

Day 2: Rock, Paper, Scissors

(izvirna naloga)

V datoteki so navodila za poteze v igri papir, škarje, kamen. Nasprotnikove poteze so A=kamen, B=papir, C=škarje, naše pa X=kamen, Y=papir, Z=škarje. Vsaka vrstica datoteke je ena poteza, npr. "A Z" ali "C Y".

V vsaki igri dobimo točke glede na to, kaj smo pokazali in kakšen je bil rezultat. Če pokažemo kamen, avtomatsko dobimo eno točko, če papir dve, če škarje 3. Poleg tega pa dobimo 0 točk za poraz, 3 za izenačenje, 6 za zmago.

Sestavljanje slovarja

Lahko si sestavimo slovar, v katerem piše, koliko točk je vredno kaj. Če si vsi ključni slovarja takšne oblike kot veljavna imena v Pythonu, nam ni potrebno pisati

```
{"X": 1, "Y": 2, "Z": 3}
```

```
{'X': 1, 'Y': 2, 'Z': 3}
```

temveč si lahko olajšamo tipkanje tako, da pokličemo `dict` z argumenti, katerih imena so ključni, vrednosti pa pač vrednosti.

```
dict(X=1, Y=2, Z=3)
```

```
{'X': 1, 'Y': 2, 'Z': 3}
```

V drugi slovar bi lahko zapisali, koliko točk dobimo zato, ker smo z določeno kombinacijo zmagali oz. zgubili oz. nič od tega.

```
dict(AX=3, AY=6, AZ=0, BX=0, BY=3, BZ=6, CX=6, CY=0, CZ=3)
```

```
{'AX': 3,  
 'AY': 6,  
 'AZ': 0,  
 'BX': 0,  
 'BY': 3,  
 'BZ': 6,  
 'CX': 6,  
 'CY': 0,  
 'CZ': 3}
```

Še preprosteje bo, če naredimo kar skupni slovar, ki bo vključeval točke za posamično kombinacijo. Če igramo AX, dobimo 3 točke za izenačenje in še eno, ker smo pokazali X. Torej v igri AX dobimo, skupno, štiri točke.

Imamo torej:

```
points = dict(AX=4, AY=8, AZ=3, BX=1, BY=5, BZ=9, CX=7, CY=2, CZ=6)
```

Preprosta rešitev

Gremo čez datoteko, vsako vrstico spremenimo v ključ (odstranimo presledek med znakoma in `\n` na koncu vrstice) ter k vsoti prištejemo, kolikor dobimo za to kombinacijo.

```
vsota = 0
for vrstica in open("input.txt"):
    vsota += points[vrstica.strip().replace(" ", "")]
print(vsota)

14264
```

Rešitev z generatorji

Po izkušnjah iz prejšnjih let bo prvih nekaj nalog takšnih, da jih bomo zlahka odpravili z eno samo vrstico.

Kaj je potrebno narediti z gornjih programom, je očitno.

```
print(sum(points[line.strip().replace(" ", "")] for line in open("input.txt")))

14264
```

Drugi del

X, Y in Z ne predstavljajo več naše poteze temveč navodilo, ali moramo zmagati ali izgubiti. Isti šmorn, le število točk je drugačno.

Spremenimo slovar v

```
points = dict(AX=3, AY=4, AZ=8, BX=1, BY=5, BZ=9, CX=2, CY=6, CZ=7)
```

in ponovno poženemo isti program kot za prvi del.

```
print(sum(points[line.strip().replace(" ", "")] for line in open("input.txt")))

12382
```

Oba dela

Če hočemo rešiti oba dela v eni vrstici, naredimo zanko čez dva slovarja

```
print([
    sum(points[line.strip().replace(" ", "")] for line in open("input.txt"))
    for points in (dict(AX=4, AY=8, AZ=3, BX=1, BY=5, BZ=9, CX=7, CY=2, CZ=6),
                    dict(AX=3, AY=4, AZ=8, BX=1, BY=5, BZ=9, CX=2, CY=6, CZ=7))
])

[14264, 12382]
```

Kotlin

Čeprav se da v Kotlinu programirati tudi "normalno" (no, tako, iterativno), je bolj zabavno delati s funkcijskim pridihom. Rešitev obeh delov te naloge je takšna.

```
import java.io.File

listOf(
    mapOf("AX" to 4, "AY" to 8, "AZ" to 3, "BX" to 1, "BY" to 5, "BZ" to 9, "CX" to 7, "CY" to 2, "CZ" to 6),
    mapOf("AX" to 3, "AY" to 4, "AZ" to 8, "BX" to 1, "BY" to 5, "BZ" to 9, "CX" to 2, "CY" to 7, "CZ" to 6)
).forEach { points ->
    println (
        File("input.txt")
        .readLines()
        .map { points[it.trim().replace(" ", "")]!! }
        .sum()
    )
}
```

Z `listOf` sestavimo seznam (Kotlin tega žal ne zna z `[in]`, kot Python...). Seznam bo vseboval dva slovarja, ki ju sestavimo z `mapOf`. Elemente opišemo s konstruktom `<key> to <value>`.

Potem pa za vsak element (**forEach**) tega seznama naredimo tole. (Tehnično "tole" je lambda funkcija, ki jo podamo funkciji **forEach** kot argument.) Element seznama bomo poimenovali **points** - tako kot smo ga v Pythonu. Izpisali (**println**) bomo tole: odpremo datoteko, preberemo njene vrstice, z vsako vrstico naredimo točno isto kot v Pythonu (**trim** je Kotlinov ekvivalent Pythonovega **strip**) in potem vse skupaj seštejemo.

Kotlin se, tako kot v prvi nalogi, spet bere bolj v pravo smer kot Python, čeprav je tu tudi Pythonova koda kar berljiva.

Kot zanimivost opozorimo na **!!**. Imamo **points**, katerega vrednosti so števila - v Kotlinu je to **Int**. Vendar **points[x]** ne vrne **Int** temveč **Int?**. To pomeni, "morda bo **Int**, morda pa bo **null**, ker **points** morda nima tega ključa. Rezultat **.map { points[it.trim().replace(" ", "")] }** zato ne bi bil seznam **Int**-ov, temveč seznam **Int?**-ov. Funkcija **sum()** pa zna sešteti **Int**-e, če ji poskušamo podtakniti **Int?**-e, pa bo Kotlin javil napako že med prevajanjem. Tako napisan program se torej sploh ne bo prevedel in zagnal!

Kotlin nas na ta način prisili, da poskrbimo tudi za primer, ko ključ ne bi obstajal. To lahko storimo na različne načine; kdor bi rad izvedel kaj več, naj si v Kotlinovi dokumentaciji prebere stran o null safety. Tule pa povejmo le, kako smo to storili v tem primeru: ker vemo, da bo ključ gotovo obstajal, smo dodali `!!`. Ta spremeni `Int?` v `Int` (in Karkoliže? v Karkoliže). Če smo se zmotili in se bo zgodilo, da ključa vseeno ne bo, pa smo si krivi sami. Kotlin nas je opozoril.