

Příklad

Napište program, který přečte text ze vstupu ukončený slovem **KONEC** a vypíše dvacet slov s největším počtem výskytů.

Funkční dekompozice

= rozdělení práce pomocí funkcí

Programování shora (dolů)

= řešení úlohy rozkladem na pod-úlohy

= využívá volání (zatím neexistujících) podprogramů

Programování zdola (nahoru)

= vytváření podprogramů (řešení pod-úloh),

o kterých myslíme, že je budeme potřebovat
a následně z nich skládáme řešení větších
pod-úloh, až k hlavní úloze

Ladění zdola

testovací (pod)programy

Ladění shora

náhradní obsah podprogramů

Testovací podprogramy, vyhodnocování správnosti,
počítání chyb, automatické testování – později.

Složitější proměnné

- **OBJEKT** `zelva.forward(100)`
- **STRING** (textový řetězec) `"abcd"`
neměnitelná
- **TUPLE** (n-tice) `( "NPRG031", 1 )`
neměnitelná
- **LIST** (seznam) `[ 1, 2, 5, 3, 2 ]`
seznam s pořadím
- **SET** (množina) `{ 1, 5, 2 }`
bez pořadí, každá hodnota je nebo není (max jednou)
- **DICTIONARY** (slovník) `{ 1:"a", 5:"xyz", 2:22 }`
dvojice klíč:hodnota

Tuple (n-tice)

- neměnitelná

```
>>> t = (10, 20)
```

```
>>> t[0]
```

```
10
```

```
>>> t[1]
```

```
20
```

```
>>>
```

Tuple (n-tice) – k čemu je to dobré

Třeba funkce může vrátet tuple.

Násobné dosazení

```
>>> a,b,c = (10,20,30)
```

nebo:

```
>>> a,b = b,a
```

Na těch závorkách nezáleží, rozhoduje čárka!

Dictionary (slovník)

- Prvky jsou dvojice klíč:hodnota
- „asociativní pole“
- Každý klíč nejvýše jednou
- Uložené bez pořadí, vyhledává se podle klíče

```
>>> d = { 'jablko' : 'apple' }
```

```
>>> d[ 'slovo' ] = 'word'
```

```
>>> d
```

```
{ 'jablko' : 'apple', 'slovo' : 'word' }
```

```
>>> d[ 'jablko' ]
```

```
'apple'
```

Dictionary (slovník)

Vymazání prvku

```
>>> del d['jablko']  
>>> d  
  
{ 'slovo': 'word' }
```

Vytvoření prázdného seznamu

```
>>> d = {} # rychlejší  
>>> d = dict() # cca 2x pomalejší
```

Dictionary (slovník)

Dotaz na existenci klíče

```
>>> 'slovo' in d  
True
```

Cyklus

```
>>> for klic in d:  
        print( klic, d[klic] )  
  
slovo word
```

Dictionary (slovník)

Indexování neexistujícím klíčem

```
>>> d['slovo']
```

```
word
```

```
>>> d['mrkev']
```

```
...
```

```
KeyError: 'mrkev'
```

Dictionary (slovník) - klíč

Klíč může být jakákoliv **neměnitelná** hodnota.

```
>>> d[ (25,17, 'xyz') ] = 'trojice'
```

```
>>> d  
{ 'slovo': 'word', (25, 17, 'xyz') :  
  'trojice' }
```

Dictionary (slovník) - proč

Dvojice bychom mohli ukládat i do LIST-u,
proč potřebujeme DICTIONARY?

PROČ 1: Možná nehledáme PRVEK toho seznamu, ale jenom jeho klíč (a pak se teprve chceme podívat na hodnotu).

PROČ 2: Vyhledávání v LISTu postupně prochází, vyhledávání v DICTIONARY je pomocí hashování a to je rychlejší ($O(n)$ vs. $O(1)^*$).

*) přibližně, nepřesně, podrobnosti později (Algoritmizace)

Příklad

Změnit započítávání slov v předchozím příkladu

- jak je těžké dělat změny v hotovém programu?
- jak je pravděpodobné, že tím něco rozbijeme?

Opakování:

Funkce je nástroj pro oddělení
CO TO DĚLÁ od **JAK TO DĚLÁ**.

Objekt (podrobnější pohled)

Obsahuje DATA a FUNKCE

Přístup k nim pomocí tečky

```
>>> a = 5
>>> a.bit_length()
3
>>> zelva.left(90)
>>> zelva.DEFAULT_ANGLEOFFSET
0
```

Skoro všechno v Pythonu je objekt.

Proč objekty (nejen v Pythonu)

Objektová dekompozice

- Rozdělení programu na části, organizace kódu
- Další **CO** to dělá **×** **JAK** to dělá
- Řeší problémy, na které jsme zatím potřebovali globální proměnné (číst vstup, seznam slov...)
- Lze chápat i tak, že obsahují dodatečné parametry svých funkcí...

Objekt obsahuje DATA a FUNKCE

...a ne vše, co obsahuje, je veřejné!

(v Pythonu tak úplně neplatí)

Příklad použití objektů

Napište program,

který čte ze vstupu slova a tiskne je do řádek,
které nebudou delší než 30 znaků.

Kde hledat chyby a kde dělat změny

= další způsob dekompozice!

(funkční, objektová, modulární)

Objekty a třídy (nejen v Pythonu)

Pojmy:

Třída (class) je datový typ (formička)

Objekt je instance (bábovička)

TH je objekt třídy Člověk

V některých jazycích třídy nejsou.

Definice třídy

```
class Komplex:  
    pass # dál už nic není
```

Vytvoření objektu

```
k = Komplex() # nový objekt třídy Komplex  
k.Re = 3.00  
k.Im = 4.00  
print( k.Re, k.Im )
```

3.0 4.0

Přidat funkci

funkce má povinný první parametr **self** = TENHLE objekt,
který (ten parametr) se při volání neuvádí

```
class Komplex:  
    def abs( self ):  
        return (self.Re**2+self.Im**2 )**(1/2)
```

```
k = Komplex()  
k.Re = 3.00  
k.Im = 4.00  
print( k.Re, k.Im, k.abs() ) # bez parametru!
```

3.0 4.0 5.0

Konstruktor

- funkce pro inicializaci nového objektu
- volá se při vytváření objektu
- vždycky se jmenuje `__init__`
- ...takže může být jenom jeden
(na rozdíl od jiných jazyků)
- má také povinný první parametr `self`
- když konstruktor nenadefinujeme,
použije se standardní (prázdný) konstruktor

Konstruktor

```
class Komplex:
    def abs( self ):
        return (self.Re**2+self.Im**2 )**(1/2)
    def __init__( self, Re, Im ):
        self.Re = Re
        self.Im = Im
```

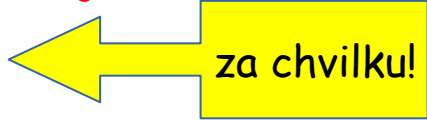
```
k = Komplex( 3.00, 4.00 )
print( k.Re, k.Im, k.abs() )
```

3.0 4.0 5.0

Další podrobnosti o objektech

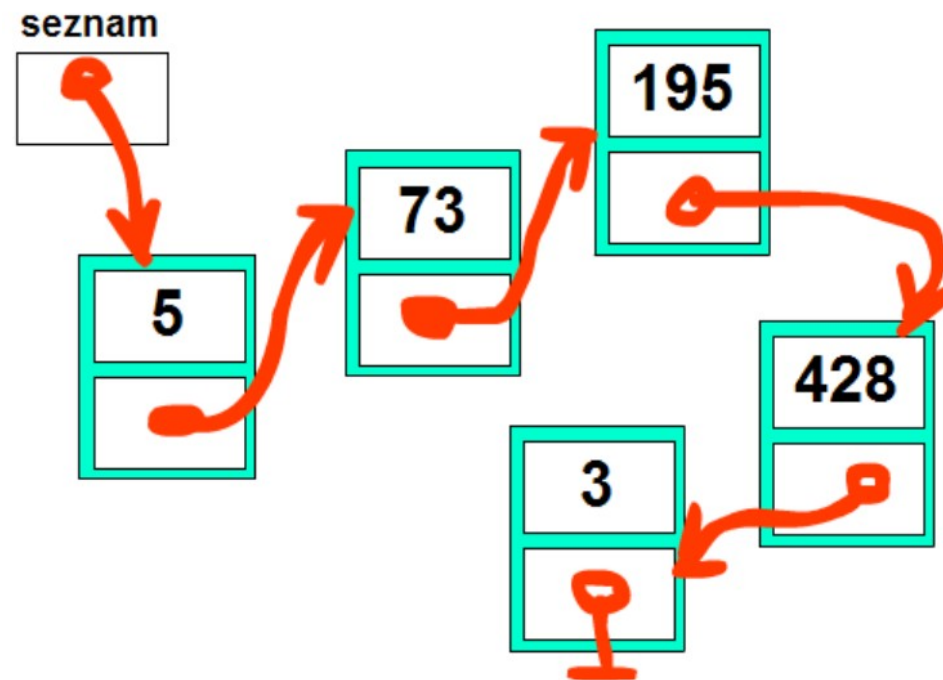
...někdy jindy.

Dynamické proměnné

- Proměnné, které vznikají (a zanikají) během výpočtu
- Mohou mít různou velikost
- Zvláštní oblast paměti = halda
- Nepotřebné proměnné se automaticky uklízejí
(ne ve všech jazycích, ale v Pythonu ANO)
= garbage collector
- Proměnná, která vypadá, že obsahuje třeba seznam,
ve skutečnosti obsahuje pouze odkaz (adresu do haldy)
- Příklad: list, string, (dlouhé) celé číslo, slovník, objekt
- ...ale také spojový seznam (vytvořený z objektů) nebo strom (také vytvořený z objektů) 
- prázdný ukazatel None (domluvená hodnota nikam neukazuje)

Lineární spojový seznam

- složený z prvků
- každý prvek obsahuje hodnotu a odkaz na další prvek



Lineární spojový seznam

```
class Prvek:  
    def __init__(self, x, dalsi):  
        self.x = x  
        self.dalsi = další
```

```
p1 = Prvek(10, None)  
p2 = Prvek(20, None)  
p3 = Prvek(30, None)  
p1.dalsi = p2  
p2.dalsi = p3  
seznam = p1
```

...nebo:

```
p3 = Prvek(30, None)  
p2 = Prvek(20, p3)  
p1 = Prvek(10, p2)  
seznam = p1
```

...nebo taky:

```
seznam = Prvek(10, Prvek(20, Prvek(30, None)))
```

Lineární spojový seznam

```
def vytiskni(s):  
    while s is not None:  
        print(s.x, end=' ')  
        s = s.dalsi
```

Vytiskni(seznam)

1 2 3

Lineární spojový seznam

...by taky mohl být objekt!

Operace...

- přidat prvek na začátek
- přidat prvek na konec
- vymazat první prvek
- vymazat poslední prvek
- ...

...a jejich složitosti.