

Datové typy

IB113
Radek Pelánek

2021

Rozcvička: Co je jiné?



- smrk, lípa, borovice, jedle
- prase, koza, husa, ovce

Odd one out: varianta pro pokročilé



Odd one out: varianta pro začátečníky

prase, pes, prase, prase

Odd one out: varianta pro začátečníky

prase, pes, prase, prase

- vstup: seznam
- výstup: prvek, který je v seznamu osamocen

Odd one out: základní řešení

```
def odd_one_out(alist):  
    for x in alist:  
        count = 0  
        for y in alist:  
            if x == y:  
                count += 1  
        if count == 1:  
            return x  
    return None
```

zpracování dotazníku, hledání cesty v bludišti, předpověď počasí, úprava obrázků

- Jaká data budu zpracovávat?
- Jaká data budu potřebovat k řešení problému?
- Jaké operace s daty budu chtít provádět?
- Jak rychle budu chtít, aby operace s daty fungovaly?

Oddělení dat a funkcionality

Důležitý princip v mnoha oblastech informatiky

- webový portál: oddělení funkcionality, dat a vzhledu
- typická realizace:
 - funkcionalita – program (Python, PHP...)
 - data – databáze (MySQL...)
 - vzhled – grafický styl (CSS)

Ukázka nevhodného kódu

```
...  
if currency == 'euro':  
    value = amount * 25.72  
elif currency == 'dollar':  
    value = amount * 21.90  
elif currency == 'rupee':  
    value = amount * 0.34  
...
```

Reálnější příklad

```
def print_stats(stat_data):  
    ...  
    if success_rate < 0.5:  
        bg_color = "red"  
    elif success_rate < 0.85:  
        bg_color = "yellow"  
    else:  
        bg_color = "green"  
    ...
```

```
def print_stats(stat_data):  
    ...  
    bg_color = get_color(success_rate,  
                          [0.5, 0.85, 1],  
                          ["red", "yellow", "green"])  
    ...
```

Další úpravy:

- parametry jako globální konstanty „vytknuté“ na začátek kódu
- použití nějaké standardní „colormap“ (tip: viridis)

např. https://seaborn.pydata.org/tutorial/color_palettes.html

Abstraktní datové typy v IB113

- abstraktní datové typy – základní myšlenka
- fronta, zásobník, množina – ilustrace principu abstraktních datových typů, základní povědomí
- slovník – důkladnější probrání a procvičení

dnes: rychlý přehled

příště: rozsáhlejší příklady

Abstraktní datové typy

Datový typ

- rozsah hodnot, které patří do daného typu
- operace, které je možno s těmito hodnotami provádět

Abstraktní datový typ (ADT)

- rozhraní
- popis operací, které chceme provádět (případně i složitost)

Konkrétní datová struktura

- implementace
- přesný popis uložení dat v paměti
- definice funkcí pro práci s těmito daty

Poznámky: hranice není vždy úplně ostrá, rozdíl mezi „formálním“ a „praktickým“ pojetím ADT; nejednotná terminologie „datový typ“, „datová struktura“

Dva pohledy na data

- abstraktní, „uživatelský“
 - operace, která s daty budu provádět
 - co musí operace splňovat, efektivita. . .
 - *množina: ulož, najdi, vymaž*
 - tento předmět
- konkrétní, implementační
 - jak jsou data reprezentována v paměti
 - jak jsou operace implementovány
 - *hašovací tabulka, binární vyhledávací strom*
 - IB002

Abstraktní pohled na data

Výhody:

- snadný vývoj
- jednodušší přemýšlení o problémech

Riziko:

- svádí k ignorování efektivity
 - typický příklad: použití `in` pro vyhledávání v seznamech

Nejpoužívanější ADT:

- seznam
- zásobník
- fronta, prioritní fronta
- množina
- slovník (asociativní pole)

Datový typ seznam: různé varianty

- obsahuje posloupnost prvků
 - stejného typu
 - různého typu
- přidání prvku
 - na začátek
 - na konec
 - na určené místo
- odebrání prvku
 - ze začátku
 - z konce
 - konkrétní prvek
- test prázdnoty, délky
- případně další operace, např. přístup pomocí indexu

Seznamy v Pythonu

opakování

- seznamy v Pythonu velmi obecné
- prvky mohou být různých typů
- přístup skrze indexy
- indexování od konce pomocí záporných čísel
- seznamy lze modifikovat na libovolné pozici

```
a = ['bacon', 'eggs', 'spam', 42]
print(a[1:3])           # ['eggs', 'spam']
print(a[-2:-4:-1])     # ['spam', 'eggs']
a[-1] = 17
print(a)               # ['bacon', 'eggs', 'spam', 17]
print(len(a))         # 4
```

Seznamy v Pythonu – operace

```
s = [4, 1, 3] # seznam
s.append(x)   # přidá prvek x na konec
s.extend(t)   # přidá všechny prvky t na konec
s.insert(i, x) # přidá prvek x před prvek na pozici i
s.remove(x)   # odstraní první prvek rovný x
s.pop(i)      # odstraní (a vrátí) prvek na pozici i
s.pop()       # odstraní (a vrátí) poslední prvek
s.index(x)    # vrátí index prvního prvku rovného x
s.count(x)    # vrátí počet výskytů prvků rovných x
s.sort()      # seřadí seznam
s.reverse()   # obrátí seznam
x in s        # test, zda seznam obsahuje x
              # (lineární průchod seznamem!)
```

Odd one out: řešení pomocí count

```
def odd_one_out(alist):  
    for x in alist:  
        if alist.count(x) == 1:  
            return x  
    return None
```

Generátorová notace pro seznamy

list comprehension

specialita Pythonu

```
s = [x for x in range(1, 7)]  
print(s)      # [1, 2, 3, 4, 5, 6]
```

```
s = [2 * x for x in range(1, 7)]  
print(s)      # [2, 4, 6, 8, 10, 12]
```

```
s = [(a, b) for a in range(1, 5)  
      for b in ["A", "B"]]  
print(s)      # [(1, 'A'), (1, 'B'), (2, 'A'),  
                # (2, 'B'), ...]
```

Odd one out: generátorová notace

```
def odd_one_out(alist):  
    return [x for x in alist  
            if alist.count(x) == 1]
```

Pozn. Mírně odlišné chování od předchozích ukázek – vrací seznam všech osamocených.

Vnořené seznamy

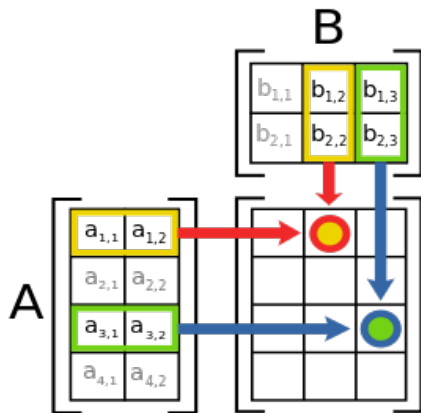
- prvky seznamů mohou být opět seznamy
- použití: vícerozměrná data (např. matice)

```
mat = [[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]]  
print(mat[1][2])      # 6
```

```
def null_matrix(m, n):  
    return [[0 for col in range(n)]  
            for row in range(m)]
```

Pozn. efektivnější způsob práce s maticemi: knihovna numpy

Násobení matic



Wikipedia

Vnořené seznamy: Násobení matic

```
def matrix_mult(matL, matR):  
    rows = len(matL)  
    cols = len(matR[0])  
    common = len(matR)  
    result = null_matrix(rows, cols)  
    for i in range(rows):  
        for j in range(cols):  
            for k in range(common):  
                result[i][j] += matL[i][k] * matR[k][j]  
    return result
```

Interpretace dvojitého indexování

- „data [i] [j] “
 - data indexují „dvojicí čísel“
 - intuitivní
 - neodpovídá implementaci v Pythonu
- „data[i] [j] “
 - indexují prvním číslem, dostanu seznam
 - tento seznam indexují druhým číslem
 - speciální případ obecného postupu

Pozn. V Pythonu můžeme mít i `data[i,j]` – to však není seznam, ale slovník indexovaný dvojicí. Více později.

Interpretace složitějšího indexování

čteme „zleva“, resp. „z vnitřku“:

- `mat[2][::-1][0]`
- `sorted(mat[1])[2]`
- `(mat[0]*5)[7]`

Čistě ilustrativní příklady, rozhodně ne ukázky pěkného kódu.

Fronta a zásobník

fronta a zásobník – „seznamy s omezenými možnostmi“

Proč používat něco omezeného?

Fronta a zásobník

fronta a zásobník – „seznamy s omezenými možnostmi“

Proč používat něco omezeného?

- vyšší efektivita implementace
- čitelnější a čistější kód
- „sebe-kontrola“

Sebe-kontrola, historická poznámka: GOTO; xkcd.com/292/



- LIFO = *Last In First Out*
- operace
 - push (vložení)
 - pop (odstranění)
 - top (náhled na horní prvek)
 - empty (test prázdnoty)
- mnohá použití
 - procházení grafů
 - analýza syntaxe
 - vyhodnocování výrazů
 - rekurze

Zásobník v Pythonu

implementace pomocí seznamu

- místo push máme append
- místo top máme [-1]

```
def push(stack, element):  
    stack.append(element)
```

```
def pop(stack):  
    return stack.pop()
```

```
def top(stack):  
    return stack[-1]
```

```
def empty(stack):  
    return stack.empty()
```


Příklad aplikace: postfixová notace

- infixová notace
 - operátory mezi operandy
 - $(3 + 7) * 9$
 - je třeba používat závorky
- prefixová notace (polská notace)
 - operátory před operandy
 - $* + 3 7 9$
 - nepotřebujeme závorky
- postfixová notace (reverzní polská notace, RPN)
 - operátory za operandy
 - $3 7 + 9 *$
 - nepotřebujeme závorky

využití zásobníku: převod mezi notacemi, vyhodnocení postfixové notace

Vyhodnocení postfixové notace

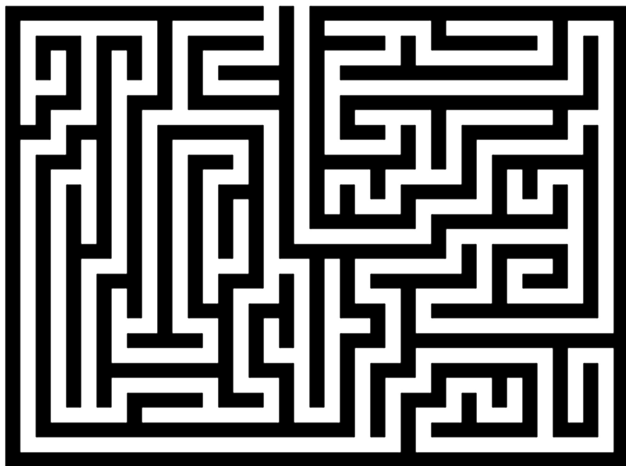
```
def eval_postfix(line):
    stack = []
    for token in line.split():
        if token == '*':
            b = pop(stack)
            a = pop(stack)
            push(stack, a * b)
        elif token == '+':
            b = pop(stack)
            a = pop(stack)
            push(stack, a + b)
        else:
            push(stack, int(token))
    return top(stack)
```

Vyhodnocení postfixové notace

vstup	akce	zásobník
7 4 7 + * 8 +	push	
4 7 + * 8 +	push	7
7 + * 8 +	push	7 4
+ * 8 +	+	7 4 7
* 8 +	*	7 11
8 +	push	77
+	+	77 8
		85

- FIFO = *First In First Out*
- operace
 - enqueue (vložení)
 - dequeue (odstranění)
 - front (náhled na přední prvek)
 - empty (test prázdnosti)
- použití
 - zpracovávání příchozích požadavků
 - procházení grafu do šířky
- pokročilejší varianta: prioritní fronta

Příklad aplikace: bludiště



- implementace pomocí seznamů snadná, ale neefektivní
 - přidávání a odebírání na začátku seznamu vyžaduje přesun
 - pomalé pro dlouhé fronty
- použití knihovny `collections`
 - datový typ `deque` (oboustranná fronta)
 - vložení do fronty pomocí `append`
 - odebrání z fronty pomocí `popleft`
 - přední prvek fronty je `[0]`

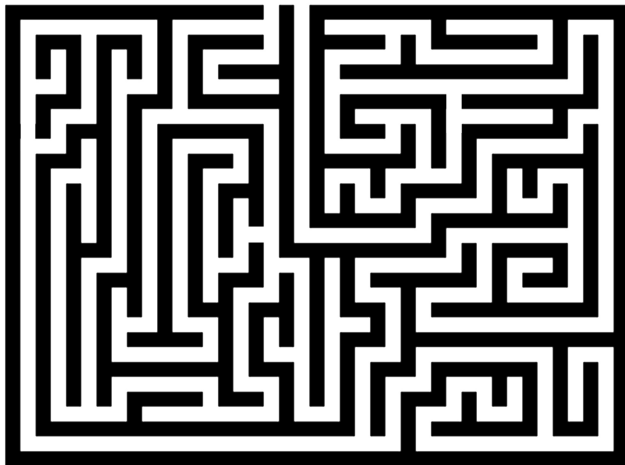
Fronta: ukázka deque

```
from collections import deque
q = deque(["Eric", "John", "Michael"])
q.append("Terry")    # Terry arrives
q.append("Graham")  # Graham arrives
q.popleft()         # Eric leaves
q.popleft()         # John leaves
print(q)            # deque(['Michael', 'Terry', 'Graham'])
```

Datový typ množina

- neuspořádaná kolekce dat bez vícenásobných prvků
- operace
 - insert (vložení)
 - find (vyhledání prvku, test přítomnosti)
 - remove (odstranění)
- použití
 - grafové algoritmy (označení navštívených vrcholů)
 - rychlé vyhledávání
 - výpis unikátních slov

Motivace pro množinu



Množina v Pythonu

```
set(alist)           # vytvoří množinu ze seznamu
len(s)              # počet prvků množiny s
s.add(x)            # přidání prvku do množiny
s.remove(x)         # odebrání prvku z množiny
x in s              # test, zda množina obsahuje x
s1 <= s2            # test, zda je s1 podmnožinou s2
s1.union(s2)        # sjednocení množin s1 a s2
s1 | s2             # -- totéž --
s1.intersection(s2) # průnik množin s1 a s2
s1 & s2             # -- totéž --
s1.difference(s2)   # rozdíl množin s1 a s1
s1 - s2             # -- totéž --
s1.symmetric_difference(s2) # symetrický rozdíl množin
s1 ^ s2             # -- totéž --
```

Množina v Pythonu

```
basket = ['apple', 'orange', 'apple', 'orange', 'banana']
fruit = set(basket)
print(fruit)           # {'orange', 'apple', 'banana'}
print('orange' in fruit) # True
print('tomato' in fruit) # False
```

```
a = set("abracadabra")
b = set("engineering")
print(a)           # {'a', 'r', 'b', 'c', 'd'}
print(b)           # {'i', 'r', 'e', 'g', 'n'}
print(a | b)       # {'a', 'c', 'b', 'e', 'd', 'g', 'i', 'r'}
print(a & b)       # {'r'}
print(a - b)       # {'a', 'c', 'b', 'd'}
print(a ^ b)       # {'a', 'c', 'b', 'e', 'd', 'g', 'i', 'r'}
```

Aplikace množiny

```
def unique(alist):  
    return list(set(alist))
```

```
def odd_one_out(alist):  
    return set(alist)-set([x for x in alist  
                           if alist.count(x) > 1])
```

dictionary, map, asociativní pole

- neuspořádaná množina dvojic (klíč, hodnota)
- klíče jsou unikátní
- operace jako u množiny (insert, find, delete)
- přístup k hodnotě pomocí klíče (indexování pomocí [])
- klíče jsou neměnné, ale hodnoty se smí měnit
- příklady použití
 - překlad UČO na jméno, jméno na tel. číslo
 - počet výskytů slov v textu
 - „cache“ výsledků náročných výpočtů

zjednodušené srovnání:

- indexování (klíče):
 - seznam: přirozená čísla
 - slovník: čísla (i desetinná), řetězce, n-tice, ... (cokoliv neměnitelného)
- pořadí klíčů:
 - seznam: fixně dáno
 - slovník: neřazeno

Slovník v Pythonu

```
phone = {"Buffy": 5550101, "Xander": 5550168}
phone["Dawn"] = 5550193
print(phone)
# {'Xander': 5550168, 'Dawn': 5550193, 'Buffy': 5550101}
print(phone["Xander"])
# 5550168
del phone["Buffy"]
print(phone)
# {'Xander': 5550168, 'Dawn': 5550193}
print(phone.keys())
# dict_keys(['Xander', 'Dawn'])
print("Dawn" in phone)
# True
```

užitečné funkce pro slovníky

```
d.keys()           # vrátí seznam klíčů
d.values()        # vrátí seznam hodnot
d.items()         # vrátí seznam záznamů (dvojic)
d.get(key, default) # pokud existuje klíč key, vrátí
                   # jeho hodnotu, jinak vrátí
                   # hodnotu default
d.get(key)        # jako předtím,
                   # jen default je teď None
```


.items() – použití např. pro kompletní výpis (nikoliv vyhledávání!)

```
for name, num in phone.items():  
    print(name + "'s number is", num)  
# Xander's number is 5550168  
# Dawn's number is 5550193
```

Odd one out: řešení se slovníkem

```
def odd_one_out(alist):  
    count = {}  
    for x in alist:  
        count[x] = count.get(x, 0) + 1  
    for x in count:  
        if count[x] == 1:  
            return x  
    return None
```

Odd one out: řešení se slovníkem II

```
def odd_one_out(alist):  
    count = {}  
    for x in alist:  
        count[x] = count.get(x, 0) + 1  
    return min(count.keys(), key=count.get)
```

Pozn. Odlišné chování od ostatních ukázek v případě, kdy není splněn předpoklad unikátního osamocenému prvku.

Převod do morseovky

připomenutí: dřívější řešení přes seznamy

```
morse = [".-.", "-....", "-.-.", "-.."] # etc
```

```
def to_morse(text):  
    result = ""  
    for i in range(len(text)):  
        if ord("A") <= ord(text[i]) <= ord("Z"):  
            c = ord(text[i]) - ord("A")  
            result += morse[c] + "|"  
    return result
```

Převod do morseovky

```
morse = {"A": ".-", "B": "-...", "C": "-.-."} # etc.
```

```
def to_morse(text):  
    result = ""  
    for c in text:  
        result += morse.get(c, "?") + "|"  
    return result
```

pokročilá verze

```
def to_morse(text):  
    return("|".join(list(map(lambda x: morse.get(x, "?"),  
                             text))))
```

Substituční šifra

```
def encrypt(text, subst):
    result = ""
    for c in text:
        if c in subst:
            result += subst[c]
        else:
            result += c
    return result

my_cipher = {"A": "Q", "B": "W", "C": "E"} # etc.

print(encrypt("BAC", my_cipher))
# WQE
```

Frekvenční analýza písmen

dříve uvedené řešení:

```
def frequency_analysis(text):
    text = text.upper()
    frequency = [0 for i in range(26)]
    for letter in text:
        if ord(letter) >= ord('A') and\
            ord(letter) <= ord('Z'):
            frequency[ord(letter) - ord('A')] += 1
    for i in range(26):
        if frequency[i] != 0:
            print(chr(ord('A')+i), frequency[i])
```

⇒ přepište pomocí slovníku

Frekvenční analýza slov

```
text = """It is a period of civil war. Rebel  
spaceships, striking [...] restore freedom to  
the galaxy """
```

```
output_word_freq(text)
```

```
the      7  
to       4  
rebel    2  
plans    2  
of       2  
her      2  
...
```


Frekvenční analýza slov

```
def is_word_char(char):  
    return char not in '!"$%&\'()*+,-./:;<=>?@[\\]^_`~' +  
  
def word_freq(text):  
    text = "".join(filter(is_word_char, text))  
    text = text.lower()  
    word_list = text.split()  
    freq = {}  
    for word in word_list:  
        freq[word] = freq.get(word, 0) + 1  
    return freq
```

```
def output_word_freq_simple(text):  
    freq = word_freq(text)  
    for word in freq:  
        print(word, "\t", freq[word])
```

Jak udělat výpis seřazený podle četnosti a vypisovat pouze nejčetnější slova?

Řazení výstupu: pomocný seznam dvojic

```
def output_word_freq(text, topN=10):  
    freq = word_freq(text)  
    tmp = [(freq[word], word) for word in freq]  
    tmp.sort(reverse=True)  
    for count, word in tmp[:topN]:  
        print(word, "\t", count)
```

Řazení výstupu: využití lambda funkce

```
def output_word_freq(text, topN=10):  
    freq = word_freq(text)  
    words = sorted(freq, key=lambda x: -freq[x])  
    for word in words[:topN]:  
        print(word, "\t", freq[word])
```

slovník můžeme indexovat „neměnitelnými“ datovými typy

- čísla
- řetězce
- dvojice
- n-tice (neměnná forma seznamu)

Indexování slovníku n-ticí

```
data = {}  
data[(1, 5)] = "white king"  
data[2, 6] = "black rook"  
print(data.keys()) #dict_keys([(1, 5), (2, 6)])
```

Můžeme vynechávat kulaté závorky u n-tice.

Pozor na rozdíl:

- `data[x][y]` – seznam seznamů
- `data[x, y]` – slovník indexovaný dvojicí

Kontrolní otázky

- Co znamenají pojmy abstraktní datový typ a konkrétní datová struktura?
- Jaké operace podporují datové typy fronta a zásobník? Jaký je mezi nimi rozdíl? K čemu je využíváme?
- K čemu slouží datový typ slovník? Jak používáme slovník v Pythonu?
- Jaký je rozdíl mezi seznamem a n-ticí?
- Co to je postfixová notace? Za využití které datové struktury ji vyhodnocujeme?
- Jakým způsobem můžeme reprezentovat dvourozměrnou mřížku (například šachovnici)?
- Jak můžeme vypsát všechny unikátní prvky v seznamu? Uveďte alespoň tři různá řešení (nápověda: množina, slovník, `count()`).
- Jaký je rozdíl mezi `data[x][y]` a `data[x,y]`?

Doporučené procvičování

`https://www.umimeprogramovat.cz/rozhodovacka`

`https://www.umimeprogramovat.cz/vystup-programu`

⇒ sada „Slovníky“

- datové typy: abstraktní vs. konkrétní
- seznam, vnořený seznam („vícerozměrné pole“)
- zásobník, fronta
- množina
- slovník

příště: rekapitulace, řešené (rozsáhlejší) příklady