4. Objekty. Abstraktní datový typ

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Objekty
- Část 2 – Abstraktní datové typy
  - Seznam
  - Zásobník
  - Fronta
  - Množina

Část I

Objekty

Záznam (Record)

Záznam (obecně)

- strukturovaný/složený datový typ
- obsahuje položky *(fields)* / prvky *(elements)*/ členy
- položek je (obvykle) pevný počet, mají každá svůj typ
- položky jsou identifikované jménem
- typ záznamu má své jméno

Příklady

- Datum = rok, měsíc, den
- Osoba = jméno, příjmení, datum narození
- Adresa = jméno ulice, číslo popisné, město, PSČ
- Bod = x, y
- Kruh = střed, poloměr

Abstrakce

- **funkční abstrakce** *(function abstraction)*
  - Klient ví, co funkce dělá. *(rozhraní, interface)*
  - Klient nemusí vědět, jak je funkce implementována.
- **datová abstrakce** *(data abstraction)*
  - Klient ví, co datový typ představuje, jak ho vytvořit a jaké operace podporuje.
  - Klient nemusí vědět, jaká je vnitřní struktura.
- Klient musí znát rozhraní *(interface)*.
- Implementace je skrytá, lze ji změnit.
- Implementace je rozdělena mezi klientský kód a knihovny (např. ve formě modulů).
- Znovupoužitelnost kódu, možnost nezávislého vývoje.
- Zjednodušení psaní klientského kódu.

Záznam v Pythonu

V Pythonu záznamy nejsou. . . , ale jsou (dynamické) **objekty**.

objekt = *záznam* + *metody*

Objekt

- Obsahuje **datové položky** / **atributy** / **proměnné**
- Může obsahovat **metody** – funkce pracující s datovými položkami
- Strukturu definuje **třída**, **objekty** jsou instance třídy.
- Základ **objektově orientovaného programování – OOP)**.

Varování

Toto není objektově orientované programování.

OOP je významně komplexnější, než je prezentováno v tomto předmětu.

Slovníček pojmů

třída	obecný uživatelsky definový typ, charakterizován atributy
objekt	konktétní instance třídy
atribut	vlastnost konkrétního objektu
metoda	funkce, která je vázaná na danou třídu
konstruktor	inicializační metoda, vytváří objekt

Intuitivní ilustrace

- třída: Pes
- instance: Alík
- datové atributy: rasa, jméno, věk, poloha
- metody: štěkej, sedni, lehni, popoběhni

Objekty v Pythonu

- Definice třídy:

```
1  class Osoba:
2    pass # prázdná třída
```

- Vytvoření objektu:

```
o=Osoba()
```

- Přidání a použití datových položek (atributů):

```
1  o.jmeno="Karel"
2  o.prijmeni="Novak"
4  print(o.jmeno+" "+o.prijmeni)
```

```
Karel Novák
```

- V Pythonu jsou datové položky přidávané *dynamicky.*
- K datovým položkám přistupujeme pomocí *tečkové notace.*

Metody

- Metody jsou funkce definované uvnitř třídy.

```
1  class Osoba:
2    def print(self):
3      print(self.jmeno+" "+self.prijmeni)
5  o=Osoba()
6  o.jmeno="Karel"
7  o.prijmeni="Novak"
```

- Pracují s konkrétním objektem.
- Mají přístup k proměnným objektu pomocí prvního parametru.

self není klíčovým slovem, jen silnou konvencí.

- Voláme je na objekt tečkovou notací.

```
1  o.print()
```

```
Karel Novak
```

Vytvoření objektu

- Objekt s konstruktorem:

```
class Osoba:
  def __init__(self,jmeno,prijmeni):
  self.jmeno=jmeno
  self.prijmeni=prijmeni

  def print(self):
    print(self.jmeno+" "+self.prijmeni)
```

- Konstruktor má speciální jméno `__init__`.
- Je volán při vytvoření objektu.

```
o=Osoba("Karel","Novak")
o.print()
```

```
Karel Novak
```

Přístup k atributům

Přímý

- přistupujeme přímo k atributu, můžeme ho přímo měnit
- `person.name`

Nepřímý

- pomocí metod (getters and setters)
- `person.get_name()`, `person.set_name()`

Který zvolit?

- závisí na použití
- objekt jen pro udržení dat – přímý přístup OK
- skrytí implementace (zapouzdření) – metody

Přístup k atributům

Přímý

- přistupujeme přímo k atributu, můžeme ho přímo měnit
- `person.name`

Nepřímý

- pomocí metod (getters and setters)
- `person.get_name()`, `person.set_name()`

Který zvolit?

- závisí na použití
- objekt jen pro udržení dat – přímý přístup OK
- skrytí implementace (zapouzdření) – metody

Příklad – body ve 2D prostoru 1/2

- definice třídy

```
1  class Point:
2    def __init__(self,x,y):
3      self.x=x; self.y=y
```

- inicializace proměnných

```
1  r = Point(10,20)
2  s = Point(13,24)
3  print("r.x=",r.x)
```

```
r.x = 10
```

- změna atributu

```
1  s.y =20
2  print("s.y =",s.y)
```

```
s.y = 20
```

Příklad – body ve 2D prostoru 2/2

- Funkce s argumenty datového typu `Point`:

```
1  import math
3  def distance(r,s):
4    """ Vypocita vzdalenost dvou bodu 'r','s' typu Point """
5    return math.sqrt((r.x-s.x)**2+(r.y-s.y)**2)
7  r=Point(10,20)
8  s=Point(13,24)
9  print(distance(r,s))
```

```
5.0
```

Příklad – knihy 1/3

- práce se seznamem knih
- atributy knihy: název, autor, rok
- chceme seznam načítat/ukládat do souboru
- implementace třídy:

```
class Book:
  def __init__(self, title, author, year):
    self.title = title
    self.author = author
    self.year = year
```

- vytvoření záznamů

```
a = Book ("Dva roky prazdnin", "Jules Verne", 1888)
b = Book ("Honzikova cesta", "Bohumil Riha", 1954)
```

Příklad – knihy 2/3

- načítání ze souboru

```
def load_library(filename):
  book_list = []
  with open(filename, "r") as f:
    for line in f:
      a, t, i = line.split(";")
      book_list.append(Book(t, a, i))
  return book_list
```

- ukládání do souboru

```
def save_library(filename, book_list):
  with open(filename, "w") as f:
    for b in book_list:
      f.write(b.title + ";" + b.author + ";"
      + b.year + "\n")
```

Příklad – knihy 3/3

- použití

```
save_library("library.csv", [a, b])
books = load_library("library.csv")
for b in books: print(b.title)
```

Reklamní vstup

CSV jsou super, ty chcete!

- CSV = Comma-separated values
- formát pro reprezentaci tabulkových dat
- jednoduchý textový formát, položky odděleny čárkami (nebo podobnými znaky)

Příklad – studenti a kurzy 1/3

- reprezentace studentů a kurzů
- kurz je seznamem zapsaných studentů
- implementace třídy `Student`

```
1  class Student:
2    def __init__(self, id, name):
3      self.id = id
4      self.name = name
```

Příklad – studenti a kurzy 2/3

```
1  class Course:
2    """ list of the students """
3    def __init__(self, code):
4      """ constructor """
5      self.code = code
6      self.students = []
8    def add_student(self, student):
9      """ add student to the list """
10     self.students.append(student)
11
12   def print_students(self):
13     """ print student enrolled to the course """
14     i = 1
15     for s in self.students:
16       print(str(i) + ".", str(s.id), s.name, sep="\t")
17       i += 1
```

Příklad – studenti a kurzy

- použití

```
jimmy = Student(555007, "James Bond")
c = Course("ZPR")
c.add_student(Student(555000, "Luke Skywalker"))
c.add_student(jimmy)
c.add_student(Student(555555, "Bart Simpson"))
c.print_students()
```

```
# 1. 555000 Luke Skywalker
# 2. 555007 James Bond
# 3. 555555 Bart Simpson
```

- co kdybychom chtěli řadit?
  - `sorted(self.students)` nefunguje – není definováno
  - definovat `__lt__(self, other)`
  - `sorted(self.students, key=lambda s: s.name)`

Část II

Abstraktní datové typy

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

Konkrétní datová struktura

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

Dva pohledy na data

Abstraktní

- operace, které budu s daty provádět
- co musí operace splňovat
- například množina: ulož, najdi, vymaž
- tento předmět

Výhody: snadný vývoj, jednodušší přemýšlení o problémech

Riziko: svádí k ignorování efektivity

Implementační

- jak jsou data uložena v paměti
- jak jsou operace implementovány
- například binární vyhledávací strom
- navazující předmět :-)

Nejpoužívanější ADT

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

II. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Různé varianty seznamu

- 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 prázdnosti, délky
- případně další operace, např. přístup pomocí indexu

Seznamy v Pythonu – operace

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

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

Seznamy v Pythonu – operace

```
s = [4, 1, 3]  # seznam
s.append(x)    # prida prvek x na konec
s.extend(t)    # prida vsechny prvky t na konec
s.insert(i, x) # prida prvek x pred prvek na pozici i
s.remove(x)    # odstrani prvni prvek rovny x
s.pop(i)       # odstrani (a vrati) prvek na pozici i
s.pop()        # odstrani (a vrati) posledni prvek
s.index(x)     # vrati index prvniho prvku rovneho x
s.count(x)     # vrati pocet vyskytu prvku rovnych x
s.sort()       # seradi seznam
s.reverse()    # obrat seznam
x in s         # test, zda seznam obsahuje x
# (linearni pruchod seznamem!)
```

Seznamy v Pythonu – generátorová notace

- Specialita Pythonu

```
1  s = [x for x in range(1, 7)]
2  print(s)
```

```
[1, 2, 3, 4, 5, 6]
```

```
1  s = [2 * x for x in range(1, 7)]
2  print(s)
```

```
[2, 4, 6, 8, 10, 12]
```

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

```
[(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B')]
```

II. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Zásobník

- strukturovaný/složený datový typ
- obsahuje předem neznámé množství položek, typicky stejného typu *(jako seznam/pole)*
- podporuje následující operace
  - přidání položky na konec – `push`
  - odebrání položky z konce – `pop`
  - test, jestli je zásobník prázdný – `is_empty`
- položky jsou odebírány v opačném pořadí, než byly přidány.
  LIFO – last in first out
- zásobník může podporovat i další operace
  - nedestruktivní čtení z konce – `peek`, `top`
  - zjištění počtu položek na zásobníku – `size`

Zásobník – příklad použití

```
1  from stack import Stack
3  s=Stack()
5  s.push(1)
6  s.push(2)
7  s.push(3)
9  print(s.pop())
10 print(s.pop())
12 s.push(10)
14 print(s.pop())
15 print(s.is_empty())
16 print(s.pop())
17 print(s.is_empty())
```

Zásobník – implementace pomocí pole

```
1  class Stack:
2    def __init__(self):
3      self.items = []
5    def size(self):
6      return len(self.items)
8    def is_empty(self):
9      return self.size()==0
11   def push(self, item):
12     self.items+=[item]
14   def pop(self):
15     return self.items.pop()
17   def peek(self):
18     return self.items[-1]
```

Příklad – převod do jiné číselné soustavy

```
1  from stack import Stack
3  def to_str(n,base):
4    cislice = "0123456789ABCDEF"
5    assert(n>=0)
6    stack=Stack()
7    while True:
8      stack.push(n % base)
9      n //= base
10     if n==0: break
11   result=""
12   while not stack.is_empty():
13     result+=cislice[stack.pop()]
14   return result
16 print(to_str(67,2))
```

II. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Fronta (Queue)

- strukturovaný/složený datový typ
- obsahuje předem neznámé množství položek, typicky stejného typu *(jako pole)*
- podporuje následující operace
  - přidání položky na konec *(enqueue, add)*
  - odebrání položky **ze začátku** *(dequeue, top)*
  - test, jestli je fronta prázdná *(is_empty)*
- položky jsou odebírány ve **stejném** pořadí, jako byly přidány. *(FIFO — first in first out)*
- fronta může podporovat i další operace
  - nedestruktivní čtení ze začátku *(peek)*
  - zjištění počtu položek ve frontě *(size)*
- Pokročilejší varianta: prioritní fronta

Fronta – implementace v Pythonu

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

```
1 from collections import deque
2 q = deque(["Eric", "John", "Michael"])
3 q.append("Terry") # Terry arrives
4 q.append("Graham") # Graham arrives
5 q.popleft() # Eric leaves
6 q.popleft() # John leaves
7 print(q) # deque(['Michael', 'Terry', 'Graham']
```

Fronta – implementace v Pythonu

- Příklad vlastní implementace

```
1  from slowqueue import Queue
3  q=Queue()
4  q.enqueue(1)
5  q.enqueue(2)
6  q.enqueue(3)
7  print(q.dequeue())
8  print(q.dequeue())
9  q.enqueue(10)
10 print(q.dequeue())
11 print(q.is_empty())
12 print(q.dequeue())
13 print(q.is_empty())
```

lec04/queue_examples.py

Fronta – použití

- Komunikace mezi procesy *(consumer,producer)*
- Čekání na asynchronní periferie — klávesnice, disk, síť ...
- 'Férový přístup' pro sdílení zdrojů *(policy)*
- Simulace čekání ve frontě
- Některé grafové a třídící algoritmy *(prohledávání do šířky, merge sort)*

Fronta – implementace pomocí seznamu

```
1  class Queue:
3    def __init__(self):
4      self.items = []
6    def is_empty(self):
7      return self.items == []
9    def enqueue(self, item):
10     self.items.insert(0,item)
12   def dequeue(self):
13     return self.items.pop()
15   def size(self):
16     return len(self.items)
```

lec04/slowqueue.py

Fronta – seznam s realokací 1/2

- Prvky ukládáme do seznamu.
- Vkládání na konec seznamu
- Prvky nemažeme, ale posouváme index počátku.
- Pokud je vynechaných prvků hodně, překopírujeme frontu do nového pole.
- Kopírujeme po poklesu využití paměti na 50 %
- Odebrání n prvků vyžaduje $\sim \log_2 n$ kopírování, dohromady $n/2 + n/4 + \cdots \sim n$ prvků prvku je $O(1)$.
- Nevýhoda – neefektivní využití paměti.

`lec04/arrayqueue.py`

Fronta – pole s realokací 2/2

```
1  class Queue:
3    def __init__(self):
4      self.front = 0   # index prvního prvku
5      self.items = []
7    def is_empty(self): return len(self.items) == self.front
9    def enqueue(self, item): self.items+=[item]
11   def dequeue(self):
12     el = self.items[self.front]
13     self.front+=1
14     if self.front >= 1024 and self.front >= len(self.items)//2:
15        self.items = self.items[self.front:]
16        self.front = 0
17     return el
19   def size(self): return len(self.items)
```

lec04/arrayqueue.py

Fronta – dva zásobníky (Donald Knuth) 1/2

- Prvky ukládáme do zásobníku `inp`
- Prvky odebíráme ze zásobníku `out`
- Když je `out` prázdný, přesuneme do něj `inp`
- Každý prvek je kopírován jen jednou

Fronta – dva zásobníky 2/2

```
1  class Queue:
3    def __init__(self):
4      self.inp = Stack()
5      self.out = Stack()
7    def is_empty(self):
8      return self.size()==0
10   def enqueue(self, item):
11     self.inp.push(item)
13   def dequeue(self):
14     if self.out.is_empty():
15       while not self.inp.is_empty():
16         self.out.push(self.inp.pop())
17     return self.out.pop()
19   def size(self):
20     return self.inp.size() + self.out.size()
```

lec04/knuthqueue.py

II. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Množina

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

Množina v Pythonu

```
set(alist) # vytvori mnozinu ze seznamu
len(s) # pocet prvku mnoziny s
s.add(x) # pridani prvku do mnoziny
s.remove(x) # odebrani prvku z mnoziny
x in s # test, zda mnozina obsahuje x
s1 <= s2 # test, zda je s1 podmnozinou s2
s1.union(s2) # sjednoceni mnozin s1 a s2
s1 | s2 # -- totez --
s1.intersection(s2) # prunik mnozin s1 a s2
s1 & s2 # -- totez --
s1.difference(s2) # rozdil mnozin s1 a s1
s1 - s2 # -- totez --
s1.symmetric_diference(s2) # symetricky rozdil mnozin
s1 ^ s2 # -- totez --
```

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', 'n',
print(a & b) # 'r'
print(a - b) # 'a', 'c', 'b', 'd'
print(a ^ b) # 'a', 'c', 'b', 'e', 'd', 'g', 'i', 'n'
```