resitev

January 28, 2024

1 Day 18: Operation Order

(povezava na nalogo)

V nalogi blago pokukamo k prevajalnikom. Samo "blago" pa zato, ker v okviru takšne naloge ne moremo implementirati ničesar resnega. Niti za to nalogo ni potrebno.

1.1 Prvi del

Imamo izraz, ki vsebuje seštevanja in množenja. Izračunati pa ga moramo od leve proti desni, ne da bi imelo množenje prednost pred seštevanjem.

```
1 + 2 * 3 + 4 * 5 + 6
  3   * 3 + 4 * 5 + 6
      9   + 4 * 5 + 6
         13   * 5 + 6
             65   + 6
                 71
```

Izraz lahko vsebuje tudi oklepaje; te pa upoštevamo.

Izraz, s katerim se bomo igrali je,

[1]:
```
s = "1 + (21 * 3) + (41 * (15 + 6))"
```

V sami nalogi pa dobimo datoteko s takšnimi izrazi in naloga je izračunati vsoto rezultatov.

Ta, prvi del bomo rešili brez trikov, z močjo lastnih mišic.

1.1.1 Tokenizacija

Najprej bomo izraz *tokenizirali*. Token je osnovni element v leksikalni analizi, prvem koraku, ki se zgodi ob prevajanju ali interpretiranju programa. Token je osnovni element, torej ločilo (vejice, oklepaji), operator (tudi, na primer, `+=`), število… V našem primeru so to števila, `+`, `*`, `(` in `)`. Izraz razbijemo na takšne stvari. Ročno, ne z regularnimi izrazi. Za vajo.

[2]:
```
def tokenize(s):
    acc = ""
    for c in s:
        c = c.strip()
        if not c:
```

```
            continue
        if not acc or (c.isdigit() and acc.isdigit()):
            acc += c
        else:
            yield acc
            acc = c
    if acc:
        yield acc

list(tokenize(s))
```

[2]:
```
['1',
 '+',
 '(',
 '21',
 '*',
 '3',
 ')',
 '+',
 '(',
 '41',
 '*',
 '(',
 '15',
 '+',
 '6',
 ')',
 ')']
```

Generator deluje tako, da gre po nizu. In v `acc` nabira, kar bo generiral. Generira pa vedno reči, ki so se nabrale pred trenutnim znakom.

- Beli prostor preskoči (`c = c.strip(); if not c: continue`). V `acc` je vedno, "kar se je nabralo".
- Nato preveri, ali se doslej ni nabralo nič, ali pa je trenutni znak števka in ima tudi v zbirki števke. Če je tako, potem sestavljamo število, torej vanj dodamo še tega. (Ali pa nismo nabrali še ničesar in tudi `c` ni število. Tak `c` bi lahko že sedaj izgenerirali, vendar bo bolj elegantno, če ga ne.)
- V nasprotnem primeru pa se je nabralo nekaj, k čemur ne smemo dodati trenutnega znaka. Torej to reč izgeneriramo in potem zapišemo, da se je "nabral" trenutni znak.

Ko je niza konec, preverimo, ali je potrebno še kaj izgenerirati.

Taisto reč je mogoče napisati še na veliko načinov. Ta je eden tistih, ki delujejo. Mogoče ni eden tistih, ki so teoretično najbolj pravilni. Je pa kar praktičen, pregleden in kratek.

1.1.2 Izračun

Zdaj nam preostane iti prek tokenov in računati. Ker množenje nima prednosti pred seštevanjem, oklepaji pa so preprosta zadeva, to ne bo težko.

Funkcija `evaluate` bo prejme generator tokenov in ga izračunala.

[3]:
```
def evaluate(s):
    res = next(s)
    if res == "(":
        res = evaluate(s)
    else:
        res = int(res)

    for op in s:
        if op == ")":
            return res

        num = next(s)
        if num == "(":
            num = evaluate(s)
        else:
            num = int(num)

        if op == "+":
            res += num
        else:
            res *= num
    return res

evaluate(tokenize(s))
```

[3]:
```
925
```

(Trenutni) rezultat beleži v `res`. Najprej ga inicializira tako, da prebere prvi token, z `next(s)`. Če je prvi token `(`, pokliče `evaluate` (da, samo sebe), da evaluira izraz v oklepaju. Sicer pa prvi token predstavlja število (ker pač ne more biti zaklepaj ali operator), torej ga spremeni iz niza v število.

Zdaj pa pride malo grdobije: generator `s` bomo brali na dveh mestih: z zanko `for` in z `next` znotraj te zanke. Z zanko bomo brali operatorje in zaklepaje.

- Če naletimo na zaklepaj, je izraza konec in vrnemo trenutno naračunano vrednost.
- Če nimamo oklepaja, pa je `op` bodisi `+` bodisi `*`, torej mu sledi še neko število ali pa zaklepaj. Preberemo torej naslednjo reč `num = next(s)`. Če gre za oklepaj, pokličemo `num = evaluate(s)`, da izračunamo izraz znotraj oklepajev. Sicer pa gre za število, torej ga pretvorimo v `int`. V vsakem primeru to število prištejemo ali primnožimo.

1.1.3 Končna vsota

Da rešimo nalogo, moramo le sešteti izračunane tokenizirane vrstice datoteke.

[4]:
```
print(sum(evaluate(tokenize(v)) for v in open("input.txt")))
```

```
13976444272545
```

1.2 Drugi del: s Pythonovim parserjem

V drugem delu naloge pa je potrebno izraze izračunati, kot da ima seštevanje prednost pred množenjem. To je bolj zapleteno, ker izrazov ne moremo več računati z leve proti desni.

Tu lenoba premaga željo po preskušanju lastnih mišic. Nalogo bomo rešili na dva načina, oba poučna. Prvič bomo izrabili Pythonov parser, drugič bomo videli posebne metode razredov.

1.2.1 Pythonov parser

Ko Python ovrednosti naš izraz - ali program - mora početi nekaj podobnega, kot smo počeli zgoraj, le veliko bolj zapleteno. Celoten izraz ali program pretvori v sintaktično drevo.

Kako to počne, nam je voljan pokazati s pomočjo modula `ast`.

[5]:
```
import ast

tree = ast.parse("2 * (3 + 4)", mode="eval")
ast.dump(tree)
```

[5]:
```
'Expression(body=BinOp(left=Num(n=2), op=Mult(), right=BinOp(left=Num(n=3),
op=Add(), right=Num(n=4))))'
```

Tole pove, da imamo dvojiški operator, in sicer `Mult`. Na levi je število, na desni pa je dvojiški operator `Add`, ki ima na levi število in na desni tudi.

[6]:
```
tree = ast.parse("2 * f(7 + 3)", mode="eval")
ast.dump(tree)
```

[6]:
```
"Expression(body=BinOp(left=Num(n=2), op=Mult(), right=Call(func=Name(id='f',
ctx=Load()), args=[BinOp(left=Num(n=7), op=Add(), right=Num(n=3))],
keywords=[])))"
```

Tule imamo dvojiški operator, ki ima na levi število, na desni pa klic. Argument tega klica je binarni operator … in tako naprej.

Funkcija `f` ne obstaja, vendar to Pythona v tem trenutku nič ne zanima, saj zgolj analizira program. Pač pa bo tu že opazil sintaktične napake, saj sintaktično nepravilnega programa ali izraza ne more zložiti v drevo.

Tule je še primer analize programa.

[7]:
```
tree = ast.parse("while i < 3: i += 1")
ast.dump(tree)
```

[7]:
```
"Module(body=[While(test=Compare(left=Name(id='i', ctx=Load()), ops=[Lt()],
comparators=[Num(n=3)]), body=[AugAssign(target=Name(id='i', ctx=Store()),
```

```
op=Add(), value=Num(n=1))], orelse=[])])"
```

Imamo `while`, ki vsebuje test, sestavljen iz primerjanja, operator je `Lt`, levo je ime `i`, … in tako naprej.

V testih za domače naloge ste morda opazili delček tega, ko sem testiral, ali vaše funkcije z izpeljanimi seznami/množicami/slovarji v resnici vsebujejo le `return`. Tam so testi s pomočjo `ast.parse` poškilili v kodo vaše funkcije.

1.2.2 Drevo izrazov iz naloge

Pythonov parser se že zaveda prioritete operatorjev, saj mora sestaviti drevo, ki se ga da izračunati, tako da mora biti množenje v drevesu nižje od seštevanja.

[8]:
```
tree = ast.parse("2 + 3 * 4")
ast.dump(tree)
```

[8]:
```
'Module(body=[Expr(value=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))))])'
```

Ko Python računa vrednost tega drevesa, mora najprej izračunati v vrednost v listu (`Mult`, levo je 3 in desno 4) in rezultat potem prišteti k vrednosti iz leve veje.

Če zamenjamo vrstni red členov, je seveda isto, le veji se zamenjata.

[9]:
```
tree = ast.parse("3 * 4 + 2")
ast.dump(tree)
```

[9]:
```
'Module(body=[Expr(value=BinOp(left=BinOp(left=Num(n=3), op=Mult(),
right=Num(n=4)), op=Add(), right=Num(n=2)))])'
```

Če torej podtaknemo Pythonu izraz iz naloge, bo sestavil pravilno drevo. Nahecamo ga tako, da seštevanje zamenjamo s potenciranjem.

[10]:
```
def parse(s):
    return ast.parse(s.replace("+", "**"), mode="eval").body

tree = parse("3 * 4 + 2")
ast.dump(tree)
```

[10]:
```
'BinOp(left=Num(n=3), op=Mult(), right=BinOp(left=Num(n=4), op=Pow(),
right=Num(n=2)))'
```

Zdaj je, vidimo, množenju v korenu, ker je potrebno prej potencirati. :)

Da se bo to pravilno izračunalo, pa moramo napisati svojo funkcijo, ki ovrednosti takšno drevo. Preprosta bo.

1.2.3 Izračun drevesa

Če imamo število (`Num`), ga, preprosto, vrnemo. Sicer pa imamo binarno operacijo. Ovrednotimo levo in desno vejo, nato pa vrnemo produkt, če gre za produkt. Če ne gre za produkt, pa se delamo neumne in vrnemo vsoto.

[11]:
```
def evaluate_tree(node):
    if isinstance(node, ast.Num):
        return node.n

    left, right = evaluate_tree(node.left), evaluate_tree(node.right)
    if isinstance(node.op, ast.Mult):
        return left * right
    else:
        return left + right
```

Da dobimo rešitev naloge, le spustimo vsoto čez poračunana drevesa.

[12]:
```
print(sum(evaluate_tree(parse(v)) for v in open("input.txt")))
```

88500956630893

1.3 Drugi del: s posebnimi metodami

Tu bomo uporabili podoben trik, seštevanje bomo zamenjali s potenciranjem, le da bomo tudi računanje prepustili Pythonu. Za to pa bomo malo drugače definirali potenciranje.

1.3.1 Redefinirano potenciranje

Izmislili si bomo novo vrsto `int`-ov, pri katerih je potenciranje pravzaprav isto kot seštevanje.

[13]:
```
class N(int):
    def __pow__(self, other):
        return N(super().__add__(other))

    def __mul__(self, other):
        return N(super().__mul__(other))
```

[14]:
```
a = N(2)
b = N(3)
```

[15]:
```
a * b
```

[15]: 6

[16]:
```
a ** b
```

[16]: 5

[17]:
```
a ** b * a
```

[17]:
```
10
```

Iz razreda `int` smo torej izpeljali nov razred `N` in mu zamenjali metodi `__pow__` in `__mul__`, ki predstavljata potenciranje in množenje. Obe vrneta nov objekt razreda `N`; `__pow__` vrne to, kar bi vrnil podedovani `__add__`, `__mul__` pa, kar bi vrnil podedovani `__mul__` (le da rezultat spremenimo v `N`!).

Več o tem, kar počnemo, je v dokumentaciji.

1.3.2 Popravek izrazov

Zdaj pa je potrebno spremeniti vsa števila v izrazu iz `int`-ov v `N`-je. Tu si bomo pomagali z regularnimi izrazi (lahko pa bi si tudi s tokenizacijo iz prvega dela naloge, če bi hoteli vsaj malo sloneti na lastnih mišicah).

Izraz `\d+` (ena ali več zaporednih števk) želimo zamenjati z `N(...)`, kjer je v oklepajih taisti izraz.

Poleg tega moramo, seveda, zamenjati `+` z `**`. To storimo tako.

Iz razreda `int` smo torej izpeljali nov razred `N` in mu zamenjali metodi `__pow__` in `__mul__`, ki predstavljata potenciranje in množenje. Obe vrneta nov objekt razreda `N`; `__pow__` vrne to, kar bi vrnil podedovani `__add__`, `__mul__` pa, kar bi vrnil podedovani `__mul__` (le da rezultat spremenimo v `N`!).

Več o tem, kar počnemo, je v dokumentaciji.

1.3.3 Popravek izrazov

Zdaj pa je potrebno spremeniti vsa števila v izrazu iz `int`-ov v `N`-je. Tu si bomo pomagali z regularnimi izrazi (lahko pa bi si tudi s tokenizacijo iz prvega dela naloge, če bi hoteli vsaj malo sloneti na lastnih mišicah).

Izraz `\d+` (ena ali več zaporednih števk) želimo zamenjati z `N(...)`, kjer je v oklepajih taisti izraz.

```
s = re.sub(r"\d+", lambda mo: f"N({mo.group()})", s)
```

Funkcija `re.sub` prejme regularni izraz in niz, s katerim ga je potrebno zamenjati, ali pa funkcijo, ki prejme `MatchObject`. Le-ta ima metodo `group()`, ki, če ji ne podamo argumentov, vrne, del niza, ki se je ujemal z vzorcem.

Poleg tega moramo, seveda, zamenjati `+` z `**`. To storimo tako.

1.3.4 Popravek in izračun

Funkcija, ki popravi izraze in jih izračuna, je takšna.

[18]:
```
import re

def evaluate_pow(s):
    s = re.sub(r"\d+", lambda mo: f"N({mo.group()})", s)
    s = s.replace("+", "**")
```

```
    return eval(s)
```

Popravek smo razložili zgoraj, `eval` pa je funkcija, ki prejme niz in ga izračuna.

Rešitev naloge je potem:

[19]:
```
print(sum(map(evaluate_pow, open("input.txt"))))
```

```
88500956630893
```