Vlastní datové typy

IB111 Základy programování

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

31. října 2025

Vlastní datové typy

Motivace

Udržování dat pohromadě

- Příklad – chceme uchovávat záznamy o knihách:
  - název knihy, autor, ISBN apod.

Jak reprezentovat?

- Ntice (tuples):
  - nepojmenované složky, neměnné (až na hodnoty jejich složek v podobě měnitelných objektů jako např. seznamy).
- Seznamy:
  - měnitelné, nepojmenované (je název první nebo autor první?),
  - nevhodné pro položky různých typů (jak otypovat?).
- Slovníky:
  - měnitelné, pojmenované, nevhodné pro položky různých typů,
  - nelze spoléhat na to, že je zachována určitá struktura a připravit si operace spoléhající na tuto strukturu.

Jde to i jinak?

Motivace

Schovávání škaredých detailů

- Příklad – vzpomeňte si na dvojrozměrné matice:
  - reprezentovány pomocí seznamu seznamů,
  - nepěkný způsob zjišťování velikosti matice,
  - problematické udržování konzistence.

Co bychom chtěli?

- Spolu s maticí si udržovat informace o její velikosti.
- Mít něco, co kontroluje přístupy do matice.

Motivace

Vlastní datové struktury

- Co kdybychom chtěli mít nějakou vlastní datovou strukturu?
- Např. zřetězený seznam, nějaký druh stromové struktury, ...

Vlastní specializované operace – metody

- Už jsme viděli u seznamů, slovníků apod.

```
s = [7, 14, 42, 0]
s.append(9)

d = {"a": 1, "b": 2}
c = d.get("c")
```

- Co znamenají, jak se liší od volání funkcí?

Odbočka: Součinové a součtové typy

Součinové typy $A \times B$

- Hodnota má 2 složky: jedna je typu A, druhá typu B.
  - Podobně pro součin více typů.
- V Pythonu: ntice, *jednoduché třídy* (uvidíme za chvíli).
- Jiné jazyky: ntice, záznamy (`record`, `struct`),
  v Haskellu typové konstruktory s více parametry.
- *Reprezentuje kartézský součin množin hodnot.*

Součtové typy $A + B$

- Hodnota je buď typu A nebo typu B.
  - Podobně pro součet více typů.
- V Pythonu: *typové anotace s* `|` (další slajd).
- Jiné jazyky: (tagged) union, variant,
  v Haskellu datové typy s více typovými konstruktory.
- *(Disjunktní) sjednocení.*

Typová anotace – sjednocení typů

budeme používat
až od 9. kapitoly

`X | Y`

- Buď hodnota typu `X` nebo typu `Y`.
- Klasické sjednocení, ne disjunktní (rozdíl proti jiným jazykům).
- Možno i více typů: `X | Y | Z` apod.

Zjištění skutečného typu

- Predikát `isinstance(výraz, typ)`.

Použití s ‘mypy‘

- Podobně jako u `X | None`, mypy umí odvodit typ hodnoty uvnitř větví příkazu `if` a za příkazem `assert` –
  - při použití `isinstance`.

Vlastní součinové typy

Záznamy, struktury

- Datový typ složený z více pojmenovaných položek.
- Typicky fixní počet.
- Deklarované typy – mohou se u různých položek lišit.
- C: `struct`.
- Pascal: `record`.

Třídy

- Rozšíření struktur.
- Kombinují data a funkce (metody).
- C++, Java, Python, ...: `class`.
- Mnohem komplikovanější, než si ukážeme:
  - dědičnost, virtuální volání (realizace polymorfismu), ...

Vlastní datové typy v Pythonu

Varování

Pozor!

- Toto není objektově orientované programování (OOP).
- OOP je *podstatně* složitější.
- Zde používáme třídy jen *velmi jednoduchým* způsobem,
  - náhrada za záznamy (struktury).

Vlastní datové typy v Pythonu

Objekty vlastních datových typů (tříd) se skládají z **atributů**.

- Něco jako položky ntice, ale
  - jsou pojmenované,
  - dá se do nich přiřadit – *vnitřní přiřazení*.
- Chovají se podobně jako proměnné, resp. položky seznamů.
- Přístup k atributu pomocí tečky: `výraz.jméno_atributu`,
  - výraz se musí vyhodnotit na objekt správného typu (typicky proměnná).

Vlastní datové typy v Pythonu

Třídy – definice nového typu.

```
class JménoTypu:
    """Dokumentace třídy."""

    def __init__(self, formální parametry, …) -> None:
        # inicializace

    # definice metod

    def metoda(self, formální parametry, …) -> typ:
        # tělo metody

    # ...
```

Vznik nového objektu vlastního typu

person.py

Inicializační funkce `__init__`

- Povinný formální parametr `self`, příp. další form. parametry.
  - `self` odkazuje na právě vznikající objekt.
- Cílem inicializace je *vytvořit atributy* (pomocí přiřazení).

Vytvoření objektu vlastního typu

- Jméno třídy se volá jako funkce.
- Skutečné parametry volání se sváží s dalšími formálními parametry `__init__`.

```
class Person:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

homer = Person("Homer Simpson", 34)
```

Metody

person.py

Definice metody

- Funkce definovaná uvnitř třídy.
- Povinný formální parametr `self`, příp. další form. parametry.
  - `self` odkazuje na *aktuální objekt*.

Volání metody

- `výraz.jméno_metody(další skutečné parametry, …)`.
- `výraz` se vyhodnotí na objekt,
  - typ objektu určí, ve které třídě se hledá `jméno_metody`,
  - objekt se předá v parametru `self`.

```
class Person:
    def say_hello(self) -> None:
        print(self.name, "says hello.")


homer.say_hello()
```

Metody

Čistá metoda

- Metoda, která je zároveň čistou funkcí.
- Nemá vedlejší efekty:
  - zejména ani nemodifikuje *aktuální objekt*.
- Při různých voláních se stejnými skutečnými parametry vrací stejné hodnoty,
  - se stejným aktuálním objektem ve stejném stavu.

Metoda-predikát

- Metoda, která je zároveň predikátem.
- Čistá metoda vracející `bool`.

Třídy a typové anotace

Jméno třídy je typ.

- Dá se použít jako typová anotace
  - za definicí třídy,
  - v *těle* inicializační funkce a metod třídy.
- V hlavičkách metod nebo před definicí třídy musíme použít tzv. *dopřednou referenci na typ*:
  - jméno typu jako řetězec,
  - při použití `|` musí být jako řetězec celé sjednocení.

Typové anotace parametrů a návratových hodnot

- `self` nemusí mít typovou anotaci.
- `__init__` nemusí mít typovou anotaci návratové hodnoty,
  - ale je pak třeba mít alespoň jeden anotovaný parametr.

Příklady vlastních datových typů

Knihovna

- Chceme si pamatovat seznam knih.
- Kniha má:
  - název,
  - autora,
  - ISBN.
- Načítání/ukládání dat z/do souborů – *pozdější přednáška*.

Příklad: Knihovna

library.py

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


# this is how we create books:
dune = Book("Frank Herbert", "Dune", "978-0441172719")
temno = Book("Bohuslav Balcar & Petr Štěpánek",
             "Teorie množin", "80-200-0470-X")
```

Příklady vlastních datových typů

Databáze studentů a předmětů

- Chceme jednoduchou databázi předmětů a studentů.
- Student má:
  - UČO a
  - jméno.
- Předmět má:
  - kód a
  - seznam studentů.

Příklad: Databáze studentů a předmětů

students.py

```
class Student:
    def __init__(self, uco: int, name: str):
        self.uco = uco
        self.name = name


class Course:
    def __init__(self, code: str):
        self.code = code
        self.students: list[Student] = []
```

Příklad: Databáze studentů a předmětů

students.py

```
class Course:
    # ...
    def add_student(self, student: Student) -> None:
        self.students.append(student)

    def get_student_names(self) -> list[str]:
        names = []
        for student in self.students:
            names.append(student.name)
        return names
```

Příklad: Databáze studentů a předmětů

students.py

```
jimmy = Student(555007, "James Bond")

ib111 = Course("IB111")

ib111.add_student(Student(555000, "Jan Novák"))
ib111.add_student(jimmy)
ib111.add_student(Student(999999, "Kryštof Harant"))
```

Příklady vlastních datových typů

matrix.py

Matice

- Chceme uchovávat i jejich velikost.
- Chceme bezpečný přístup k prvkům.

```
class Matrix:
    def __init__(self, rows: int, cols: int):
        self.rows = rows
        self.cols = cols
        self.matrix = [[0 for j in range(self.cols)]
                       for i in range(self.rows)]
```

Příklad: Matice

matrix.py

```
class Matrix:
    # ...
    def check(self, row: int, col: int) -> None:
        assert 0 <= row < self.rows
        assert 0 <= col < self.cols

    def get(self, row: int, col: int) -> int:
        self.check(row, col)
        return self.matrix[row][col]

    def set(self, row: int, col: int,
            value: int) -> None:
        self.check(row, col)
        self.matrix[row][col] = value
```

Příklad: Matice

matrix.py

```
def matrix_mult(left: Matrix, right: Matrix) -> Matrix:
    assert left.cols == right.rows, \
        "Incompatible matrices."

    result = Matrix(left.rows, right.cols)
    for i in range(left.rows):
        for j in range(right.cols):
            for k in range(left.cols):
                result.set(i, j, result.get(i, j) +
                           left.get(i, k) *
                           right.get(k, j))

    return result
```

Vlastní datová struktura

linkedlist.py

Jednosměrně zřetězený seznam


```
class Node:
    def __init__(self, data: str):
        self.data = data
        self.next: Node | None = None
```

Příklady: Jednosměrně zřetězený seznam

linkedlist.py

```
class LinkedList:
    def __init__(self) -> None:
        self.first: Node | None = None

    def add_to_beginning(self, data: str) -> None:
        node = Node(data)
        node.next = self.first
        self.first = node

    def delete_first(self) -> None:
        if self.first is not None:
            self.first = self.first.next
```

Příklady: Jednosměrně zřetězený seznam

linkedlist.py

```
class LinkedList:
    # ...
    def to_list(self) -> list[str]:
        result = []
        node = self.first
        while node is not None:
            result.append(node.data)
            node = node.next
        return result

my_ll = LinkedList()
my_ll.add_to_beginning("Hello")
my_ll.add_to_beginning("Ahoj")
elems1 = my_ll.to_list()
my_ll.delete_first()
elems2 = my_ll.to_list()
```

Vlastní datová struktura: zásobník

Všimněte si

- Pomocí zřetězeného seznamu můžeme implementovat zásobník.
- Vlastně už ho skoro máme ve třídě `LinkedList`:
  - *push*: `add_to_beginning`.
  - *pop*: `delete_first` až na to, že nevrací hodnotu.
  - *top*, *empty*: snadno doplnitelné.

Vlastní datová struktura: zásobník

stack.py

Zásobník (řetězců) pomocí zřetězeného seznamu.

```
class Stack:
    def __init__(self) -> None:
        self.top: Node | None = None

    def push(self, data: str) -> None:
        node = Node(data)
        node.next = self.top
        self.top = node

    def pop(self) -> str:
        assert self.top is not None
        result = self.top.data
        self.top = self.top.next
        return result
```

Vlastní datová struktura: fronta

Datový typ **fronta**

- Prvky v pořadí FIFO: *First In First Out*.
- Operace:
  - vložení (*enqueue*, někdy též *push*),
  - odebrání (*dequeue*, *pull* či *pop*),
  - náhled na první prvek,
  - test prázdnosti.
- Možné implementace?

Implementace fronty

Seznam s vkládáním na konec

- push: `q.append(…)`,
- pull: `q.pop(0)` –
  - lineární složitost (vůči délce fronty).

Seznam s vkládáním na začátek

- pull: `q.pop()`,
- push: `q.insert(0, …)` –
  - lineární složitost (vůči délce fronty).

Jiné možnosti?

- Ideálně s konstantním vkládáním i odebíráním prvků z fronty.

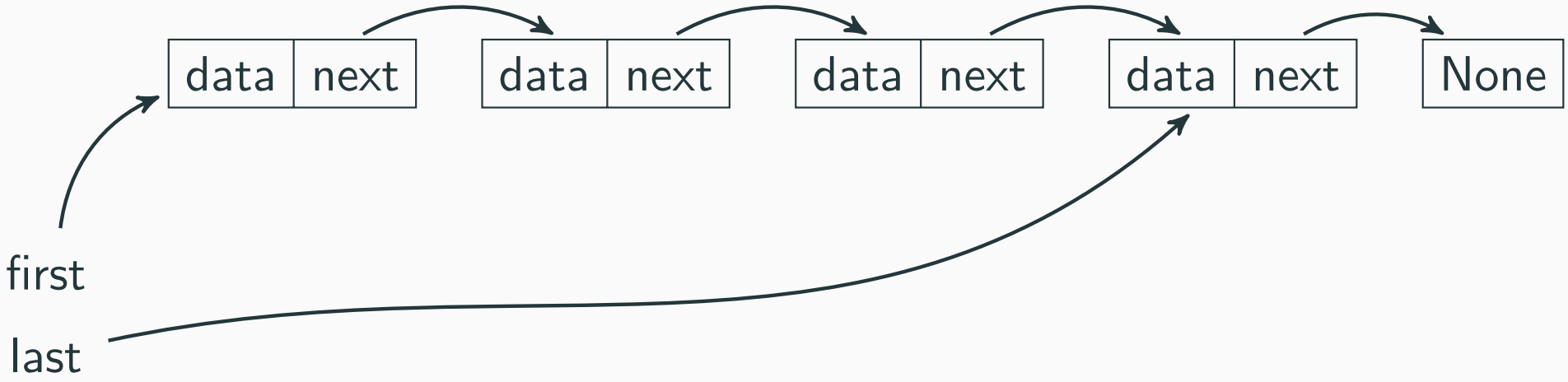
Implementace fronty

Obousměrně zřetězený seznam

- Ano, snadné přidávání/odebírání na obou koncích, ale zbytečně komplikované.

Jednosměrně zřetězený seznam

- Umíme přidávat a odebírat prvky na začátku.
- Umíme přidávat prvky na konec –
  - když si budeme pamatovat odkaz na poslední prvek.



Implementace fronty

queue1.py

```
class Queue:
    def __init__(self) -> None:
        self.first: Node | None = None
        self.last: Node | None = None

    def push(self, data: str) -> None:
        node = Node(data)
        if self.last is None:
            self.first = node
        else:
            self.last.next = node
        self.last = node
```

Odebrání prvku z fronty

- Je toto řešení korektní? NE!

```
class Queue:
    # ...

    def pull(self) -> str:
        assert self.first is not None
        result = self.first.data
        self.first = self.first.next
        return result
```

Odbočka: Invariant

Invariant

- V matematice: vlastnost struktury, která se nemění jejími transformacemi.

Invariant datové struktury

- Vlastnost *datové struktury*, která se nemění *voláními metod*,
  - tj. logická podmínka, která platí vždy mezi voláními metod.
  - (Nemusí platit uvnitř metod.)
- Součást vstupní **i** výstupní podmínky každé metody.
- Součást výstupní podmínky inicializační funkce `__init__`.

Jiné druhy invariantů (v pozdějších předmětech)

- Invariant funkce: součást vstupní i výstupní podmínky funkce.
- Invariant cyklu: platí při každém příchodu na hlavičku cyklu.

Invariant datové struktury

queue1.py

```
class Queue:
    def __init__(self) -> None:
        self.first: Node | None = None
        self.last: Node | None = None
```

Jaký invariant má (mj.) mít náš typ `Queue`?

- „*`first` je `None` právě tehdy, když `last` je `None`.*“

Je to skutečně invariant?

- `__init__`: OK, po skončení je podmínka splněna.
- `push`: OK, když platí podmínka na začátku, platí i na konci.
- `pull`: NE!
  - Má-li fronta jen jeden uzel, skončíme v situaci, kdy `first` je `None`, ale `last` se odkazuje na původní uzel.

Implementace fronty: znovu a lépe

queue2.py

```
def pull(self) -> str:
    assert self.first is not None
    result = self.first.data
    self.first = self.first.next
    if self.first is None:
        self.last = None
    return result
```

Statické atributy

- Definovány přímo ve třídě.
- Patří samotné třídě, ne objektům.
- V tomto předmětu **nepoužíváme**.

```
class MyClass:
    x = 0
    def __init__(self, n):
        self.y = n

print(MyClass.x)      # 0
my_object = MyClass(17)
print(my_object.y)    # 17
print(my_object.x)    # 0 (fskutečnosti MyClass.x)
```

Statické atributy: na co si dát pozor

hobbies.py

V čem je problém?

```
class Student:
    hobbies: list[str] = []
    def __init__(self, name: str):
        self.name = name
    def add_hobby(self, hobby: str) -> None:
        self.hobbies.append(hobby)

mirek = Student("Mirek Dušín")
mirek.add_hobby("pomáhání slabším")
mirek.add_hobby("světový mír")

bidlo = Student("Dlouhé Bidlo")
bidlo.add_hobby("alkohol")

print(mirek.hobbies)
```

Tečková notace

animals.py

Jaký je (mj.) rozdíl mezi běžnou funkcí a metodou objektu?

- V Pythonu nelze mít dvě funkce, které se jmenují stejně,
- ale různé objekty mohou mít stejně pojmenované metody.

```
class Dog:
    def make_sound(self) -> None:
        print("Whoof!")
class SmallFurryAnimalFromAlphaCentauri:
    def make_sound(self) -> None:
        print("Qwxrq!")

fido = Dog()
fido.make_sound()
frrq = SmallFurryAnimalFromAlphaCentauri()
frrq.make_sound()
```