Rekurze

Jiří Zacpal

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

KMI/ZPP1 Základy programování v Pythonu 1

2. úkol (řešení do 4.12.2023)

Vytvořte datovou strukturu pro uložení informací o skupině osob. O každé osobě budete ukládat tyto informace: `jméno, příjmení, adresa bydliště, datum narození, telefon a e-mail`. Napište tyto funkce:

```
vytvor_osobu(jmeno, prijmeni, adresa, den, mesic, rok, telefon, email)
```

- která vytvoří a vrátí novou osobu.

```
vloz(s, o)
```

- která vloží osobu o do seznamu s.

```
najdi_osobu(kde, co, s)
```

- která vrátí True pokud v seznamu s existuje osoba podle zadaných parametrů: v proměnné zadané parametrem kde (jméno, příjmení, adresa, email, telefon), obsahuje řetězec, zadaný parametrem co.

```
nejmladsi(s)
```

- která vrátí nejmladší osobu ze seznamu osob.

```
tisk_osoby(o)
```

- která vytiskne osobu o.

```
tisk(s)
```

- která vytiskne osoby ze seznamu s.

2. úkol

Všechny funkce otestujte v tomto kódu:

```
seznam_osob=[]
o1 = vytvor_osobu("Alice", "Pokorna", "Holicka 62", 2, 1, 1992, "214 145 478", "alice.pokorna@email.cz")
o2 = vytvor_osobu("Pavel", "Novak", "tr. 17 listopadu 24", 13, 1, 1992, "654 784 478", "pavel.novak@seznam.cz")
o3 = vytvor_osobu("Ales", "Maly", "Holicka 62", 6, 5, 1989, "772 847 457", "ales.maly@upol.cz")
vloz(seznam_osob, o1)
vloz(seznam_osob, o2)
vloz(seznam_osob, o3)
tisk(seznam_osob)

if (najdi_osobu("jmeno", "Alice", seznam_osob)):
    print("Alice nalezena.\n")
else:
    print("Alice nenalezena.\n")

if (najdi_osobu("prijmeni", "Novotny", seznam_osob)):
    print("Novotny nalezen.\n")
else:
    print("Novotny nenalezen.\n")

o = nejmladsi(seznam_osob)
print("Nejmladsi osobou v seznamu je ", tisk_osoby(o))
```

Volání funkce

- Již víme, že z těla uživatelské funkce můžeme volat jinou uživatelskou funkci. To se například děje v následujícím programu počítajícím součet čtverců dvou čísel:

```
def ctverec (x):
    return x*x

def soucet_ctvercu(x,y):
    return ctverec(x)+ctverec(y)
```

Zanoření volání

- Vidíme, jak se dvakrát při volání funkce `soucet_ctvercu` volala funkce `ctverec`.
- Volání uživatelské funkce při vykonávání uživatelské funkce se nazývá **zanořené**.
- Můžeme také mluvit o úrovni zanoření:
  - **úroveň jedna** - volání uživatelské funkce z globálního prostředí,
  - **vyšší úroveň** - volání uživatelské funkce v rámci vykonávání těla uživatelské funkce má o jedna vetší úroveň než její volání.
- Například volání square v rámci `soucet_ctvercu` vyhodnocení výrazu `soucet_ctvercu(3,2)` má úroveň zanoření dva.
- Jistě si dokážete představit program, jehož vykonávání by probíhalo v úrovni zanoření tři.

Příklad

- Napišme si funkci pro výpočet faktoriálu pro číslo n:

```
def faktorial(n):
    f=1
    for i in range(1,n+1):
        f*=i
    return f
```

Příklad

- Pro výpočet faktoriálu existuje předpis:

$$0! = 1$$
$$\forall n \geq 1\colon\ n! = n \cdot (n-1)!$$

- Pokud bychom vyšli z tohoto předpisu, můžeme zapsat algoritmus takto:

```
def faktorial(n):
    if(n==0):
        return 1
    else:
        return n*faktorial(n-1)
```

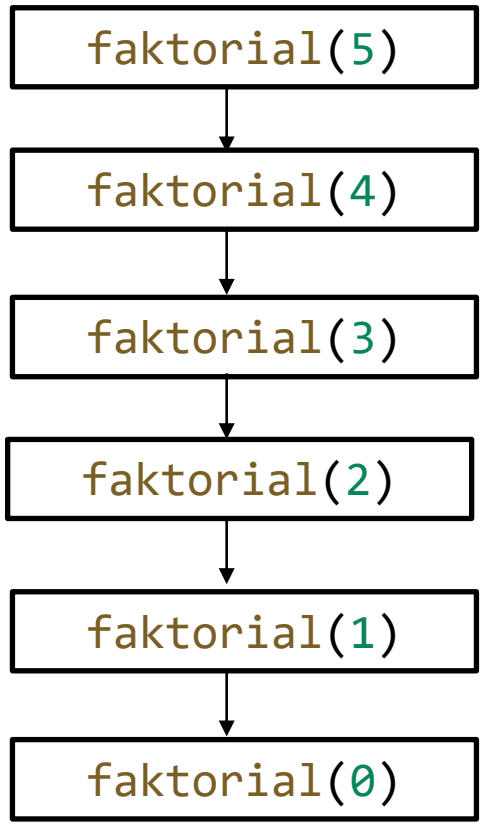
Rekurze

Rekurze

- Pojem rekurze, v programování, označuje situaci, kdy funkce volá ve svém těle sebe sama.
- Rekurze obvykle umožňuje jednoduché řešení problémů, které je možné rozložit na menší podproblémy.
- Součástí rekurzivní funkce by měla být **limitní podmínka** určující, kdy se má vnořování zastavit.

```
def faktorial(n):
    if(n==0):
        return 1
    else:
        return n*faktorial(n-1)

print(faktorial(5))
```



Chyba při rekurzi

- S rekurzí se pojí nový druh chyb:

```
print(faktorial(1000))
RecursionError: maximum recursion depth exceeded in comparison
```

- Logicky je sice program v pořádku, ale interpret Pythonu umožnuje pouze úroveň zanoření tisíc.
- Ve skutečnosti je tato hodnota o pár jednotek nižší, protože sám interpret pro svoji činnost nějakou úroveň potřebuje.

- Všimnete si, že volání funkce se záporným argumentem nyní nespadne do nekonečné smyčky, ale skonči chybou:

```
print(faktorial(-1))
RecursionError: maximum recursion depth exceeded in comparison
```

- Interpret nepozná, zda se jedná o nekonečnou smyčku a nebo o příliš náročný výpočet.

Příklad

- Algoritmus pro výpočet faktoriálu opravíme:

```
def faktorial(n):
    if(n==0):
        return 1
    elif(n>=1):
        return n*faktorial(n-1)
```

Typy rekurze

- **Lineární rekurze**
  - nastává, pokud podprogram při vykonávání svého úkolu volá sama sebe pouze jednou,
  - vytváří se takto lineární struktura postupně volaných podprogramů.
- **Stromová rekurze**
  - nastává, pokud se funkce nebo procedura v rámci jednoho vykonání svého úkolu vyvolá vícekrát,
  - strukturu volání je možné znázornit jako zakořeněný strom,
  - pro dvě volání v jednom průchodu vzniká binární strom, pro tři ternární strom, atd.
  - počet rekurzivních volání nemusí být konstantní, např. při rekurzivním procházení grafu voláme zpracování na všechny sousedy vrcholu, a těch je obecně různý počet.

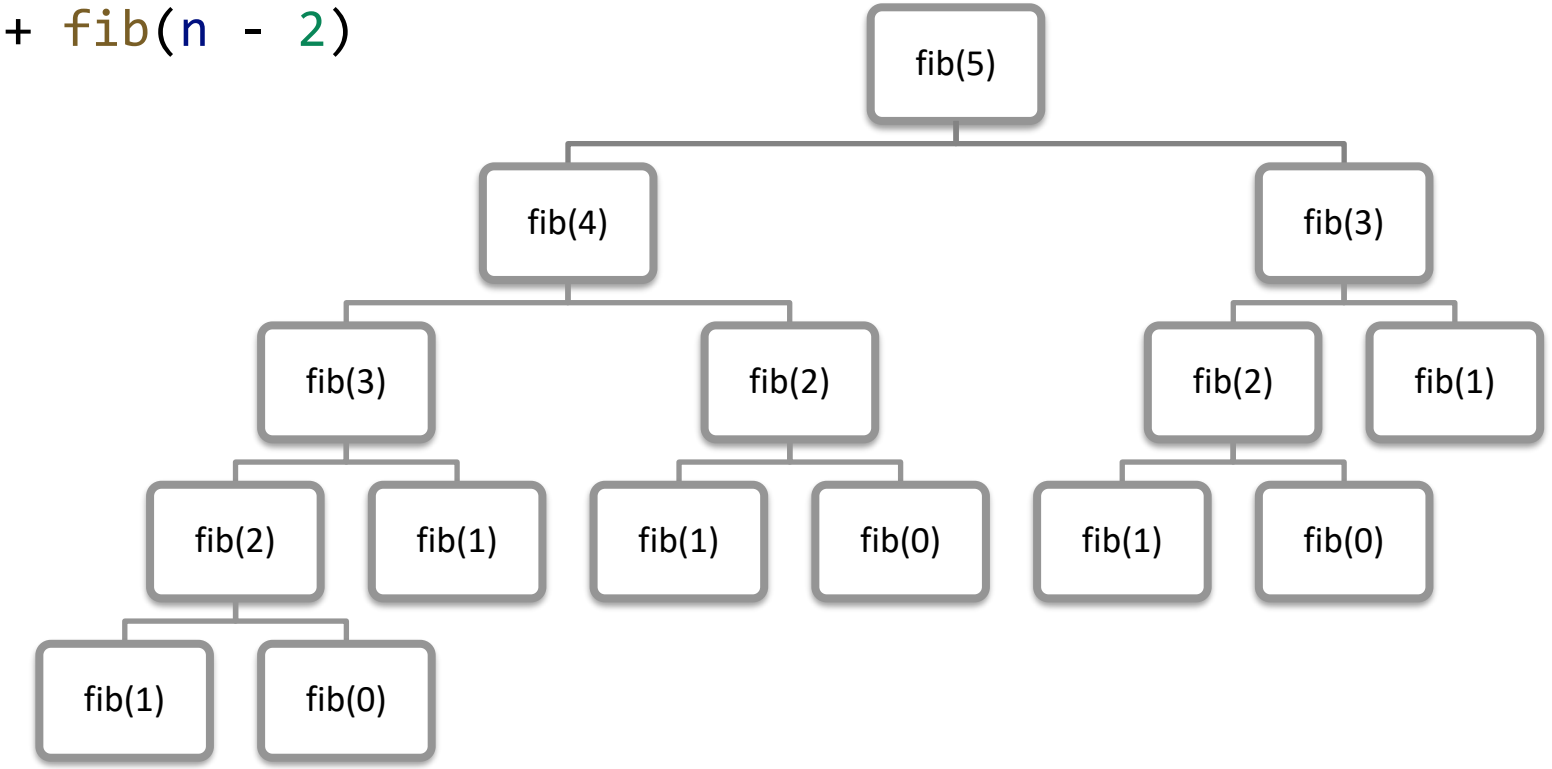
Příklad - Fibonacciho posloupnost

- Jako Fibonacciho posloupnost je v matematice označována nekonečná posloupnost přirozených čísel, začínající 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … (čísla nacházející se ve Fibonacciho posloupnosti jsou někdy nazývána Fibonacciho čísla), kde každé číslo je součtem dvou předchozích.
- Rekurzivní definice Fibonacciho posloupnosti tedy je:

$$F(n) = \begin{cases} 0\,, & \text{pro } n = 0\,; \\ 1, & \text{pro } n = 1; \\ F(n-1) + F(n-2) & \text{jinak.} \end{cases}$$

Příklad - Fibonacciho posloupnost

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```



Rekurze

- Rekurzivní funkce jsou obvykle jednodušší a kratší než jejich iterativní protějšky.

```
def fib(n):
    prvni_clen = 0
    druhy_clen = 1
    for i in range(0, n):
        pomocna = prvni_clen
        prvni_clen = druhy_clen
        druhy_clen = pomocna + druhy_clen
    return prvni_clen
```

Koncová rekurze

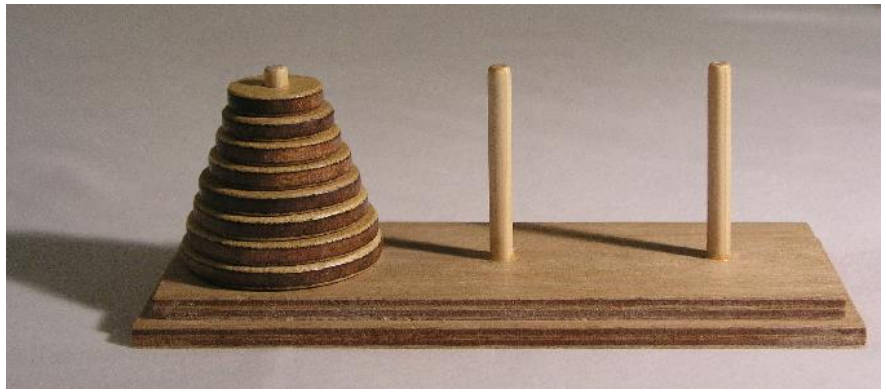
- Při koncové rekurzi je rekurzivní volání funkce posledním příkazem v těle funkce, je pouze jedno a výsledek rekurzivního volání je použit jako návratová hodnota funkce.
- V případě koncové rekurze obvykle není nutné udržovat všechna nedokončená rekurzivní volání => **výpočet se podstatně zrychlí**.

```
def fib(n, prvni_clen, druhy_clen):
    if n <= 1:
        return prvni_clen
    else:
        return fib(n - 1, druhy_clen, prvni_clen + druhy_clen)

print(fib(55, 0, 1))
```

Příklad – Hanojské věže

- Hanojské věže
  - je matematický hlavolam, který vymyslel francouzský matematik Édouard Lucas v roce 1883.
  - Skládá se ze tří kolíků (věží).
  - Na začátku je na jednom z nich nasazeno několik kotoučů různých poloměrů, seřazených od největšího (vespod) po nejmenší (nahoře).
  - Úkolem řešitele je přemístit všechny kotouče na druhou věž (třetí přitom využije jako pomocnou pro dočasné odkládání) podle následujících pravidel:
  - V jednom tahu lze přemístit jen jeden kotouč.
  - Jeden tah sestává z vzetí vrchního kotouče z některé věže a jeho položení na vrchol jiné věže.
  - Je zakázáno položit větší kotouč na menší.



Příklad – Hanojské věže

```
def hanoj(n, pocatecni, cilova, odkladaci):
    if n>1:
        hanoj(n-1, pocatecni, odkladaci, cilova)
    print("Přesuneme kotouč z věže ", pocatecni, " na věž ",
cilova)
    if n>1:
        hanoj(n-1, odkladaci, cilova, pocatecni)

hanoj(5,'A','B','C')
```