Tartalomjegyzék

- 1 Függvényhívás példa
- 2 Ciklus vagy rekurzió
  - 2.1 Faktoriális kiszámolása
  - 2.2 Ciklus el?nyei
  - 2.3 Rekurzió el?nyei
- 3 Mutable és Immutable típusok
  - 3.1 Immutable
  - 3.2 Mutable
  - 3.3 Problémák

Függvényhívás példa

Múlt órán volt részletesen szó függvényhívásról. Itt egy példa:

link

Amit érdemes megfigyelni, hogy amikor a `negyzetel()` függvény meghívja a `negyzet()` függvényt (pl. Step 8-nál), amíg a `negyzet()` fut, a `negyzetel()` még nem ért véget, csak fel lett függesztve azon a ponton, ahol vár a `negyzet()` visszatérési értékére, majd onnan folytatódik.

Ciklus vagy rekurzió

A mai órán két programozási módszerr?l lesz szó, amik már el?z? félévben is szerepeltek valamennyire. Megvizsgáljuk a különbségeket és hasonlóságokat, így talán jobban megértjük mindkett?t.

El?ször nézzünk egy nagyon egyszer? példát hogy felfrissítsük is hogy hogy m?ködnek a cilusok és a rekurzív függvények:

Faktoriális kiszámolása

Ciklus:

link

A ciklus kódjának végére érve bizonyos feltétel mellett visszaugrunk a ciklus kódjának elejére, de bizonyos elérhet? változók értéke megváltozott.

Rekurzió:

link

Ebben az esetben a függvény végén bizonyos feltétel mellett visszaugrom a függvény kódjának elejére (meghívom a függvényt), de bizonyos elérhet? változók (a függvény paraméterei) értéke megváltozott.

Ennél a példánál láthatóak a hasonlóságok a kett? között, de mik a különbségek?

Ciklus el?nyei

A ciklus fontos el?nye, hogy nem hoz egy új névteret létre: minden változó amit a függvényben használtunk, elérhet? a cikluson belül is. Ezzel ellenben ha egy függvényt, akár önmagát, hívjuk, akkor ott csak azok a változók érhet?ek el, amiket paraméterként átadunk.

Emiatt, ha pl. egy táblázatot akarunk kitölteni, akkor azt ciklussal nagyon egyszer?:

link

Ha ennek próbáljuk az analógiáját megcsinálni rekurzívan, meg lehet, de bonyolult kódot kapunk:

link

Látható, hogy az **i** és **j** változókat is a függvény paraméterévé kellett tenni, hogy elérhet?ek legyenek a bels? kódban.

Egy másik hátrány amit valamennyire mutat az el?z? kódrészlet, ha megnézzük a Python Tutor-on a futását, hogy ilyenkor a végére a függvény összes változata egyszerre fut. Ez nem minden programozási nyelven, és nem mindig, igaz, de ha igen, akkor jelent?sen több memóriát foglal ez a változat, mint a ciklussal megírt változat.

Mivel rekurziónál a régebben kiszámolt értékek lehet hogy nem elérhet?ek, ezért a gondtalanul megírt rekurzív kód lehet hogy fölöslegesen sokat számol. Példának itt van ez a **rosszul megírt** változata a fibonacci számok kiszámolásának:

link

Végigléptetve, a `fibonacci_rekurziv(3)` meg van hívva a 11-es és a 34-es lépésnél is, és mindkétszer újra ki van számolva. A matematikai gondolkodásmód szerint ugyan igaz hogy mindkét esetben a 3. fibonacci szám értékére van szükség, de programozásnál azt is végig kell gondolni hogy ez milyen lépésekkel fog járni.

Összességében a ciklus mindenképp könnyebben használható ha:

- sok változóra van szükség a kód belsejében
- egy táblázatot töltünk fel
- a ciklus sok egymástól független elemmel dolgozik

Olyan esetben is érdemesebb ciklust használni mint rekurziót, amikor a rekurzió mélysége (hány példánya fut a függvénynek egyszerre) nagyon megn?hetne, mert az a hatékonyságot csökkenti nagy memóriaigényével.

Rekurzió el?nyei

Tekintsük a következ? feladatot:

Egy *k* szám osztósora egy olyan pozitív egészekb?l álló számsor, ami *k*-val kezd?dik, 1-el végz?dik, és a következ? szám mindig az el?z? osztója. Pl. a 24, 12, 3, 1 az a 24 egy osztósora. Írjunk egy függvényt ami kiírja egy szám összes osztósorát.

Egy lehetséges megoldás rekurzióval:

link

Próbáljuk meg ugyanezt a megoldást reprodukálni ciklussal. Itt egy ilyen megoldás:

link

Ez a legegyszer?bb kód amit sikerült írnom, és még így is a kommentekkel együtt sem vagyok benne teljesen biztos hogy jól érthet?. Itt van kommentek nélkül is, hogy látható legyen hogy mennyivel hosszabb és bonyolultabb:

```
def osztosor_ciklus(k):
    l = [k, 1]
    while 1:
        print l
        while len(l) > 1:
            l[-1] = l[-1] + 1
            if l[-1] == l[-2]:
                del l[-1]
            elif l[-2] % l[-1] == 0:
                l.append(1)
                break
        else:
            return
```

Ez egy olyan feladat, amire a rekurzió alkalmasabb. Amikor ciklussal csináljuk meg ugyanazt, akkor is vigyáznunk kell, hogy a lista hányadik eleménél éppen hol tartunk, mikor kell el?re illetve visszalépnünk a listában. Ezek a rekurzív változatban a függvény hívásának és visszatérésének felelnek meg, így bizonyos mértékig "http://wiki.math.bme.huautomatikusan"http://wiki.math.bme.hu vannak kezelve.

De amit igazán fontos megérteni, hogy **továbbra is a ciklusos kód a hatékonyabb**. A rekurzív kód ugyan rövidebb, és a Python Tutor szerint kevesebb lépésb?l áll, de:

- Egyszerre a félig kész listának több másolata létezik a függvény több futó példányában.
- Maga a függvény több egyszerre futó példánya is memóriát fogyaszt, csökkenti a hatékonyságot.

Azonban ebben az esetben ez a felesleges munka nem lesz sokszorosa a szükséges munkának nagyobb esetekre se. Például úgyis a lista minden példányát ki kellett írni, azoknak mind létezni kell a függvény futása után, tehát az hogy menet közben több példány is létezik, az nem sokszorozza a memória igényt.

Viszont, a rekurzív kód könnyebben olvasható, és a m?ködése könnyebben megérthet?. Ez nem elvetend? szempont, mindenki a programozóként dolgozik, **több id?t tölt kód olvasásával, mint írásával**.

Milyen esetben érdemes tehát rekurziót használni? Amikor érthet?bb kódot ad, és a hatékonyság vesztés nem jelent?s, vagy nem számít. Néhány példa:

- Gráfok bejárása.
- Más olyan feladat, ahol a rekurzív változat egy ciklusban hívja önmagát.
- Olyan táblázatok kitöltése, ahol a táblázatnak végül csak néhány eleme számít. Ilyenkor úgynevezett letárolásos rekurziót használunk.
- Olyan problémáknál, ahol a kimenet maga rekurzív jelleg?. Ilyen példa az osztósoros feladat, vagy fraktálok kirajzolása, különféle természetes folyamatok szimulációja.

Nézünk majd példákat a gyakorlaton.

Mutable és Immutable típusok

A pythonban két fajta változó van, mutable (kb. "http://wiki.math.bme.huváltoztatható"http://wiki.math.bme.hu), és immutable (kb. "http://wiki.math.bme.hunem változtatható"http://wiki.math.bme.hu). (Inkább az angol neveket használom, mert annak hogy nem változtatható változó nincs értelme, legalábbis nem az, amit igazából jelent az immutable.)

Immutable

Az immutable változó az egyszer?bb eset, ez az olyan változó, amiben bízhatunk, hogy csak akkor változik az értéke, ha mi értéket adunk neki:

link

Itt az látható, hogy ugyan azt mondtuk, hogy "http://wiki.math.bme.hub = a"http://wiki.math.bme.hu, amikor utána megváltoztatjuk a *b* értékét, annak nincs hatása az *a* értékére.

Ilyen fajta, tehát immutable, például a:

- minden féle szám, tehát egész, valós (float) vagy akár komplex
- igaz/hamis (boolean) változók
- karakterláncok
- tuple

Mutable

Kicsit nehezebb eset a mutable változók. El?ször nézzünk példa kódot

link

Itt azt láthatjuk, hogy amikor megváltoztatjuk a *jojatekok* értékét, akkor a *jatekok* értéke is megváltozik. Ez azért van, mert ugyan *jatekok* és *jojatekok* két külön változó, de ugyanarra a listára hivatkoznak. Ezt a python tutor is jelzi a fenti kódábrázolásban, azzal hogy nem a változó neve mellett van a lista tartalma, hanem onnan mutat egy nyíl a listára, és ott látható a tartalom.

Néhány változótípus, ami mutable:

- Lista
- Szótár
- Halmaz
- Általában minden bonyolultabb dolog.

Problémák

Miért fontos ez számunkra? Profibb felhasználók számára van amikor ezt jól ki tudják használni, hogy hatékony legyen a kódjuk. Azonban a jelenlegi szinten nekünk ez több problémát okoz, mint amiben segít. Ezért most arra koncentrálok, hogy hogyan tudjuk elkerülni hogy módosítsunk valamit amit nem akarunk. Egy példa:

link

Ha csak az utolsó 4 sort olvassuk el, teljesen logikusnak t?nik a dolog, de mégsem m?ködik, mert a függvényen belül megváltoztattuk a listát amit kaptunk. Általános megoldás amit javaslok, függvényen belül soha ne változtassunk meg olyan mutable változókat amiket paraméterként kaptunk. Ha úgy egyszer? a kódunk hogy az eredeti listából/szótárból törlünk vagy módosítunk, akkor el?ször készítsünk egy másolatot. Például az el?z? kódunk javított változata így nézhet ki:

link

Lista másolásához ez a **[:]** jelzés általános módszer a másoláshoz, a szótárnak van egy külön **copy()** metódusa, amit így lehet használni:

```
szotar = {"cica" : "kitten", "kutya" : "dog"}
szotar_masolat = szotar.copy()
```

Egyel?re ennyit elég tudni a mutable típusokról, még a félév folyamán kés?bb visszatérünk rájuk.