Obligatorisk innlevering 2, høsten 2014, INF3331

Kristoffer Brabrand <kristrek@student.matnat.uio.no>

October 23, 2014

Innledende beskrivelse

Omfanget av denne oppgaven er vesentlig større enn forrige oppgave, og jeg vil nedenfor redegjøre i korte trekk for flyten i prepro.py for å gjøre det litt klarere hva som skjer.

1. Parsing av kommandolinjeparametre.
2. Sjekker at inputfilen finnes og laster filen.
3. Legger til instruksjoner nødvendig for pretty-printing (hvis pretty-printing er aktivert).
4. Prosessering av innholdet i filen.
   (a) Erstatter alle input-instruksjoner med filen de referer til og prosesserer den inkluderte filen som beskrevet i pkt. 1 m/ underpunkter (rekursjon).
   (b) Setter inn kildekode referert til med `%@import`.
   (c) Setter inn eksekveringsresultat referert til med `%@exex`.
   (d) Setter inn inline-kode referert til med `%@import` eller `%@exec`.
   (e) Setter inn eksekveringsresultat av inline kode refert til med `\%@bash` eller `%@python`.
5. Skriv den bearbeidede filen til den spesifiserte output-filen.

Parametrene prepro.py tar vises enklest ved å kalle den med -h som parameter. Outputen som gis når man kaller `prepro.py -h` er vist nedenfor.

Terminal

```
$ python prepro.py -h
usage: prepro.py [-h] [--pretty] [--verbose] source dest

Preprocess latex file.
```

```
positional arguments:
  source         Path to preprocess source file
  dest           Destination folder for the processed file

optional arguments:
  -h, --help     show this help message and exit
  --pretty, -p   Enable fancy verbatims
  --verbose, -v  Enable verbose script output
```

Oppgave 1: Kodeimport

Oppgaven er løst ved bruk av et regulært uttrykk som går over flere linjer. Den ser etter et innledende linjeskift før mønsteret for kodeimporten og deretter én eller flere linjer før avslutningstaggen.

Jeg har brukt følgende mønster for å matche kodeimport-instruksjoner;

```
\n(%@import ([^\ ]+) (.*))\n
```

Når mønsteret er funnet i latex-dokumentet blir innholdet i filen det referers til kjørt mot det regulære uttrykket, og det første treffet på dette blir returnert.

Kodeimporten gjøres som en del prosesseringen som gjøres når `prepro.py` kjøres på en latex-fil. Implementasjonen finnes i src/code_import.py.

Oppgave 2: Eksekvering av script

I denne oppgaven skjer matchingen på samme måte som i forrige oppgave, men når et treff for mønsterest er funnet kjøres kommandoen og resultatet av kjøringen settes inn i dokumentet.

Selve kjøringen av kommandoen gjøres ved hjelp av `subprocess`-modulen og jeg bruker `shlex`-modulen for å splitte kommando-parametrene til et array med parametre som `subprocess.Popen` er fornøyd med.

Koden nedenfor er det som skal til for å splitte en kommando-string til et array med parametre, kjøre denne og returnere output;

```
    # Parse/split the command into arguments
    arguments = shlex.split(command);

    try:
        # Open a sub process with the arguments
        process   = subprocess.Popen(arguments, stdout=subprocess.PIPE);

        # Get the piped output and return
        out, err = process.communicate();
    except OSError as e:
        raise Exception('Execution of command failed');
```

Eksekvering av scripts skjer som en del av prosesseringen som gjøres når prepro.py kjøres på en latex-fil. Implementasjonen finnes i src/script_execution.py.

Oppgave 3: Kodeformatering

Kodeformateringen i oppgaven håndteres med verbatim-blokker i det ferdige latex-dokumentet. Ettersom dette er noe som brukes av nesten alle modulene har jeg valgt å skille ut denne logikken i egne metoder; verbatim_exec og verbatim_code i verbatim-modulen.

Ved å kalle disse uten pretty-parametert (eller med parameteret satt til False) omsluttes stringen som sendes inn i metodenen som første parameter (navngitt hhv. code og result) av instruksjoner som gir en enkel vebatim-blokk.

Ved å sende med True for pretty-parametert gis det en litt stiligere verbatim-blokk.

Verbatim-metodene kan brukes slik;

```
# Fancy verbatim exec
print verbatim_exec('echo 2', pretty=True);

# Plain verbatim code
print verbatim_code('print 2');
```

Kodeformatering gjøres som del av de ulike leddene i prosesseringen som gjøres når prepro.py kjøres på en latex-fil. Implementasjonen finnes i src/verbatim.py.

Oppgave 4: Vanlig innliming av kode

I denne oppgaven bruker jeg i likhet med de fleste andre verbatim-modulen for formatering. Denne oppgaven ble med dette redusert til å gjøre en multiline-regex for å finne kodeblokker og deretter kalle på `verbatim.verbatim_code` eller `verbatim.verbatim_exec` avhengig av om det var en `%import`- eller `\%@exec`-instruksjon.

Jeg bruker dette regulære uttrykket for å finne kode som er limt inn.

```
(%@(import|exec)\n((.*\n)+?)%@)
```

Når uttrykket matches mot teksten gir det en gruppe med nøkkelordet (import eller exec) i, og jeg bruker dette til å finne ut om jeg skal pakke linjene mellom start og sluttaggen i code- eller exec-verbatim.

Formatering av innlimt kode er et av leddene i prosesseringen som gjøres når prepro.py kjøres på en latex-fil. Implementasjonen finnes i src/inline_blocks.py.

Oppgave 5: Kode-eksekvering

Jeg har løst denne oppgaven ved å først matche alle blokker med kode i teksten. Dette regulære uttrykket lignet mye på det som ble brukt i oppgave 4, men denne gangen med ytterligere én gruppe som inneholder scriptnavnet og evt. parametre vi vil det skal se ut som om vi har kjørt.

Implementasjonen støtter seg på subprocess.Popen for å starte en subprosess. Kommandoen som brukes for å kjøre python- og bashkoden er `bash -c 'kode her..'` og `python -c 'kode her...'`.

Output fra stdout pipes og process.communicate() brukes for å få tak i innholdet. Til slutt pakkes det hele inn i en verbatim-blokk ved bruk av `verbatim_exec` fra verbatim-modulen.

Formatering av innlimt kode er et av leddene i prosesseringen som gjøres når prepro.py kjøres på en latex-fil. Implementasjonen finnes i src/inline_blocks.py.

Oppgave 7: Kompilering av preprosessert fil

Kompilatoren er så begrenset i omfang at jeg valgte å implementere den som én enkelt fil. Kompilatoren starter en subprosess av `pdflatex` ved hjelp av `subprocess.Popen`

Som standard kjøres `pdflatex` i nonstopmode ved at interaction-flagget settes til nonstopmode. Dette kan deaktiveres ved å sette interactive-flagget når compile.py-scriptet kalles. Output fra compile.py går – som i `pdflatex` – som standard til den samme mappen som latex-filen som prosesseres ligger i. Dette kan overstyres ved å kalle compile.py med et destination-parameter.

Alle parametre til compile.py vises enkelt ved å kalle `python compile.py -h`. Dette gir følende output;

Terminal

```
$ python compile.py -h
usage: compile.py [-h] [--interactive] [--destination D] [--verbose] S

Compile PDF from latex file.

positional arguments:
  S                     Latex source file

optional arguments:
  -h, --help            show this help message and exit
  --interactive, -i     Interact with the latex compiler
  --destination D, -d D
                        Path to where the compiled file should be put
  --verbose, -v         Enable verbose script output
```

Oppgave 8: Inkludering av filer

Oppgaven går i korte trekk ut på å gjenskape latex sin enkle input-funksjonalitet – denne går ut på å putte innholdet fra en fil inn i en annen. Disse instruksjonene matches med et regulært uttrykk, og filnavnet trekkes ut slik at innholdet kan puttes inn i filen som inkluderer den.

Et problem som oppstår så fort man ikke har alle filene i én mappe er at relative stier blir feil hvis en fil inneholder en relativ sti og legges inn i en fil som lå på et annet nivå. For å løse dette sender jeg stien til filen som inkluderer

som `base_path` slik at filer som blir inkludert har en referanse til hvor de lå før de ble slått sammen.

For å håndtere uthenting av mappesti for filer og sammenslåing av relative stier og base_path har jeg brukt `os.path`.

Inkludering av filer er det første leddet i prosesseringen som gjøres når prepro.py kjøres på en latex-fil. Hele preprosesseringen kjøres for hver fil som inkluderes, slik at alle instruksjoner i alle inkluderte filer blir utført slik brukeren forventer. Implementasjonen finnes i src/latex.py.

Oppgave 11: Front-end til preprosessor

Denne oppgaven er løst og vist som en del av de andre oppgavene. Man kan se det brukervennligegrensesnittet ved å kalle `prepro.py` og `compile.py` med parameteret -h. Slik;

Terminal

```
$ python prepro.py -h
usage: prepro.py [-h] [--pretty] [--verbose] source dest

Preprocess latex file.

positional arguments:
  source         Path to preprocess source file
  dest           Destination folder for the processed file

optional arguments:
  -h, --help     show this help message and exit
  --pretty, -p   Enable fancy verbatims
  --verbose, -v  Enable verbose script output
```

og

Terminal

```
$ python compile.py -h
usage: compile.py [-h] [--interactive] [--destination D] [--verbose] S

Compile PDF from latex file.

positional arguments:
  S                     Latex source file

optional arguments:
  -h, --help            show this help message and exit
  --interactive, -i     Interact with the latex compiler
  --destination D, -d D
                        Path to where the compiled file should be put
  --verbose, -v         Enable verbose script output
```

Oppgave 12: Testing og dokumentasjon

Samtlige interne funksjoner i preprosessoren er testet med doctester som tester forventet respons, og `exceptions` dersom gal/ugyldig input gis til metodene.

Doctestene kan kjøres ved å navigere inn i src-mappen og kjøre hver enkelt modul slik; `python code_import.py`.

Test suiten krever `nose`, og kjøres ved å utføre følgende kommando i roten av mappen oblig2;

Terminal

```
nosetests
```

Oppgave 13: Rapport

Du har lest rapporten nå. Hvis du har lyst til å generere den kan du kjøre følgende kommando fra roten av oblig2;

Terminal

```
python prepro.py --pretty report/oblig2.tex oblig2.tex
python compile.py oblig2.tex
```