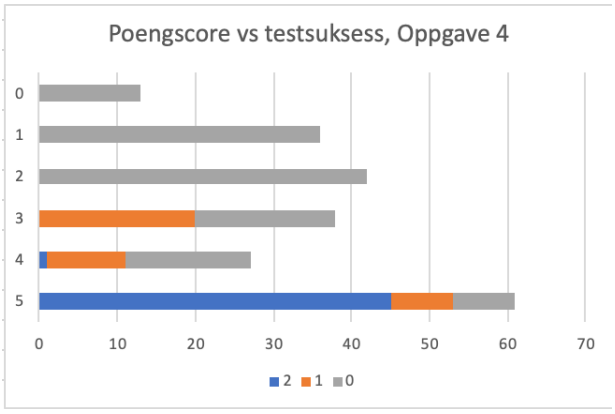
Oppgave 4 handlet som vist i Figur 1 og 2 om å konvertere et tuppel til en streng, f.eks. `(4,2)` → `'4²'`, hvor det allerede forelå en funksjon for å få til superskriptet. Det er naturlig å ha to testtilfeller for denne oppgaven, ett hvor potensen er 1, en annen hvor potensen er > 1. Som Tabell 3 viser, var det bare 26,4% som hadde kode som kjørte med korrekt resultat for potens 1, mens litt flere (34,9%) hadde korrekt returverdi for det generelle tilfellet. Mange hadde unnlatt å håndtere spesialtilfellet og returnerte strengen `'3¹'` gitt (3,1) som argument. Mer enn halvparten av studentene hadde kode som hadde syntaks- eller kjøretidsfeil og dermed ikke returnerte noen verdi.

Tabell 3: Testresultater for Oppgave 4

	Korrekt svar	Feil svar	Kjøretidsfeil	Syntaksfeil
Test potens = 1	26,4%	21,6%	27,4%	24,5%
Test potens >1	34,9%	7,0%	33,4%	

Figur 5 viser sammenhengen mellom testscore på oppgave 4 (dvs. om studentens kode passerte 0, 1 eller 2 tester) og poeng gitt av sensor ved manuell sensur (som var heltall 0-5). Over 60% av studentene hadde full score på deloppgaven. Et flertall av de som hadde full score (46 av 62) hadde kode som kjørte korrekt for begge testtilfellene, disse er vist med blått. Men det var også 8 som kun passerte 1 test (oransje) og 8 som passerte 0, men som likevel fikk full pott. Disse vil typisk ha hatt nesten korrekt kode, med småfeil som sensor enten overså eller valgte å ikke trekke for. Det er også én student som hadde kode som passerte begge testtilfeller med bare fikk 4 poeng. Denne hadde løst oppgaven på en tungvint måte, enten har sensor her trodd at koden inneholdt feil, eller valgt å trekke for tungvint løsning, men 4/5 representerer et trekk på 20%, og virker svært strengt gitt at 16 fikk 5/5 med kode som ikke virket. Alt i alt er det likevel et brukbart samsvar mellom testresultater og sensors manuelle vurdering, korrelasjonen var 0,73.

Selv om over 60% hadde full pott på oppgave 4, er en p-verdi på bare 0,61 (dvs. snittscore 3,05 av 5) lavt for den antatt letteste programmeringsoppgaven. Som figur 5 indikerer, vil automatisk kjøring mot testtilfeller i liten grad være egnet til å avdekke hva som var årsaken til at mange fikk lite poeng, siden studenter med null passerte tester fordelte seg over hele poengspekteret.



Figur 5. Resultat Oppgave 4, x-akse: antall studenter, y-akse: sensorscore, farger: passerte tester

Siden Oppgave 4 har ganske kort kode som er kjapt gjort å inspisere manuelt, kan det være spesielt interessant å se på koden til de 16 studentene som fikk full pott på tross av kode med feil. Hva slags feil hadde de gjort, og hvor kjapt ville det eventuelt ha vært å rette disse feilene hvis de hadde mulighet til å teste under eksamen og se at koden ikke virket? De vanligste feilene var:

- Syntaksfeil: 5 av 16 studenter hadde parentesfeil, enten manglende sluttparentes eller et overflødig parentessymbol. 3 hadde stavefeil / navnefeil, f.eks. at en variabel var kalt `result` ett sted i koden, `resultat` et annet sted.
- Semantiske feil: 6 studenter hadde feil betingelse i if-setning. Disse hadde gjort num2exp()-konverteringen tidlig i koden, og så i if-setningen testet om den resulterende strengen var lik `"1"` eller `1` (mens det riktige i gitt kontekst ville ha vært å teste mot superskript `"¹"`). 3 av studentene hadde andre semantiske feil, dels av en slik art at 5/5 poeng virker i overkant snilt.

En automatisk analyse som kan gi litt mer innsikt i årsaken til lav poengscore på oppgaven, er å søke etter kodefragmenter som man typisk ville forvente å finne i en korrekt løsning. For oppgave 4 ville de fleste korrekte løsninger typisk inneholde følgende regulære uttrykk (hvor annen tekst kan byttes inn for *): **`tup[*]`** (f.eks. `tup[0]`, `tup[1]` eller `tup[-1]` for å aksessere elementene i tuppelet), **`str(*)`** for å konvertere tallet for mantissen til streng, **`num2exp(*)`** for å konvertere potenstallet til en superskript, **`if`** for å teste om potensen skal vises, + for å konkatenere de to delene av strengen, og **`return`** for å returnere resultatet fra funksjonen. Tabell 4 angir hvor mange prosent av besvarelsene som inneholdt disse forventede uttrykkene. Som man kan se, hadde de aller fleste studentene (90%) en return-setning. De som ikke hadde det, hadde enten nesten blank kode, f.eks. kun funksjonshodet, eller de hadde print i stedet for return, som er en typisk tabbe nybegynnere gjør. For en del av de andre fragmentene er resultatene mye dårligere. Kun 71% av besvarelsene hadde if-setning. Dette indikerer at en grunn til at så få hadde korrekt test for potens 1 (jfr. Tabell 3), var at mange studenter forsøkte slett ikke å håndtere dette spesialtilfellet i koden sin. Enda dårligere resultater er det for aksessering av elementer i tuppelet og konvertering av mantissen til streng. Her må det riktignok bemerkes at det fins andre måter å oppnå det samme på. Elementer i tuppelet kan aksesseres uten bruk av indekser, f.eks. **`mantisse, eksponent = tup`**; og konvertering fra tall til streng kan gjøres ved bruk av f-string, f.eks. **`return f"{tup[0]}{num2exp(tup[1])}"`**. Manuell gjennomgang av besvarelsene viste likevel at det kun var et ensifret antall av kandidatene som hadde slike alternative løsninger; for de aller fleste innebar mangelen på de forventende uttrykkene at de ikke hadde klart å aksessere elementer fra tuppelet på riktig måte, og hadde glemt å konvertere mantissen til streng. Noen kandidater hadde de forventende uttrykkene i koden men feil utført, f.eks.

`str(tup[0]+num2exp(tup[1]))` hvor man prøver å konkatenere et heltall med en streng før man utfører strengkonverteringen.

Tabell 4: Forekomst (%) av forventende kodefragmenter, Oppgave 4

Regexp	tup[*]	str(*)	num2exp(*)	if	+	return
% inkl	64	63	82	71	75	90

Basert på hvor mange av de forventede uttrykkene en besvarelse inneholder, kan vi regne ut en omtrentlig «innholdsscore» for koden. Tabell 5 viser i hvilken grad denne samsvarer med poeng gitt av sensor på oppgaven (sensorpoeng 0-5 vist i radene, innholdsscore 0-6 vist i kolonnene).

Tabell 5: Resultat Oppgave 4, sensors poeng (rad) vs. # forventende kodefragmenter (kolonne)

Sensors poeng vs antall forventede uttrykk							
	6	5	4	3	2	1	0
5	53	8	0	0	0	0	0
4	14	12	1	0	0	0	0
3	1	20	16	1	0	0	0
2	0	16	16	8	1	1	0
1	1	3	10	8	11	3	0
0	0	0	0	0	0	4	9

Korrelasjonen mellom innholdsscore og sensorpoeng er 0,83, som er høyere enn korrelasjonen mellom sensorpoeng og testscore (0,73). Blant de 65 studentene som har full pott på deloppgaven, er det 53 som har alle de forventede uttrykkene, og 8 som mangler ett, enten fordi de har løst en av utfordringene på en alternativ måte, eller har fått 5/5 ved overseelse av feil fra sensor). Det fins også noen kandidater som har alle de forventede uttrykkene men likevel har dårligere score (14 med 4/5 poeng, 1 med 3/5 og 1 med bare 1/5). Dette vil typisk være kandidater som har med de forventede uttrykkene men har brukt dem på feil måte, f.eks. if-setning med feil betingelse, strengkonvertering med feil argument, return-setning uten korrekt returverdi.

Oppgave 5: Sette sammen en streng fra liste

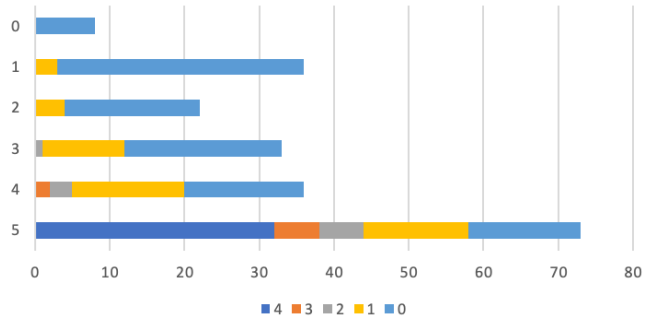
Tabell 6 viser resultater for testing av studentenes kode på Oppgave 5, med fire forskjellige argumenter: tom liste (som skal gi retur av tom streng), liste med bare ett element, to elementer og tre elementer. Andelen av kode som kjører korrekt er mindre for Oppgave 5 enn for Oppgave 4. Kun 16% av studentene hadde kode som passerte alle fire testtilfellene. Dog er det en vesentlig større andel (43%) som returnerer korrekt svar for spesialtilfellet med tom liste. De fleste håndterte dette helt i starten av funksjonen sin, med en if-setning som testet for tom liste, med påfølgende retur av tom streng, slik at returverdien for denne ble korrekt selv om man skulle ha gjort feil lenger nede i koden (med mindre det var syntaksfeil som hindret kjøring overhodet).

Tabell 6: Testresultater for Oppgave 5

	Korrekt svar	Feil svar	Kjøretidsfeil	Syntaksfeil
Test tom liste	43%	6%	15%	36%
Test 1 element	24%	14%	26%	
Test 2 element	20%	9%	34%	
Test 3 element	20%	8%	36%	

Figur 6 viser sammenhengen mellom sensorscore (rader) og testscore (farger) på Oppgave 5. Studentene som passerte alle de fire testene (mørkeblå) har full score på

oppgaven. Sensor ga også full score til en del studenter som har passert 3 (oransje), 2 (grå), 1 (gul) og 0 (lyseblå) tester. Jo nærmere man kommer 0 poeng, jo mer dominerende blir studenter som passerte null tester.



Figur 6. Resultat for Oppgave 5, x-akse: antall studenter, y-akse: sensorscore; farger: passerte tester

Også for Oppgave 5 ble det gjort en manuell inspeksjon av besvarelsene til de studentene som hadde fått full pott på tross av kode som ikke kjørte feilfritt. Vanlige feil:

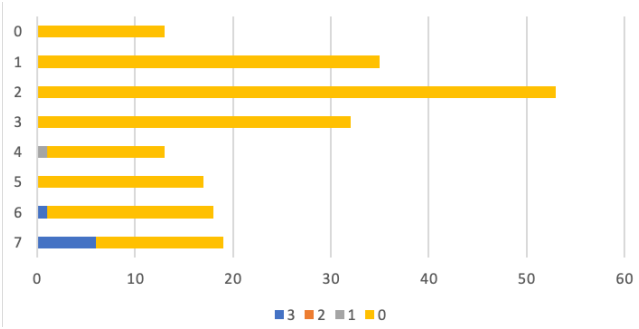
- Syntaksfeil: Feil med parentes eller streng-apostrof (7), manglende / overflødig kolon (4), stavefeil (7), navnefeil annet enn staving (5).
- Semantiske feil: Her var det mange ulike, ingen feil var felles for mange kandidater. For en del av kandidatene skyldtes feilen et mislykket forsøk på å unngå et overflødig gangetegn – kandidatene hadde med kodelinjer for dette, men de virket ikke helt som de skulle.

Oppgave 10: Lese fra tekstfil, håndtere unntak, returnere liste av lister

Oppgave 10 etterspurte en funksjon som skulle lese fra tekstfil og returnere en liste av lister med strenger, heltall og flyttall, samt håndtere unntak ved manglende fil eller feil dataformat. Ikke nødvendigvis den vanskeligste deloppgaven, men den klart lengste – løsning 22 kodelinjer. På denne oppgaven var det **ingen studenter som hadde kode som kjørte korrekt**. 69% hadde syntaksfeil, 31% hadde kode som kjørte men uten å resultere i korrekt returverdi. Det er dermed uinteressant å se noen figur av sensorscore vs.. Poengscore fra sensor dekket hele spekteret fra full pott 7/7 til 0/7, og ganske jevnt fordelt (prosenttall, full pott først): 13, 14, 10, 15, 13, 15, 15, 5. Sensor her valgt å la være å trekke for småfeil, og gitt delvis score for kode som hadde med deler av det som trengtes.

Oppgave 13: Sammenligne rader og kolonner i matrise

Oppgave 13 var algoritmisk mer kompleks enn Oppgave 10, men krevde kortere kode. Av 201 studenter som hadde skrevet noe på oppgaven, var det kun 7 (3,5%) som hadde kode som kjørte korrekt og passerte alle testene. 65% hadde kode som ikke kjørte i det hele tatt på grunn av syntaksfeil, og de resterende hadde kode som kjørte men returnerte feil resultat. Figur 7 viser sensorscore vs. testresultater. De studentene som hadde korrekt kjørende kode er vist med mørkeblått, disse har fått 7/7 poeng unntatt én som bare har fått 6/7. Dette kan klassifiseres som en sensortabbe. Kandidaten hadde løst oppgaven på en litt atypisk måte, og sensor må ha trodd at koden inneholdt en feil. På 4/7 poeng er det 1 kandidat som hadde kode som passerte 1 av 3 testtilfeller. At denne bare endte opp med 4/7 var rimelig, det var betydelige feil i koden.



Figur 7. Resultat Oppgave 13; x-akse: antall studenter, y-akse: sensorscore; farger: passerte tester

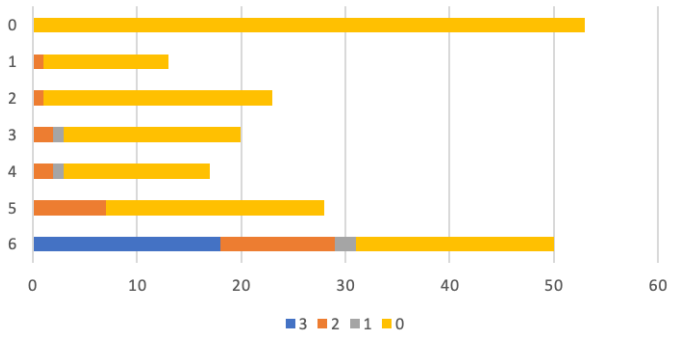
Igjen ser man at det er mange kandidater som har fått høy poengscore på deloppgaven selv om de ikke hadde kode som kjørte korrekt. Mange av disse har små syntaksfeil som sensor har valgt å ikke trekke for, eller sensor har oversett feil i kode som så tilforlatelig korrekt ut.

Oppgave 18: Numerisk integrasjon

Tabell 7 viser resultatene på Oppgave 18. Mer enn halvparten av studentene hadde kode som overhodet ikke kjørte pga. syntaksfeil, og av de som kjørte, var det en betydelig andel som hadde kjøretidsfeil eller returnerte feil svar. Om trent 20% returnerte riktig svar for testtilfeller der integrasjonen startet ved x=0, mens bare 9% når den startet med negativ x. Figur 9 viser at en relativt stor andel studenter hadde 0 poeng på denne deloppgaven, på grunn av nesten blanke svar (f.eks. kun skrevet funksjonshodet, som allerede var gitt i oppgaveteksten og dermed ikke ga poeng). Studenter med kode som passerte alle tre testene, fikk alle full pott 6/6 fra sensor. Det fins også studenter med kode som ikke passerte tester men likevel fikk full pott – samt studenter med kode som passerte en eller to av testene men fikk lavere poengsum.

Tabell 7: Testresultater for Oppgave 18

	Korrekt svar	Feil svar	Kjøretidsfeil	Syntaksfeil
Test -1,1	9%	15%	17%	57%
Test 0,2pi	20%	4%	17%	
Test 0,0	22%	1%	19%	



Figur 9. Resultat Oppgave 18; x-akse: antall studenter, y-akse: sensorscore, farger: antall passerte tester

Den store andelen av 0 poeng på denne deloppgaven skyldes at mange studenter hadde tilnærmet blanke svar, f.eks. kun funksjonshode og ordet return. Dels kan dette skyldes at det var en oppgave langt bak i settet, så mange hadde lite tid igjen. De som var svake i numerikk, vil kanskje også bare ha gitt opp denne deloppgaven.

Diskusjon og konklusjon

Relatert til FS1, i hvilken grad studentene hadde kode som kjørte korrekt, viser resultatene at dette var tilfelle i liten grad. Selv for den enkleste av programmeringsoppgavene var det bare en tredel av studentene som hadde en funksjon som returnerte korrekt svar, og andelen avtok for mer komplekse oppgaver. For den lengste oppgaven (10) var det ingen studenter som passerte noen av testtilfellene. Karaktersetting basert på testscore (andel testtilfeller passert) ville dermed ha gitt dramatisk dårligere eksamensresultater enn det som fremkom ved manuell sensur. **Men** – hvis studentene har mulighet til å teste koden sin underveis, som typisk vil være en forutsetning for bruk av testscore ved sensur, ville man anta at en del av de som her hadde ikke-fungerende kode, ville ha klart å rette opp i dette. Inspeksjon av koden til de som hadde fått full pott for ukorrekt kode på Oppgave 4 og 5 indikerte at dette ofte dreide seg om små feil som kunne ha vært kjapt å rette. For svakere studenter dreier det seg ofte om vesentlige mangler i koden, f.eks. at kodelinjer som var nødvendige for å løse oppgaven, rett og slett er utelatt. Selv for den antatt enkleste oppgaven (4) var det en relativt stor andel av studentene som enten manglet if-setning, typekonvertering eller korrekt aksess av elementer fra tuppel. For den mer omfattende Oppgave 10 manglet de fleste studenter noe av koden som skulle til, jfr. Figur 7. Studenter som ved manuell retting hadde middelmådig score på en deloppgave må derfor forventes å trenge vesentlig lenger tid for å komme fram til fungerende kode etter testing – om i det hele tatt. Selv om den nye oppgavesjangeren «Code-Compile» i Inspera gir studentene mulighet til å kjøre koden mot en testsuite, har den ikke noen nevneverdig funksjonalitet for debugging. Studier har vist at studenter syns bruk av en programmeringsomgivelse på eksamen føles mest naturlig [19] – Code Compile i Inspera vil nok føles mer naturlig enn papir, men mindre enn en programmeringsomgivelse på grunn av begrenset støtte. For studenter som har en del feil i koden sin, vil det lett bli slik at man prøver å kjøre, men det virker ikke på grunn av en syntaksfeil. Man retter syntaksfeilen, kjører på nytt – ny syntaksfeil. Etter hvert som man har rettet alle syntaksfeilene, kjører koden – men gir feil resultat. Hvis det blir mange slike runder, vil man trenge lang tid før koden fungerer.

Hvis man skal basere sensur kun på automatiske tester, vil fordelingen av poengscore lett bli mer bipolar. På oppgave10, der ingen studenter hadde korrekt kjørende kode, var likevel manuell poengscore fra sensor fordelt ganske flatt utover hele spekteret 0-7. Med testmulighet er det rimelig å anta at sterkere studenter som hadde kode nær ved å virke (typisk de som fikk 7/7 eller 6/7, til nød kanskje også 5/7) kunne ha rukket å rette koden og få den til å fungere i løpet av eksamen, mens de med midlere og lavere score ikke ville klare dette og i stedet ende opp med 0/7 hvor de nå kanskje hadde 4/7. Dette tilsier at:

- den enkleste deloppgaven bør gjøres tilstrekkelig enkel til at selv relativt svake studenter kan få koden til å fungere. Som Tabell 4 indikerer, hadde selv den letteste oppgaven med bare 5 linjer kode, en god del ulike ting studenten måtte huske på, og for svake studenter ble det en eller to komplikasjoner for mye.
- Oppgavesettet må ikke ha for mye tidspress, siden svake studenter vil bruke mye tid på å teste kode, finne og rette feil, før de vil ha noe som fungerer. Antall deloppgaver må da reduseres i forhold til eksamen uten testmulighet.

Et alternativ til fullstendig automatisk retting av programmeringsoppgaver basert på testsuiter, kan være en kombinasjon av automatisk og manuell retting. Hvis man f.eks. automatisk gir full pott til de som passerer testene, men manuelt ser igjennom de som ikke kjører korrekt for å se om de likevel skal ha noen poeng, vil dette spare noe tid i forhold til manuell retting av alt. Det kan også forhindre enkelte feil ved manuell sensur.

Selv med mulighet for automatisk retting av programmeringsoppgaver kan man også ønske oppgaver med kodeforståelse, siden evne til å forstå kode skrevet av andre er et relevant læringsutbytte. Noe man må være obs på da, er risikoen for at studenter kan bruke

en «Kompilering»-oppgave i eksamenssettet til å teste kode fra forståelsesoppgaver. Det tryggeste er å ha oppgaver med «Kompilering» bakerst i settet, uten mulighet for å navigere tilbake. Dette vil også forhindre at høy tidsbruk til feilretting gjør at man blir blank på andre deloppgaver. For oppgaver av typen «Kompilering» er det også viktig å ha andre tester i suiten som sensor kjører, enn det som gis til studentene – slik at det ikke er mulig å «løse» oppgaven ved å skrive noe enkel kode som akkurat passer til testsuiten.

Referanser

1. Sheard, J., et al., *Assessment of programming: pedagogical foundations of exams*, in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 2013, ACM: Canterbury, England, UK. p. 141-146.
2. Simon, et al. *Introductory programming: examining the exams*. in *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*. 2012. Australian Computer Society, Inc.
3. Petersen, A., M. Craig, and D. Zingaro. *Reviewing CS1 exam question content*. in *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011. ACM.
4. Hillier, M. and A. Fluck, *Arguing again for e-exams in high stakes examinations.* Electric dreams. proceedings ascilite, 2013: p. 385-396.
5. Sindre, G. and A. Vegendla, *E-exams and exam process improvement*, in *UDIT 2015*. 2015, Bibsys OJS: Ålesund.
6. Trovåg, J.M., *Digital eksamen–mer enn strøm på gamle eksamensoppgaver?* 2017, Høgskolen Vestlandet.
7. Qian, Y. and J. Lehman, *Students' misconceptions and other difficulties in introductory programming: A literature review.* ACM Transactions on Computing Education (TOCE), 2017. **18**(1): p. 1-24.
8. Kaczmarczyk, L.C., et al. *Identifying student misconceptions of programming*. in *Proceedings of the 41st ACM technical symposium on Computer science education*. 2010.
9. Hristova, M., et al., *Identifying and correcting Java programming errors for introductory computer science students.* ACM SIGCSE Bulletin, 2003. **35**(1): p. 153-156.
10. Brown, N.C. and A. Altadmri, *Novice Java programming mistakes: Large-scale data vs. educator beliefs.* ACM Transactions on Computing Education (TOCE), 2017. **17**(2): p. 1-21.
11. McCall, D. and M. Kölling. *Meaningful categorisation of novice programmer errors*. in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 2014. IEEE.
12. Kennedy, C. and E.T. Kraemer. *What Are They Thinking? Eliciting Student Reasoning About Troublesome Concepts in Introductory Computer Science*. in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. 2018.
13. Kohn, T. *Variable evaluation: An exploration of novice programmers' understanding and common misconceptions*. in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 2017.
14. Kallia, M. and S. Sentance. *Learning to use functions: The relationship between misconceptions and self-efficacy*. in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019.
15. Rørnes, K.M., R.K. Runde, and S.M. Jensen. *Students' mental models of references in Python*. in *Norsk IKT-konferanse for forskning og utdanning*. 2019.
16. Johnson, F., S. McQuistin, and J. O'Donnell. *Analysis of Student Misconceptions using Python as an Introductory Programming Language*. in *Proceedings of the 4th Conference on Computing Education Practice 2020*. 2020.
17. Barstad, V., M. Goodwin, and T. Gjøsæter. *Predicting source code quality with static analysis and machine learning*. in *Norsk IKT-konferanse for forskning og utdanning*. 2014.
18. Veerasamy, A.K., D. D'Souza, and M.-J. Laakso, *Identifying novice student programming misconceptions and errors from summative assessments.* Journal of Educational Technology Systems, 2016. **45**(1): p. 50-73.
19. Rytkönen, A. and V. Virtakoivu. *Comparative Student Experiences on Electronic Examining in a Programming Course-Case C*. in *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 2019.