TDT4113

Oppgave 4: Kalkis

Oversikt over oppgaven

“**EXP(3 PLUSS 3 GANGE 3 PLUSS (1 MINUS 1))**”

--> 162754.79141900392

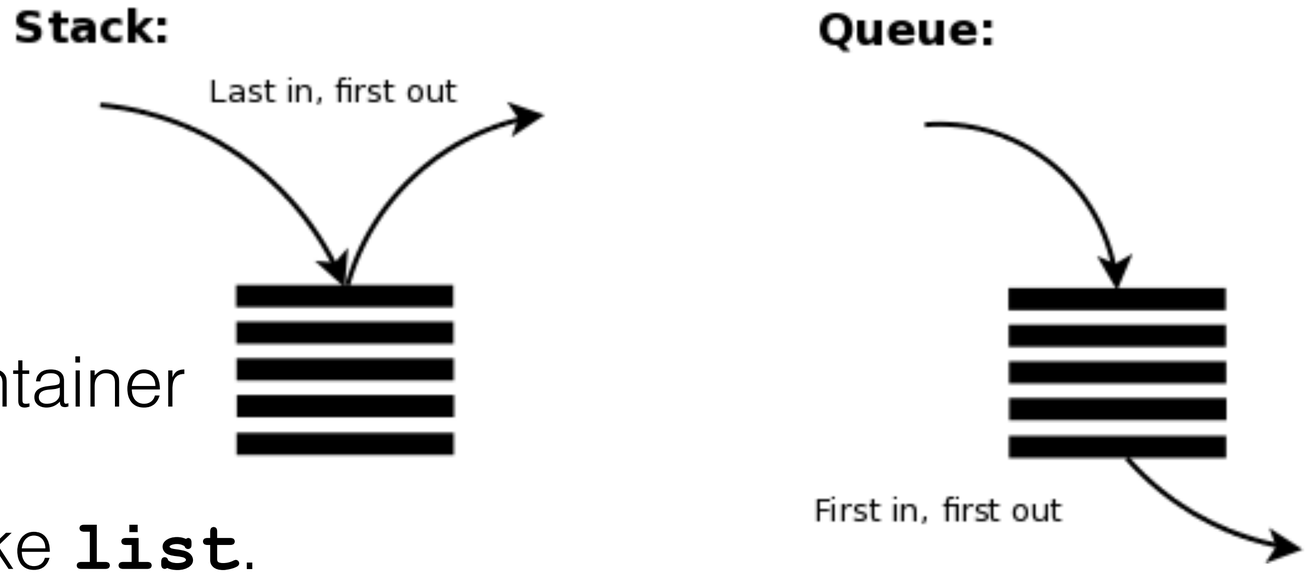
- Implementere en kalkulator som tar “fritekst” som input og regner ut svaret
  - Skal støtte de fire regneartene
  - Skal støtte funksjoner som **exp**, **log**, etc
  - Skal støtte parenteser og nøstede parenteser
- Kan anta at **input er syntaktisk riktig**

Lære-elementer

- Stacker og køer
- Regex
- “Unit-testing”

Kø, stack

- Køer og stacker er “containere” av data-elementer, som støtter **`push, pop`** (og evt. **`peek`**, **`size_of`** og **`is_empty`**).
  - Kø: First in - First Out,
  - Stack: Last in - First out
- Bygges på en generell data-container
  - I Python er det naturlig å bruke **`list`**.



```python
class Container:
    def __init__(self):
        self._items = []

    def is_empty(self):
        return self.size() == 0

    def push(self, item):
        self._items.append(item)

    def pop(self, position):
        assert not self.is_empty()
        return self._items.pop(position)

    def peek(self, position):
        assert not self.is_empty()
        return self._items[position]

    def size(self):
        return len(self._items)

    def clear(self):
        self._items.clear()
```

```python
class Stack(Container):
    def __init__(self):
        super(Stack, self).__init__()

    def pop(self, position=-1):
        return super(Stack, self).pop(position)

    def peek(self, position=-1):
        return super(Stack, self).peek(position)


class Queue(Container):
    def __init__(self):
        super(Queue, self).__init__()

    def peek(self, position=0):
        return super(Queue, self).peek(position)

    def pop(self, position=0):
        return super(Queue, self).pop(position)
```

Funksjoner og operatorer

- Vi trenger en **`Function`**-klasse som wrapper implementasjoner i **`numpy`**.
  - **`numpy`** er en pakke med mye numerisk matematikk.
    - Kjør **`pip install numpy`** for å få fatt på den.
    - **Spør studass om det ikke funker**
  - Eksempler på funksjoner: **`numpy.exp`**, **`numpy.log`**, **`numpy.sin`**, **`numpy.cos`**, **`numpy.sqrt`**.
- Gjør det samme for **`Operator`**
  - Her er gullet **`numpy.add`**, **`numpy.subtract`**, **`numpy.multiply`**, **`numpy.divide`**.

```
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
```

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

Operatorene

- Mye godt på samme vis som funksjonene, men
  - Trenger ***to*** input til sin **`execute`**-metode.
  - Må vite sin egen “styrke” så vi får til operator-presedens

Resten av implementasjonen

1. Parse tekst-input:

   “**`2 PLUSS 2`**” —>**`[2, add, 2]`**

2. Oversette til Reverse Polish Notation (RPN):

   `[2, add, 2]` —> **`[2, 2, add]`**

3. Gjøre beregningen på RPN-representasjonen:

   `[2, 2, add]` —> **`4`**

Vi gjør oppgaven “bakfra” (Først pkt. 3, så 2 og 1).
Det gir oss muligheten til å lage mindre kode-elementer som kan testes fortløpende.

Python og unit-tester

- I oppgaven blir dere bedt om å teste delene av implementasjonen fortløpende.

- Selv om det ikke er påkrevd her er det god praksis å lage unit-tester; det er flere løsninger for hvordan dette kan gjøres.

- Et oppsett som er enkelt i bruk er **`unittest`** (inspirert av JUnit), som evt kan kombineres med f.ex. **`nosetests`**

```
Helges-MacBook-Pro-Mid:Python helgel$ nosetests test_calculator.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK
```

```
1   import unittest
2   import numpy
3   from calculator import Stack, Function, Operator, Calculator
4
5
6   class TestCalculator(unittest.TestCase):
7       def test_part_1(self):...
17
18      def test_part_2(self):...
23
24      def test_part_3(self):...
29
30      def test_part_4(self):...
37
38      def test_part_5(self):
39          calc = Calculator()
40          calc.output_queue.push(1.)
41          calc.output_queue.push(2.)
42          calc.output_queue.push(3.)
43          calc.output_queue.push(calc.operators['GANGE'])
44          calc.output_queue.push(calc.operators['PLUSS'])
45          calc.output_queue.push(calc.functions['EXP'])
46
47          value = calc.evaluate_output_queue()
48          self.assertAlmostEqual(value, numpy.exp(7), 1E-6)
49
```

Evaluere Reverse Polish Notation

- Tar elementene i en input-kø og evaluerer denne. Elementene i køen er **tall**, **operatorer** og **funksjoner**.
  - Her er **`isinstance`** nyttig. Siden vi har definert egne klasser for funksjoner og operatorer kan vi **teste direkte**.
- Bruker en stack internt til **mellomlagring**
- Returnerer en tall-verdi.
- Hvert element behandles separat, etter regler bestemt av sin **type**.

Evaluering av RPN — Detaljene

1. Opprett en tom stack.
2. Gå gjennom alle elementene i en kø:
   A. **Tall**: Push på stack'en
   B. **Funksjon**: Pop et element av stack'en; evaluer func med denne, push svar på stack'en
   C. **Operasjon**: Pop to elementer av stacken, **pass på rekkefølgen,** utfør operasjonen, push svar på stacken
3. Returner det gjenstående elementet på stack'en. Det er **kun ett element** der nå!

Eksempel: Beregner `exp(1 + 2 * 3)`

[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

Lage RPN: Shunting-yard

- “Stokker om” på elementene i input-køen, samtidig som den løser ut alle parenteser.
- Tar en kø av elementer som input
  - Bruker en kø og en stack inne i metoden
  - Stack’en benyttes kun til mellom-lagring.
  - Returnerer køen.
- Hvert element i input behandles separat, etter regler bestemt av sin **type**; lovlige typer er tall, operator, funksjon, start-parentes, slutt-parentes.

Shunting-yard: Hvis elementet er …

- Et **tall**: Push på **køen**.
- En **funksjon**: Push på **stack'en**.
- En **start-parentes**: Push på **stack'en**.
- En **slutt-parentes**: Pop elementer av **stack'en** og push dem på **køen** en etter en til vi finner **start-parentesen**. Pop og kast start-parentesen fra **stack'en**.
- En **operator**: Pop elementer av **stack'en** og push dem på **køen** en etter en så lenge stack'en gir enten en **funksjon** eller en operator som er **minst like sterk** som den vi behandler. Etterpå skal operatoren pushes på **stack'en**.
- **Til slutt:** Når input er tom, pop'er vi elementene som er på **stack'en** over på **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`~~

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~~

```python
class Calculator:
    # Initialization: Binds operators are functions, defines storage for output queue
    def __init__(self, debug_mode=True):
        self.debug_mode = debug_mode

        # Define the operators supported. Link them to a python func (typically from numpy or math)
        self.operators = {'PLUSS': Operator(numpy.add, 0),
                          'GANGE': Operator(numpy.multiply, 1),
                          'DELE': Operator(numpy.divide, 1),
                          'MINUS': Operator(numpy.subtract, 0)}

        # Define the functions supported. Link them to a python func (typically from numpy or math)
        self.functions = {'EXP': Function(numpy.exp),
                          'LOG': Function(numpy.log),
                          'ABS': Function(numpy.abs),
                          'SIN': Function(numpy.sin),
                          'COS': Function(numpy.cos),
                          'SQRT': Function(numpy.sqrt)}

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

    # Generator of tokens from the textual input.
    # Tokens are numbers, funtions, operators, parenthesis
    def __get_next_token(self, txt):...

    # Evaluates the RPN in self.output_queue
    def evaluate_output_queue(self):...

    # Run through input text using __get_next_token, and build up an RPN using
    # the shunting yard algorithm
    def parse_text(self, text):...

    # Kicks off parse_text to fill the output queue with RPN,
    # then evaluates the RPN
    def calculate_expression(self, text):...
```

Parse tekst

- Kalkulatoren tar tekst-input, som vi må parse:
  `"2 GANGE 3 PLUSS 1"` -> **`[2, multiply, 3, add, 1]`**

- Gjør dette ved hjelp av regex (**`import re`**)

  - Vi starter først i input-strengen, finner ut hva det er, oppretter et objekt av riktig type (**`Number`**, **`Function`**, **`Operator`**, **`str`**) med riktig verdi, og legger dette i en kø.

  - Flytt fram i strengen, og fortsett til strengen er tom

Litt om **`re.search`**

- **`check = re.search(pattern, txt)`** returnerer et såkalt “match-object”.
- Et match-object har mange interessante metoder/attributter (sjekk dokumentasjonen). Av spesiell interesse for oss er flg.:
  - **`check == None`** betyr at det ikke var noe treff
  - **`check.start(0)`** gir start-posisjonen i **`txt`** der første match er. Dersom **`pattern`** alltid starter med “**`^`**” vil denne alltid være **`0`**.
  - **`check.group(0)`** gir delen av **`txt`** som matcher **`pattern`** første gang.
  - **`check.end(0)`** gir slutt-posisjonen i **`txt`** for første match. Vi kan dermed bruke **`txt = txt[check.end(0):]`** for å “gå videre” i **`txt`** etter å ha prosessert matchen.

Mer regex-snadder

- **`check = re.search("^[-0123456789.]+", txt)`** sjekker om **`txt`** **starter** med en **sekvens** av symboler “-“, “.”, “0”, “1”, …, “9”.
  - **Merk at vi “bruker opp” symbolet “-” til som fortegn.** Vi vil kalle operasjonene ved navnene “PLUSS”, “MINUS”, “GANGE” og “DELE.

- **`check = re.search("^\\(", txt)`**: Starter **`txt`** med “(“ ? **NB!!!** IKKE “**+**” her, fordi vi vil ha dem enkeltvis og vite det spesielt om det er flere — fordi da skal de behandles separat.

- Anta at vi har en dictionary med mulige keys gitt ved operatorene vi kjenner igjen, og value den instansen fra **`Operator`**-klassen vi binder det mot. Da vil

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

  sjekker om **`txt`** starter med noen av disse. Merk at “**`.join`**”-statemente genererer noe a la **`“^PLUSS|^GANGE|^DELE|^MINUS”`**.

- For enkelthets skyld bør du sette input-teksten i uppercase og fjerne alle space:

```
txt = txt.replace(" ", "").upper()
```

Demo

- Oppgaven har **2 ukers frist**.
- Demonstrasjonen vil typisk være at studeass gir en streng av typen
  “**`EXP(3 PLUSS 3 GANGE 3 PLUSS (1 MINUS 1))`**”
  som du må evaluere.
- Det er mange “moving parts” her — lurt å lage gode unit-tester, og også noen interne run-time sjekker:
  - Er input til **`execute`** av funksjoner og operatorer **`numbers.Number`**?
  - Er det kun ett element på output-stacken når RPN evalueringen er ferdig?
  - Er det deler av input-strengen du ikke klarer å parse? (Jeg har tabbet meg ut mange ganger med å gi inn f.ex. “**`1 + 1`**” i stedet for “**`1 PLUSS 1`**”…)