

## Rekurzivne funkcije na seznamih

Tole je običajen način poučevanja rekurzije. Strinjam se z argumenti, zakaj je zgrešen. Vendar sem pogosto naletel na študente, ki jim je bilo tole lažje kot karkoli drugega. Zategadelj poskušam tudi tako; nekateri doživijo razsvetljenje po tej poti.

### Palindrom

Niz je palindrom, če se naprej in nazaj bere enako.

Očitno obstaja kup načinov, da preverimo, ali je niz palindrom. Najlažji v Pythonu je, da ga pač obrnemo in pogledamo, ali ostane pri tem enak, `s[::-1] == s`.

Vendar nas danes zanima rekurzivna rešitev. Zato se moramo najprej domisliti rekurzivne definicije palindroma: niz je palindrom, če sta prvi in zadnji znak enaka, vse, kar je vmes, pa je palindrom. Obenem moramo pristaviti, da so tudi prazni nizi palindromi.

Kar smo napisali, *"niz je palindrom, če je prazen ali pa je prvi znak enak zadnjemu in je vse, kar je vmes palindrom"* dobesedno prevedemo v Python.

```
def palindrom(s):  
    return s == "" or s[0] == s[-1] and palindrom(s[1:-1])  
  
palindrom("pericarežeracirep")  
True  
  
palindrom("abcba")  
True  
  
palindrom("abccba")  
True  
  
palindrom("abcdba")  
False
```

Študente tole pogosto zbega. Funkcija `palindrom` je definirana s funkcijo `palindrom`. Sama s sabo. Je to sploh dovoljeno? Deluje?

Če bi napisali

```
def palindrom(s):  
    return palindrom(s)
```

to seveda ne bi delovalo. Razlog, da gornje deluje, pa je v tem da je niz vedno krajši.

- Ko se vprašamo, ali je "abccba" palindrom, funkcija ugotovi, da sta prvi in zadnji znak enaka in pokliče neko funkcijo, ki bo preverila, ali je "bccb"

palindrom. Ta "neka funkcija" je slučajno ravno taista funkcija, a s tem ni nič narobe.

- Funkcija, katere naloga je preveriti, ali je "bccb palindrom, ugotovi, da sta prvi in zadnji znak enaka, nato pa pokliče neko funkcijo, ki bo preverila, ali je "cc" palindrom.
- Funkcija, katere naloga je preveriti, ali je "cc" palindrom, ugotovi, da sta prvi in zadnji znak enaka, nato pa pokliče neko funkcijo, ki bo preverila, ali je "" palindrom.
- Funkcija, katere naloga je preveriti, ali je "" palindrom, odgovori, da je.

Kako Python zmore slediti temu? Kako se ne zgubi v tem, do kje je prišel, kaj preverja, kaj bo rezultat tega preverjanja? To naj vas ne skrbi. Zna pač. Da bi razložili, moramo vedeti več, kot bomo izvedeli pri tem predmetu.

Funkcijam, ki kličejo same sebe, rečemo "rekurzivne funkcije".

## Vsota elementov seznama

Kako bi izračunali vsoto elementov seznama. Z zanko seveda znamo. Pa poskusimo še s trikom, kakršnega smo uporabili zgoraj. Za to spet potrebujemo rekurzivno definicijo vsote seznama - definicijo, ki se nanaša sama nase.

Takole lahko rečemo: vsoto elementov seznama dobimo tako, da k prvemu elementu prištejemo vsoto ostalih. Če je seznam prazen, pa je vsota 0.

```
def vsota(s):  
    if not s:  
        return 0  
    return s[0] + vsota(s[1:])
```

```
vsota([5, 8, 2])
```

```
15
```

## Vsebuje seznam sama soda števila?

Seznam vsebuje sama soda števila, če je prazen ali pa je prvi element sod in vsebuje preostanek seznama sama soda števila.

```
def soda(s):  
    return not s or s[0] % 2 == 0 and soda(s[1:])
```

```
soda([4, 8, 6, 1, 2, 4])
```

```
False
```

```
soda([4, 8, 6, 2, 4])
```

```
True
```

## Vsota sodih števil v seznamu

Napisati želimo funkcijo, ki vrne vsoto vseh sodih elementov v podanem seznamu.

Recimo tako:

- Če je seznam prazen, je vsota enaka 0.
- Če je ničti element sod, vrnemo vsoto ničtega elementa in vsote ostalih.
- Sicer vrnemo le vsoto ostalih.

```
def vsota_sodih(s):  
    if not s:  
        return 0  
    if s[0] % 2 == 0:  
        return s[0] + vsota_sodih(s[1:])  
    else:  
        return vsota_sodih(s[1:])
```

```
vsota_sodih([4, 2, 5, 11, 6])
```

12

## Prvi sodi element seznama

Napisati želimo funkcijo, ki vrne prvi sodi element seznama.

Če je ničti element sod, vrnemo tega. Sicer vrnemo prvega sodega med ostalimi.

```
def prvi_sodi(s):  
    if s[0] % 2 == 0:  
        return s[0]  
    else:  
        return prvi_sodi(s[1:])
```

```
prvi_sodi([5, 11, 3, 4, 8, 7, 12])
```

4

Ta funkcija ima manjšo težavo, kadar v seznamu ni nobenega sodega elementa. Ker prvi ni sod, išče sodega v preostanku. Ker prvi iz preostanka ni sod, išče sodega v preostanku. Ker prvi v tem preostanku ni sod ... in tako naprej, dokler ne pride do praznega seznama. Takrat bomo v prvi vrstici funkcije spraševali po prvem (no, ničtem) elementu, `s[0]` in Python bo javil napako, `Index out of range`.

Recimo, da naj funkcija v takšnem primeru vrne `None`.

```
def prvi_sodi(s):  
    if not s:  
        return None  
    if s[0] % 2 == 0:
```

```

        return s[0]
    else:
        return prvi_sodi(s[1:])

```

## Zadnji sodi element seznama

Bolj zanimiva naloga je poiskati zadnji sodi element seznama. (Seveda, če se dogovorimo, da bomo spet gledali prvi element in ostanek; če smemo iti od zadnjega elementa proti začetku, je to seveda natančno enaka naloga kot prejšnja, spremené se le indeksi.)

Tako razmišljamo.

- Če je seznam prazen, vrnemo `None`, itak.
- Če ni prazen, (rekurzivno) preverimo, ali se zadnji sodi element nahaja v preostanku seznama. Če v odgovor ne dobimo `None`, je to pač zadnji element seznama.
- Če dobimo v odgovor `None`, pa v ostanku ni sodih elementov. Preverimo, ali je prvi element sod; če je tako, je to tudi zadnji sodi element in ga moramo vrniti.
- Sicer ni sodega elementa v ostanku, pa tudi prvi element ni sod. Potemtakem sodih elementov ni in vrnemo `None`.

```

def zadnji_sodi(s):
    if s == []:
        return None
    zadnji = zadnji_sodi(s[1:])
    if zadnji is not None:
        return zadnji
    if s[0] % 2 == 0:
        return s[0]
    return None

```

```
zadnji_sodi([1, 3, 5, 4, 7, 3, 12, 8, 1, 3])
```

8

## Vsi sodi elementi seznama

Funkcija, ki vrne vse sode elemente seznama je železna klasika napačnih rešitev. Tipični prvi poskus je

```

def vsi_sodi(s):
    if s == []:
        return []

    t = []
    if s[0] % 2 == 0:
        t.append(s[0])

```

```

    vsi_sodi(s[1:])
    return t

```

```

vsi_sodi([4, 2, 5, 11, 6])

```

```

[4]

```

To ne deluje, ker vsaka funkcija naredi svoj seznam `t`, vanj doda en sem element in ga vrne. Na koncu dobimo le seznam, ki ga ustvari najbolj zunanja funkcija.

V drugem poskusu poskusimo zagotoviti, da bodo vse funkcije uporabljale isti `t` tako, da ga postavimo pred funkcijo - tako da postane globalna spremenljivka.

```

t = []

```

```

def vsi_sodi(s):
    if s == []:
        return []

    if s[0] % 2 == 0:
        t.append(s[0])
    vsi_sodi(s[1:])
    return t

```

To deluje - vendar le enkrat. Če tole poskusimo poklicati večkrat, bo funkcija vedno dodajale elemente v ta seznam - in to ne bo dobro.

```

vsi_sodi([4, 2, 5, 11, 6])

```

```

[4, 2, 6]

```

```

vsi_sodi([4, 2, 5, 11, 6])

```

```

[4, 2, 6, 4, 2, 6]

```

```

vsi_sodi([4, 2, 5, 11, 6])

```

```

[4, 2, 6, 4, 2, 6, 4, 2, 6]

```

To lahko poskušamo rešiti na vse žive načine. Recimo tako, da seznam praznimo na začetku funkcije.

```

t = []

```

```

def vsi_sodi(s):
    t.clear()

    if s == []:
        return t

    if s[0] % 2 == 0:
        t.append(s[0])

```

```

    vsi_sodi(s[1:])
    return t

vsi_sodi([4, 2, 5, 11, 6])

[]

```

To ne deluje, ker se seznam sprazni ob *vsakem* klicu. Obupani študenti potem pišejo maile, kako narediti, da se bo seznam spraznil samo prvič.

Ne gre. Tu ni problem v tem, da ne znamo prav sprogramirati, temveč v tem, da narobe razmišljamo. Kakšna je rekurzivna definicija vseh sodih elementov seznama? To je seznam, ki vsebuje prvi element, če je le-ta sod, poleg tega pa vse sode elemente iz ostanka. Recimo tako:

```

def vsi_sodi(s): if s == []: return []
if s[0] % 2 == 0:
    return [s[0]] + vsi_sodi(s[1:])
else:
    return vsi_sodi(s[1:])

```

Obstajajo tudi elegantnejši načini, obstajajo tudi hitrejši načini. A za nas je tale dovolj dober.

## Sodolihi in lihosodi

Seznam je sodolih, če se v njem izmenjujejo sodi in lihi elementi, začenši s sodim.

Seznam je lihosod, če se v njem izmenjujejo lihi in sodi elementi, začenši z lihim.

Napisati moramo funkciji `sodolih` in `lihosod`, ki preverita, ali je seznam tak, kot pravi ime funkcije.

```

def sodolih(s):
    return s == [] or s[0] % 2 == 0 and lihosod(s[1:])

def lihosod(s):
    return s == [] or s[0] % 2 == 1 and sodolih(s[1:])

```

```
sodolih([4, 5, 8, 13, 2, 7, 0])
```

True

Tu vidimo primer, ko se dve funkciji kličeta med sabo. Tudi to je rekurzija in tudi za takšno obliko ni razloga, da ne bi delovala.