Programozói dokumentáció
Neo

matrix.py:

class

Létrehoztam a Mátrix osztályt, ami minden mátrixhoz egy 2D listát rendel, illetve a lista (mátrix) sorai -és oszlopainak számát is tartalmazza. Amennyiben nincs megadva sor vagy oszlop, akkor ezt automatikusan kiszámolja de 0-nál többnek kell lenniük. A sor -és oszlopszámot kompatibilitás ellenőrzése, vagy ciklus lépésszámának megadására használom majd a program további részében.
Azért írtam minden függvényt a Mátrix osztályba, mert egyrészt tetszik az objektumorientáltság, másrészt ezzel is jelzem, hogy ezek a függvények idegen környezetbe nem feltétlenül implementálhatóak, túl szorosan kapcsolódnak a Mátrix osztályhoz.

__str__:

Lehetővé teszi a mátrixok egyszerű kiírását. Az abs_max() és float_e() függvények meghívásával képes saját magát rendezni, így biztosítva az elegáns megjelenést. Működését tekintve, egy üres sztringhez fűzi hozzá a mátrix egyes elemeit, majd az aktuális lista végén új sort kezd. Kiszűri ha egy elem (-0) és lecseréli 0-ra. Nem meglepő módon sztring a visszatérési értéke.

abs_max:

Megtalálja egy Mátrix abszolút maximumát. Azért van rá szükség, hogy a legtöbb számjegyből álló elemhez lehessen igazítani a kiírást. Ha ez a szám negatív akkor megszorozza 10-el, az előjel hossza miatt.
Csak az __str__ függvényhez használja.

float_e:

Ez is az __str__-hez kell. Ha van benne float akkor nagyobb közökkel kell kiírni a számokat.

__mul__:

Mátrix és szám szorzatának kiszámolására alkalmas, mely nem más, mint az összes elem szorzata egy skalárral.
Ciklussal kiválasztja a 0. sort, majd másik ciklussal elemenként végigmegy rajta és megszorozva azokat berakja egy átmeneti sor változóba. Mikor a külső ciklus lefut- azaz elértünk a mátrix következő soráig- az átmeneti sort berakja a visszatérési listába.
Mátrix a visszatérési értéke, melynek sorai -és oszlopainak száma megegyezik az eredetivel.

__add__:

Mivel két mátrix összeadása csak akkor lehetséges, ha mindkettő $n \times m$-es, így elég csak az egyik mátrix dimenzióival dolgozni. Működése megegyezik a szorzáséval, csak ebben az esetben mx1 és mx2 azonos indexű elemei összeadódnak.
Mátrix a visszatérési értéke, melynek sorai -és oszlopainak száma megegyezik az eredetivel.

__sub__:

Egy mínuszjelet kivéve megegyezik az összeadás függvénnyel. Csak azért kell mert így a program további része érthetőbben és rendezettebben tud működni. Persze nem rossz funkció ezért a felhasználói felületről is elérhető.

determináns:

Kifejtési tétellel minden $n \times n$- es mátrix determinánsa kiszámolható rekurzívan, vagyis vissza lehet vezetni n db $(n - 1) \times (n - 1)$ -es almátrix determinánsára. Ezt addig kell ismételni amíg el nem jutunk 2×2-es mátrixokig melyeknek már 1 sorban ki lehet számolni a determinánsát- ez található a base case -ben.
A rekurziót sor indexre írtam, tehát mindig a sor indexre hívja meg az almatrix() és sakktabla() függvényeket.
Az almátrixok determinánsa a Mátrixra és sor_index-re meghívott determináns függvény.
A determináns a (sakktábla szabály alapján megadott -1 vagy 1-es szorzó), (a mátrix sor_idex-edik sora, 0 index-edik oszlopa által kijelölt elem), és az (almátrix determinánsának) szorzatainak összege.
Visszatérési értéke egy szám.

almatrix:

Az almátrix a (kapott mátrix összes sora), kivéve a sor_index-edik, (összes eleme), kivéve az oszlop_index-edik. Visszatérési értéke egy Mátrix almátrix, aminek mindkét dimenziója egyel csökkent. Ezért is működik a rekurzió.
Inverzszámoláshoz kell sor -és oszlop indexre is működnie, determinánshoz elég lenne csak az egyik.

sakktabla:

Itt is az Inverz függvény miatt kell két index, de a determináns csak a sort használja, az oszlop ezért alapból 0. Visszatérési értéke (+1), ha a sor -és oszlop indexek párosok, (-1) ha valamelyik páratlan. Azért nem fordítva, mert az első sor igazából a 0 a mátrix listában .

transzponalt:

Egy mátrix transzponáltja nem más, mint annak összes oszlopa sorrá alakítva. Így egy $n \times m$-es mátrixból $m \times n$- es lesz.
Az algoritmus veszi a 0. oszlopot, annak összes elemét belerakja egy átmeneti sor (temp_sor) változóba, amikor vége az oszlopnak, a temp_sor változót beteszi a visszatérési listába (transzp_mx). A végén már csak a

sort -és oszlopszámot kell megcserélni.
Visszatérései értéke egy Mátrix.

mx_mul_mx:

Két mátrixot akkor lehet egymással összeszorozni, ha egyik n×k másik k×m méretű. Ezt a függvényt először megírtam nagyon bonyolultan, majd rájöttem hogy lebontva kisebb problémára a nagyot sokkal egyszerűbb lesz. Így jött létre egy skalarszorzat() függvény is, és transzponáltam az egyik mátrixot. Ennek eredményeképp a skalárszorzat a transzponált mátrix és a másik mátrix k hosszú sorait kapja és 1 számot ad vissza. Mivel a végeredmény egy n×m-es mátrix, a skalárszorzatot n×m-szer fut le.

skalarszorzat:

Két 1D listán -vektoron- megy át ciklussal, és szorozza össze az azonos indexű elemeiket. Visszatérési értéke a skalárszorzat, ami a szorzatok összege.
Nem adom át neki a self-et, ezért kezdtem használni a staticmethod- ot.

inverz:

Nem tudom a metódus hivatalos nevét, de lényeg, hogy egy n×n-es mátrix összes indexéről képzett almátrixainak a determinánsaiból kell készíteni egy új mátrixot, majd ennek minden elemét el kell osztani az eredeti mátrix determinánsával, és az így kapott mátrix lesz az inverz. Az osztásra felhasználtam a __mul__ függvényt és a determináns reciprokával szoroztam. Az almátrixok determinánsaiból képzett mátrixnál még arra kell figyelni, hogy ezeket a sakktábla szabály alapján kell előjelezni.

gauss_lmnc:

Gauss eliminálni csak n×n+1- es bővített mátrixon lehet.
A függvény első felében három ciklus van: egy külső és két belső amelyikből 1 külsőre vagy az egyik, vagy a másik belső fog lefutni. A külső annyiszor lép, ahány változós az egyenletrendszer- 1 (azért, mert az n. oszlopban lévő változó az elimináció miatt már eleve kijön), tehát ahány oszlopa van -2× fut le. A belső ciklusoknál el kell dönteni, hogy nulla-e az oszlop oszlopadik eleme (azért [oszlop] [oszlop], mert az n×n-es mátrix átlóján kell haladni tehát ez lesz a főátló). Ha ez nem nulla, akkor el kell osztani a sort az elem[oszlop] [oszlop]-al, hogy a főátlóban majd csupa egyesek legyenek, és ki kell vonni az összes alatta lévő sorból úgy, hogy az alatta lévő sorok aktuális oszlopában lévő elemének a többszörösével be kell szorozni: ettől lesz felsőháromszög mátrix. Ha az [oszlop] [oszlop] == nulla, akkor nem lehet vele osztani ezért áttérünk a másik belső ciklusra, ahol olyan sort kell keresni, aminek a sor [oszlop]-adik eleme nem nulla, ezzel ki kell cserélni. Ha ilyen nincs akkor is csak lefutott a belső ciklus és nem okoz különösebb gondot mivel abban az oszlopban az aktuális sor alatt már az összes elem nulla, vagy jöhet a következő oszlop.
Amikor minden lefutott kapunk egy felsőháromszög mátrixot, aminek a főátló elemei mind 1-esek vagy nullások kivéve az [n][n] elemet mert az nem feltétlenül == 1, ezért le kell osztani (sor_1_1 függvénnyel).
A függvény második fele lényegében ugyanez, csak alulról felfele halad, és a főátlók elemei már mind egyesek (vagy nullák). Ez alsóháromszög

mátrixot csinál, de egységmátrixot kapunk, mert felsőháromszög mátrixból indult.

sor_1_1:

Ez osztja le az egyes sorokat a Gauss eliminálás során úgy, hogy a főátlóbeli elemük 1 legyen.

masol:

Másolatot készít egy mátrixról, arra az esetre, ha nem referenciával akarnánk dolgozni.
Például a Gauss eliminálásnál folyamatosan a másolt mátrixot módosítja a program.

rang:

A rang egy $n \times n+1$ -es mátrixon kezdetben egyenlő a sorok számával, majd Gauss-elimináció után minden főátlóbeli nullás érték 1-gyel csökkenti az értékékt.

main.py:

class Tarhely:

Ezt azért találtam ki, mert szerintem sokkal kényelmesebb úgy használni a programot, ha van egy tárhely, ami módosítható, és amikor műveletet szeretnénk elvégezni csak a tárhelyet kell kiválasztani. Azért van három, hogy két különböző mátrixon lehessen kétmátrixos műveleteket elvégezni, és az eredmény felülírás nélkül menthető legyen.

main:

Neo, menüvezérelt program lévén kénytelen egy végtelenciklusban futni. A kezelés átláthatósága végett almenüket alakítottam ki, így a main() is maximálisan rövid lett. Négy úton lehet belőle kijutni, ebből három almenü, egy kilépés.

import_menu:

A mátrixbevitel két opciója közül lehet választani.

mx_fajlbol:

Fájlból való beolvasáshoz hívja meg a Matrix class-ból a beolvas() függvényt.
Itt kerül elő először a főprogram egyik fontos függvénye a mentés.

mentes:

Csaknem minden mátrixművelet elvégzése után elmentheti a felhasználó a kapott eredményt, ezért nagyon sokszor fordul elő ez a függvény a kódban. Nem csinál semmi különöset, csak feltesz egy kérdést hova szeretné menteni a felhasználó a mátrixot, és a válasz alapján felülírja A/ B/ C tárhelyek valamelyikét.

user_input:

Lehetővé teszi hogy a felhasználó- a sorok és oszlopok számának megadása után- feltöltse a mátrixot értékekkel, utána elmentse a kívánt tárhelyre.

muveletek:

Ez a következő menü, jóval több opcióval, mint az import.
Ebben az elosztóközpontban lehet kiválasztani Neo fő büszkeségeinek (mátrixműveletek) alkalmazásait.

gauss_elim -és többi mátrixművelet:

Természetesen az összes fő művelethez a Matrix class-ban itt is tartozik egy függvény, ami meghívja, hogy alkalmazni lehessen őket, ezért ezek elég unalmasak lesznek. Viszont nem a gauss_elim() mivel itt találkozunk először a tarhely_valaszt() és mented_e() függvényekkel melyek fontos építőelemei a felhasználói-felület-legónak.
Nem írok le minden műveletet egyesével mert egy sémára épülnek. Megkérdezik hogy a Tarhely melyik mátrixán szeretné a felhasználó elvégezni az adott műveletet, elvégzi -és kiírja a műveletet, majd rákérdez hogy szeretné-e menteni, és ha igen akkor hova.

tarhely_valaszt:

Minden művelet előtt felcsendül az a kérdés, hogy melyik Tárhely mátrixán

(A/B/C) fusson le a választott függvény. Természetesen ezt sem írtam le minden művelethez, csak meghíják a tarhely_valaszt()-ot, ami bekér egy értéket (lehetőleg A/B/C) és visszaküldi az ott található Matrixot, illetve a választott mátrix nevét. Ez szerintem csak a név miatt sikerült érdekesre, mivel így érhető el az, hogy nem „adott mátrix” kifejezések szerepelnek a műveletek közben, hanem a nevük jelenik meg.

mented_e:

E kérdés megválaszolására is érdemes volt írni egy függvényt, amely he igenleges választ kap, meghívja a már korábban taglalt mentes() függvényt.

kiir_menu:

A Főmenüből elérhető harmadik almenü. Itt lehet fájlba mentés, vagy képernyőre kiírás között dilemmázni.

kepernyo_kiir:

A Matrix class-ban megírt __str__ függvény jócskán leegyszerűsíti ezt a feladatot, ennek köszönhetően itt tényleg csal mátrixot kell választani és már ki is lehet írni.

fajl_kiir:

Meghívja a Matrix class-ban lévő export() függvényt, és bekéri a felhasználótól a fájlnevet, amivel menteni szeretné.

cls:

Debugging közben elkezdett zavarni, hogy nagyon hamar tele lesz a konzol, ezért keresgélni kezdtem rá valami megoldást. Ahogy van, mind az egész két sort stackoverflow- ról másoltam; még a függvény nevét sem változtattam meg.

fajlmuveletek.py

export:

Kap egy nevet és egy Matrix-ot, amivel .txt fájlba ment. Minden értéket 2 tizedesjegy pontossággal rögzít, és módosítja az esetleges (-0) elemeket 0-ra.

beolvas:

Megpróbálja megnyitni a ’fajlnev’.txt fájlt, majd a split() miatt lista típusú tartalmát soronként a fájl listába teszi.
Hogy szövegből számok legyenek végig kell menni a fájl listán majd azok tartalmát egyesével számmá alakítani. Mielőtt visszatér, fontos átnézni, hogy minden sor ugyanannyi oszlopból áll e, hiszen ez az egyetlen olyan pont a programban, amikor hibás mátrix kerülhet a rendszerbe, ezzel kiakasztva a műveletek döntő többségét.

sor_vizsgalat:

Ellenőrzi, hogy van-e olyan sor a kapott 2D listában (mátrix sorai), aminek az elemszáma eltér a legelső (nulladik) sor hosszától. Igazzal tér vissza amennyiben a 2D lista összes sorában ugyanannyi oszlop van.