TDT4113 - Datateknologi, programmeringsprosjekt

Oppgave 4: Kalkulator

Dette dokumentet beskriver den fjerde oppgaven i ProgLab 2, "Kalkulator". For denne oppgaven gjelder at:

- Oppgaven skal løses individuelt
- Oppgaven skal løses med objektorientert kode skrevet i Python
- Fristen for oppgaven er *2 uker*, dvs. implementasjonen din lastes opp på Blackboard senest 12/10 kl 0800 og demonstreres senest kl 16:00 samme dag.

1 Bakgrunn for oppgaven

Når du er ferdig med denne oppgaven har du laget en kalkulator som tar inn en regne-oppgave skrevet som en tekst-streng, oversetter teksten til en dertil egnet representasjon ("Reverse Polish Notation" a.k.a. RPN, vi kommer tilbake til denne), og evaluerer RPN-representasjonen for å finne svaret. Vi antar at input-teksten er formelt riktig, så vi får ikke tekst av type "`3 + 3 *`". "Språket" kalkulatoren skal støtte er fleksibelt: Den skal håndtere de fire regneartene (addisjon, subtraksjon, multiplikasjon og divisjon), parenteser (inkludert nøstede parenteser), og vanlige matematiske funksjoner som `exp` og `log`.

2 Hjelpeklasser og generelt oppsett

Vi starter arbeidet med å lage noen hjelpe-klasser. Om du ikke forstår poenget med disse eller hvordan de skal brukes kan det være lurt å lese hele dokumentet før du begynner på implementasjonen.

2.1 Kø og Stack

Det meste av beregningene foregår ved hjelp av en vanlig kø (en liste der det elementet som ble satt inn først er det vi tar ut først, altså "first-in first-out") og en stack (en kø

med "last-in, first-out" prinsippet). Dere skal implementere disse to selv. Ettersom de to klassene er ganske like er det lurt å lage en super-klasse `Container`, for så å lage sub-klassene `Stack` og `Queue` etterpå. Det er ikke vanskelig, det kan for eksempel være en Python-`list` med egnede metoder rundt. Det er nyttig å se litt på hva en Python-`list` kan gjøre, og som du kan bruke "gratis" i din kode. Om du sjekker dokumentasjonen vil du for eksempel se at `list.append(element)` legger til `element` bakerst i listen, som er alt vi trenger for å implementere `push`. Du kan sjekke hvilket element som er på en gitt posisjon i en liste med "slicing", så for eksempel gir `list[-1]` det **siste** elementet (som tilsvarer `peek` i en `Stack`) mens `list[0]` gir det **første** elementet, som dermed er `peek` for en `Queue`. Endelig har `list` i Python allerede en `list.pop()` som pop'er av og returnerer det elementet som er sist i listen (og dermed er relevant for `Stack`-klassen din). Hvis du kaller `list.pop()` med et argument `index` vil den pop'e elementet på den plassen i listen, så `list.pop(0)` er det du trenger for `Queue`-klassen. Starter vi med `Container` kan den se ut som dette:

```
class Container:
    def __init__(self):
        self._items = []
    def size(self):
        # Return number of elements in self._items
    def is_empty(self):
        # Check if self._items is empty
    def push(self, item):
        # Add item to end of self._items
    def pop(self):
        # Pop off the correct element of self._items, and return it
        # This method differs between subclasses, hence is not
        # implemented in the superclass
        raise NotImplementedError
    def peek(self):
        # Return the top element without removing it
        # This method differs between subclasses, hence is not
        # implemented in the superclass
        raise NotImplementedError
```

Sub-klassingen er deretter gjort ved å overskrive `pop` og `peek`. Dermed kan `Queue` for eksempel se slik ut (og mye det samme for `Stack`, men med "last-in", "first-out"):

```
class Queue(Container):
    def __init__(self):
        # Initialization is done at superclass
        super(Queue, self).__init__()

    def peek(self):
        # Return the *first* element of the list, do not delete it
```

```
        assert not self.is_empty()
        return self._items[0]

    def pop(self):
        # Pop off the first element
        assert not self.is_empty()
        return self._items.pop(0)
```

Implementasjon – Del 1:
Implementer ferdig `Container`, `Queue` og `Stack`, slik at alle metodene gitt over har innhold. Lag en "unit-test" der du kontrollerer at oppførselen er som du forventer – for eksempel kan du legge inn noen elementer i en stack, og så lage en løkke som så lenge stacken ikke er tom tar av ett og ett element, skriver det ut, og forteller hvor mange elementer som er igjen på stacken. Gjør samme testen for `Queue` klassen.

2.2 Function

Det neste vi trenger er å definere hjelpeklasser for regneartene og for funksjoner kalkulatoren skal støtte. La oss starte med funksjonene, der du kan bruke denne klassen:

```
class Function:
    def __init__(self, func):
        self.func = func

    def execute(self, element, debug=True):
        # Check type
        if not isinstance(element, numbers.Number):
            raise TypeError("Cannot execute func if element is not a number")
        result = self.func(element)
        # Report
        if debug is True:
            print("Function: " + self.func.__name__
                  + "({:f}) = {:f}".format(element, result))
        # Go home
        return result
```

Det denne klassen gjør er følgende: Når du kjører `__init__` bindes instansen opp mot funksjonen som gis som input i `func`. Senere kan vi kjøre instansens `execute`-metode, og få evaluert funksjonen i `self.func` for et gitt argument. På ett vis ser dette ut som en kronglete måte å evaluere funksjoner på, men det vil vise seg fordelaktig senere at alle funksjoner som

kalkulatoren kjenner til er tilgjengelig gjennom et vel-definert interface, og det er også nødvendig for oss å kunne gjenkjenne objekter som `Function`-er. Vi kan nå få vite om elementet `element` er en funksjon ved å bruke kallet `isinstance(element, Function)`. Legg merke til at `execute` sjekker at den mottar et tall. Her benyttes definisjonen av tall fra pakken `numbers`, så du vil trenge `import numbers` i koden din for at dette skal virke.

Vi vil benytte oss av `numpy` for å definere den faktiske numeriske beregningen, så hvis du ikke har denne pakken installert må du laste den ned nå med kommandoen `pip install numpy`. `numpy` må deretter importeres inn i programmet. `numpy` inneholder mange matematiske funksjoner, for eksempel `numpy.exp`, `numpy.log`, og du kan gi kalkulatoren din tilgang til dem ved å "pakke dem inn" i `Function` - klassen slik:

```
exponential_func = Function(numpy.exp)
sin_func = Function(numpy.sin)
print(exponential_func.execute(sin_func.execute(0)))
```

Implementasjon – Del 2:
Implementer `Function` – eller bruk implementasjonen over. Prøv selv-testen og sjekk at det kommer riktig resultat.

2.3 Operator

På samme måte som vi pakker inn funksjoner vil vi også pakke inn regne-operasjonene. Du må lage en klasse `Operator` som eksekverer en av `numpy.add`, `numpy.multiply`, `numpy.divide`, og `numpy.subtract`. En `Operator` er **nesten** som en `Function`, men det er to viktige forskjeller: For det første må en operator ha **to** input til sin `execute`-metode. For det andre må regneartene vite hvor "sterke" de er. Når vi skal løse ut et uttrykk som "`1 + 2 * 3`" er svaret `7`, ikke `9`. Vi evaluerer fra venstre mot høyre, men ettersom multiplikasjon har rangen over addisjon må vi utføre `2 * 3 = 6` først, deretter `1 + 6 = 7`. Vi får ikke bruk for operator-presedens før helt mot slutten av oppgaven, men sørg for å lage en god implementasjon av `Operator` med en gang. Det gir oss muligheten for å kjøre noe a la dette:

```
add_op = Operator(operation=numpy.add, strength=0)
multiply_op = Operator(operation=numpy.multiply, strength=1)
print(add_op.execute(1, multiply_op.execute(2, 3)))
```

Implementasjon – Del 3:
Implementer `Operator`, og verifiser at alt virker ved hjelp av testen.

2.4 Calculator

Hovedklassen i denne oppgaven er `Calculator`-klassen. Det første du skal gjøre er å lage `Calculator` sin `__init__`-metode, der du gir kalkulatoren tilgang til regneartene og funksjonene. Det er lurt å ha disse i `dictionary`-er, der du bruker navnet til operasjonen som `key`. I tillegg trenger kalkulatoren lagringsplass for regne-oppgaven sin. Når systemet ditt skal gå gjennom en regne-oppgave må det iterere seg gjennom uttrykket fra venstre til høyre, så for å gjøre implementasjonen enkel bruker vi en "first-in first-out"-kø til dette. Dette har du allerede implementert, så nå skal `Queue`-klassen i arbeid.

Dermed kan `__init__` for eksempel se ut som dette:

```
def __init__(self):
    # Define the functions supported by linking them to Python
    # functions. These can be made elsewhere in the program,
    # or imported (e.g., from numpy)
    self.functions = {'EXP': Function(numpy.exp),
                      'LOG': Function(numpy.log),
                      'SIN': Function(numpy.sin),
                      'COS': Function(numpy.cos),
                      'SQRT': Function(numpy.sqrt)}

    # Define the operators supported.
    # Link them to Python functions (here: from numpy)
    self.operators = {'PLUSS': Operator(numpy.add, 0),
                      'GANGE': Operator(numpy.multiply, 1),
                      'DELE': Operator(numpy.divide, 1),
                      'MINUS': Operator(numpy.subtract, 0)}

    # Define the output-queue. The parse_text method fills this with RPN.
    # The evaluate_output_queue method evaluates it
    self.output_queue = Queue()
```

Du kan sjekke at det virker med å kjøre dette:

```
calc = Calculator()
print(calc.functions['EXP'].execute(
    calc.operators['PLUSS'].execute(
        1, calc.operators['GANGE'].execute(2, 3))))
```

Implementasjon – Del 4:
Implementer klassen, og sjekk at alt virker ved hjelp av testen.

3 Evaluere "Reverse Polish Notation"

"Reverse Polish Notation" (RPN) er en måte å skrive regnestykker på som ikke krever parenteser. Regelen er at en operator eller funksjon kommer etter operanden(e) sin(e). Så, i stedet for å skrive "`1 + 2`" vil man i RPN skrive "`[1, 2, +]`", og "`exp(7)`" blir "`[7, exp]`". Bare operasjonene er sortert riktig kan RPN beskrive alle slags uttrykk uten å bruke parenteser; for eksempel er "`(1 + 2) * 3`" gitt som "`[1, 2, +, 3, *]`" i RPN.

Nå skal du lage den delen av kalkulatoren som får inn en oppgave i RPN og løser denne. Når man skal løse ut RPN er algoritmen ganske enkel. Regne-oppgaven kommer inn som en **kø**, og vi bruker `self.output_queue` som ble definert i `__init__` til dette. Hvert element i køen er et tall, en funksjon, eller en operator. For å gjøre beregningen trenger du en `Stack` for mellom-lagring.

Pseudo-koden for å evaluere RPN er som følger:

1. Gå gjennom hvert element i køen ved å pop'e dem av en etter en:
   (a) Hvis elementet er et tall skal du pushe det på stack'en. Du kan sjekke typen til elementet med `isinstance`.
   (b) Hvis det er en funksjon skal du pop'e ett element av stacken, og evaluere funksjonen med dette elementet (som er et tall hvis RPN-syntaksen er riktig). Resultatet pusher du på stacken.
   (c) Hvis elementet er en operasjon må du pop'e to elementer av stacken. Gjør operasjonen med disse to elementene, og push resultatet tilbake på stacken. Pass på rekkefølgen av elementene: Hvis stacken din har elementene [2, 1] og du skal gjennomføre en subtraksjon er svaret gitt som `2 - 1 = 1` og ikke `1 - 2 = -1`. Noe a la `value = element.execute(stack.pop(), stack.pop())` er dermed **galt**; du må pop'e av de to elementene først til lokale variable, og deretter sende dem inn til `execute` i riktig rekkefølge.
2. Nå som køen er tom er det **ett** element på stacken. Dette er svaret du skal returnere.

Dette eksempelet viser hvordan vi beregner at `exp(1 + 2 * 3) = 1096.63` ved å håndtere køen [`1` , `2` , `3` , `multiply` , `add` , `exp`].

Element	Handling	Stack
1	`stack.push( 1 )`	1
2	`stack.push( 2 )`	1, 2
3	`stack.push( 3 )`	1, 2, 3
multiply	`multiply.execute( 2 , 3 ) ⇒ 6`	1, ~~2~~, ~~3~~
	`stack.push( 6 )`	1, 6
add	`add.execute( 1 , 6 ) ⇒ 7`	~~1~~, ~~6~~
	`stack.push( 7 )`	7
exp	`exp.execute( 7 ) ⇒ 1096.63`	~~7~~
	`stack.push( 1096.63 )`	1096.63

Implementasjon – Del 5:
Implementer RPN-beregningen, og lag en test for systemet ditt som beregner eksempelet og kontrollerer at svaret blir $\exp(7) \approx 1096.63$. For å gjøre testen må du bygge opp `calc.output_queue`. Til dette kan du benytte deg av operasjoner som `calc.output_queue.push(1.)` og `calc.output_queue.push(calc.functions['EXP'])`.

4 Fra "vanlig" notasjon til RPN

Kalkulatoren gjør ikke så mye nytte dersom den kun kan håndtere input i form av RPN. I siste del av oppgaven skal vi lage en parser som får en streng som input, og oversetter det til RPN. Når denne er på plass kan vi så evaluere teksten, og vi er i stand til å gjøre beregninger automatisk. Parseren skal kjenne igjen de fire regneartene, alle funksjonene, og kunne løse ut vilkårlig nøstede parentes-uttrykk. Men først skal du lage den delen som får elementene i regnestykket (tall, operatorer, funksjoner, parenteser) på "vanlig vis" og stokker det om til RPN. Etterpå står tekst-parseren for tur.

For å bygge RPN-køen skal vi bruke *shunting-yard* algoritmen. Denne algoritmen forutsetter at du har en kø med elementer som er hentet ut fra teksten. Her er hvert element enten et tall, en operasjon, en funksjon, en start-parentes, eller en slutt-parentes. Algoritmen ser på elementene en etter en ("first-in, first-out"), og bygger opp en **output-kø**. Noen elementer må mellom-lagres på en **operator-stack** for å sørge for at ordningen av elementene blir riktig.

Pseudo-koden er som følger:

1. Gå gjennom hvert element i input-køen:
   (a) Hvis elementet `elem` er et tall skal du pushe det på **output-køen**.

(b) Hvis `elem` er en funksjon skal pushe den på **operator-stacken**.

(c) Hvis `elem` er en start-parentes skal den pushes på **operator-stacken**.

(d) Hvis `elem` er en slutt-parentes:

- Pop av elementer fra **operator-stacken** en etter en og legg dem til **output-køen** fortløpende. Gjør dette helt til det øverste elementet på operator-stacken er en start-parentes.
- Pop av start-parentesen fra **operator-stacken**, og kast den.
- Vi kaster `elem` (slutt-parentesen) også.

(e) Hvis `elem` er en av de fire regneartene må du sortere det inn på riktig sted, og det betyr blant annet å passe på styrkeforholdet mellom operasjonene. Det gjør vi ved å pop'e av elementer fra **operator-stacken** og pushe dem på **output-køen** helt til vi er "ferdig". Vi fortsetter med å flytte elementer en etter en så lenge det er elementer på operator-stacken, og det øverste elementet enten er en **funksjon** eller en **operator** som er minst like sterk som `elem`. For å få til dette må du bruke `peek` for å smugtitte på topp-elementet på stacken, `isinstance` for å se hva slags objekt det er der, og ikke pop'e av stacken før du har bestemt deg for å flytte elementet over på køen. Når du er ferdig med å flytte elementer fra operator-stacken til output-køen må du pushe `elem` på **operator-stacken**.

2. Pop av hvert element på **operator-stacken** og push dem på **output-køen**.

Dette eksempelet viser hvordan vi oversetter `exp(1 + 2 * 3)` til RPN:

Element	Output-kø	Operator-Stack
`exp`		`exp`
`(`		`exp` , `(`
1	1	`exp` , `(`
`add`	1	`exp` , `(` , `add`
2	1 , 2	`exp` , `(` , `add`
`multiply`	1 , 2	`exp` , `(` , `add` , `multiply`
3	1 , 2 , 3	`exp` , `(` , `add` , `multiply`
`)`	1 , 2 , 3 , `multiply`	`exp` , `(` , `add` , ~~`multiply`~~
	1 , 2 , 3 , `multiply` `add`	`exp` , `(` , ~~`add`~~
	1 , 2 , 3 , `multiply` `add`	`exp` , ~~`(`~~
	1 , 2 , 3 , `multiply` , `add` , `exp`	~~`exp`~~

Legg merke til at det aldri er slutt-parenteser på operator-stacken, og at det aldri er parenteser i det hele tatt i output-køen. Start-parentesene brukes på operator-stacken, men løses ut før elementene flyttes til output-køen.

Det neste eksempelet oversetter `2 * 3 + 1`. Multiplikasjon har presedens over addisjon, så når vi skal putte `add` på operator-stacken må `multiply` først flyttes over på output-køen.

Element	Output-kø	Operator-Stack
2	2	
multiply	2	multiply
3	2 , 3	multiply
add	2 , 3 , multiply	~~multiply~~
	2 , 3 , multiply	add
1	2 , 3 , multiply , 1	add
	2 , 3 , multiply , 1 , add	~~add~~

Implementasjon – Del 6:
Implementer shunting-yard, og lag en test for å sjekke at implementasjonen virker.

5 Tekst-parseren

Nå skal du lage den delen av programmet som kjenner igjen de forskjellige delene i input-teksten og produserer elementene som "shunting-yard" algoritmen skal håndtere. Denne metoden vil dermed motta en tekst-streng, for eksempel `"2 * 3 + 1"`, og produsere `[2, multiply, 3, add, 1]`. Her er elementene i listen Python-objekter, så `2` er en `float`, `multiply` er en `Operator`, og så videre.

Det er anbefalt å bruke regex uttrykk her, som er implementert i `re`; `re` er en del av standard-distribusjonen av Python så du kan importere den direkte. Hvis du ikke husker hvordan regex'ene er definert kan det være verdt det å sjekke en tutorial online. For eksempel vil `check = re.search("^[−0123456789.]+", txt)` se på tekst-strengen gitt i `txt`, og se etter minst en av elementene av symbolene "-", "." eller tallene fra "0" til "9". Det betyr at vi vil for eksempel kjenne igjen `"−322.7623"` som en match. Dette uttrykket vil også tro at `"−3−2"` er et enkelt tall, men vi kommer oss unna den typen problemer med å re-definere hvordan vi skriver inn operatorene (vi kommer tilbake til dette). Når søkestrengen starter med `^` betyr det at vi leter fra **starten** av `txt`, så vi vil ikke få en match dersom `txt="exp(−322.7623)"`. Hvis det er en match vil `check` gi nyttig informasjon: `check.end(0)` er posisjonen for første element i teksten som ikke er en match, så `txt[check.end(0):]` er den delen av `txt` som fremdeles må parses. `check.group(0)` er teksten som matcher, så `float(check.group(0))` gir tallverdien "oversatt" fra strengen. Hvis det ikke er en match vil `check` være `None`.

En nyttig shortcut i `re.search` er at vi kan lete etter flere sub-strenger samtidig, dersom vi gir dem inn som en sammensatt streng med "|" mellom delene. Dersom `self.functions` er en dictionary med alle funksjonene programmet ditt støtter som keys, så vil

```
targets = "|".join(["^" + func for func in self.functions.keys()])
check = re.search(targets, txt)
```

lete etter alle funksjonene du har definert på en gang. Ettersom vi ser på match kun fra starten av teksten (legg merke til `"^"`) vil dette funke så lenge vi antar at det ikke er to funksjoner som har navn der navnet til den ene er starten på navnet til den andre. Det betyr at dersom du har definert at `"SQRT"` skal beregne kvadratrot kan du ikke bruke `"SQR"` for å kvadrere. I stedet kan du for eksempel bruke `"SQUARE"`.

Et potensielt problem er at det vil bli vanskelig å skille mellom **operatoren** "−" og **fortegnet** "−". Det er selvsagt mulig å sette opp regler for å parse dette riktig, men en enklere løsning du kan benytte her er å anta at de fire regneartene skrives med tekst. Det betyr at vi vil få en oppgave-tekst som for eksempel `"1 MINUS −2"`, men ikke `"1 − −2"`. Hvis du har brukt `Calculator.__init__` - metoden gitt tidligere vil du se at du har definert regneartene slik der allerede; dette er `keys` i `self.operators` som gir de tekst-elementene vi leter etter. Operasjonene heter dermed `"PLUSS"`, `"MINUS"`, `"GANGE"`, og `"DELE"` nå.

Før du begynner å parse må du fjerne alle mellomrommene fra teksten, og gjøre den om til uppercase. Du gjør dette med `text = text.replace(" ", "").upper()`.

Metoden som implementerer parseren din skal ta inn en tekst-streng, lete gjennom denne, og bygge opp en liste med alle elementene som er i input-teksten. Den skal returnere en liste med enten instanser av riktig type: tall er kodet som `float`, funksjoner som `Function`, regneartene som `Operator`, og parentesene som `str`. Listen som produseres skal være slik at implementasjonen din av shunting-yard algoritmen kan bruke den som input.

Hvis du har lyst til å være veldig fancy kan du lage parseren som en `generator` som `yield`-er elementene enkeltvis i stedet for å bygge opp en full liste. Generatorer er en veldig kraftig del av Python som kanskje ikke brukes så ofte som de burde (men de er strengt tatt ikke nødvendig i denne oppgaven).

Implementasjon – Del 7:
Implementer parseren som beskrevet. Lag en test der du sender in tekster du vet at skal kunne parses og sjekk at elementene som kommer tilbake er av riktig type.

6 Sett sammen alt sammen

Nå har du alle delene du trenger, og alt som gjenstår er å sette dem sammen.

Implementasjon – Del 8:
Lag en hoved-rutine `Calculator.calculate_expression(txt)` som tar en tekst-streng som input, og evaluerer denne. Test systemet ditt med forskjellige input-strenger, som for eksempel `"EXP(1 pluss 2 gange 3)"` og `"((15 DELE (7 MINUS (1 PLUSS 1))) GANGE 3) MINUS (2 PLUSS (1 PLUSS 1))"`.

7 Hva kreves for å bestå oppgaven

For å bestå denne oppgaven må du:

- Løse alle de obligatoriske del-oppgavene.
- Du skal gjøre arbeidet alene, og få det godkjent innen fristen.
- Systemet skal implementeres med objekt-orientert Python.