České vysoké učení technické v Praze FIT

# Programování v Pythonu

Jiří Znamenáček

*Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.*

*Praha & EU: Investujeme do vaší budoucnosti*

# Python – Bajtové objekty

## Bajtové objekty jako „řetězce“ bajtů

**I.** Bajtový objekt je v podstatě „řetězec bajtů“. Jako takový jeho neproměnnou (*immutable*) variantu nejsnáze zavedeme pomocí podobné přímé notace jako řetězec:

```
>>> xb = b"ahoj"
>>> xb
b'ahoj'
>>> type(xb)
<class 'bytes'>
```

→ Jelikož různých znaků vyjádřitelných pomocí jednoho bajtu je právě 256 a znich pouze část spodní poloviny představuje tisknutelné znaky (stará dobrá ASCII-tabulka), setkáte se s bajtovými literály spíše v podobě hexadecimální, např. `b'\xf0\xf1\xf2'`.

**II.** Řetězce obsahují unicodové znaky, tj. sekvence bajtů, které daným kódováním určují odkazy do tabulky unicodových znaků:

```
>>> [x for x in 'ahoj']
['a', 'h', 'o', 'j']
```

Bajtové řetězce na druhou stranu obsahují pouze bajty, tj. čísla 0-255:

```
>>> [x for x in b'ahoj']
[97, 104, 111, 106]
```

**III.** S výjimkou metod *encode()*, *format()* a *isidentifier()* sdílejí bajtové řetězce s řetězci unicodovými stejné atributy:

```
>>> dir(xb)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getn
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshoo
'capitalize', 'center', 'count', 'decode', 'endswith', 'expandtabs
'fromhex', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'i
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketra
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase'
'translate', 'upper', 'zfill']
```

→ Většina z nich nám ale pro práci s bajtovými objekty je stejně k ničemu :-)

Stejně jako „obyčejné“ unicodové řetězce jsou řetězce bajtové neměnné (*immutable*):

```
>>> xb = b'ahoj'
>>> xb[1]
104
>>> xb[1] = 'H'
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    xb[1] = 'H'
TypeError: 'bytes' object does not support item assignment
```

# Bajtové řetězce jako sekvence

Stejně jako unicodové řetězce podporují bajtové řetězce tradiční sekvenční operace:

```
>>> xb = b'ahoj'

# délka sekvence
>>> len(xb)
4

# konkrétní prvek
>>> xb[3]
106
>>> xb[-3]
104

# různé výřezy
>>> xb[1:3]
b'ho'
>>> xb[1::2]
b'hj'
>>> xb[2:]
b'oj'
>>> xb[-3:]
b'hoj'

# dotaz na výskyt prvku
>>> 111 in xb
True
>>> 110 in xb
False
>>> b'a' in xb
True
>>> b'\xf1' in xb
False
>>> xb.index(b'h')
1
>>> xb.index(b'D')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> xb.count(b'a')
1

# dvě spojené kopie
>>> xb * 2
b'ahojahoj'
```

A samozřejmě máme k dispozici i oblíbenou smyčku *for-in* v obou jejích variantách:

```
>>> xb = b'ahoj'

>>> for x in xb:
...     print(x)
...
97
104
111
106

>>> for i, x in enumerate(xb):
...     print(i, x)
...
0 97
1 104
2 111
3 106
```

# Dekódování textových řetězců

Znakové a bajtové řetězce (*představující text*) mezi sebou můžeme pomocí zvolených kódování navzájem převádět:

```
>>> xs = '狼.cz'
>>> len(xs)
4

# ukázka tří různých způsobů bajtového zakódování téhož řetězce
>>> xs.encode('utf-8')
b'\xe7\x8b\xbc.cz'
>>> len( xs.encode('utf-8') )
6
>>> xs.encode('gb18030')
b'\xc0\xc7.cz'
>>> len( xs.encode('gb18030') )
5
>>> xs.encode('big5')
b'\xafT.cz'
>>> len( xs.encode('big5') )
5

# řetězec → bajtový řetězec → řetězec
>>> xs.encode('big5').decode('big5')
'狼.cz'
```

→ Častěji ale budou bajtové objekty představovat „skutečná“ binární data, např. obrázek nebo zvuk. Pak jsou samozřejmě výše uvedené konverze k ničemu.

# Bajtová pole

K bajtovým řetězcům existuje proměnný protějšek – *bajtová pole*. Jejich konstruktorem je `bytearray()`:

```
>>> xb = bytearray(b'ahoj')
>>> xb
bytearray(b'ahoj')

# ukázka měnitelnosti
>>> xb[1]
104
>>> xb[1] = 72
>>> xb
bytearray(b'aHoj')
```

# Bajtová pole jako sekvence

Bajtová pole podporují standardní operace, jaké bychom u proměnného typu čekali (v podstatě se chovají jako seznamy bajtů):

```
>>> xb = bytearray(b'ahoj')

# délka sekvence
>>> len(xb)
4

# konkrétní prvek
>>> xb[3]
106
>>> xb[-3]
104

# různé výřezy
>>> xb[1:3]
bytearray(b'ho')
>>> xb[1::2]
bytearray(b'hj')
>>> xb[2:]
bytearray(b'oj')
>>> xb[-3:]
bytearray(b'hoj')

# dotaz na výskyt prvku
>>> 111 in xb
True
>>> 110 in xb
False
>>> b'a' in xb
True
>>> b'\xf1' in xb
False
>>> xb.index(b'h')
1
>>> xb.index(b'D')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: subsection not found
>>> xb.count(b'a')
1

# dvě spojené kopie
>>> xb * 2
bytearray(b'ahojahoj')
```

Stejně tak máme k dispozici tradiční procházení smyčkou *for-in*:

```
>>> xb = bytearray(b'ahoj')

>>> for x in xb:
...     print(x)
...
97
104
111
106

>>> for i, x in enumerate(xb):
...     print(i, x)
...
0 97
1 104
2 111
3 106
```

# Bajtová pole – přidávání prvků

Prvky do bajtového pole přidáváme pomocí stejných metod jako u seznamů, tzn.:

**I.** Přidání jednoho bajtu na konec pomocí metody `append(BAJT)`:

```
>>> xb = bytearray(b'ahoj')

>>> xb.append(72)
>>> xb
bytearray(b'ahojH')
```

**II.** Vložení jednoho bajtu doprostřed pole pomocí metody `insert(POZICE_PRVKU_ZA, BAJT)`:

```
>>> xb = bytearray(b'ahoj')

>>> xb.insert(2, 72)
>>> xb
bytearray(b'ahHoj')
```

**III.** Rozšíření pole o dané bajty pomocí metody `extend(BAJTY)`:

```
>>> xb = bytearray(b'Ahoj')

>>> xb.extend(b', svete!')
>>> xb
bytearray(b'Ahoj, svete!')
>>> xb + b' Jak se mas?'
bytearray(b'Ahoj, svete! Jak se mas?')
```

→ Nenechte se zmást tím, že ukázky ukazuji na *cestine*. Je to jenom proto, že se to čte líp než nějaké sekvence netisknutelných bajtů.

# Bajtová pole – odebírání prvků

Prvky z bajtového pole odstraňujeme pomocí stejných metod jako u seznamů, tzn.:

**I.** Pomocí univerzálního příkazu `del`:

```
>>> xb = bytearray(b'Ahoj, svete! Jak se mas?')

>>> del xb[-1]
>>> xb
bytearray(b'Ahoj, svete! Jak se mas')

>>> del xb[4:]
>>> xb
bytearray(b'Ahoj')
```

**II.** Pomocí metody `pop([INDEX])`, která navíc vrací odebraný prvek:

```
>>> xb = bytearray(b'Ahoj, svete! Jak se mas?')

>>> xb.pop()
63
>>> xb
bytearray(b'Ahoj, svete! Jak se mas')

>>> xb.pop(0)
65
>>> xb
bytearray(b'hoj, svete! Jak se mas')
```

**III.** Pomocí metody `remove(PRVEK)`, která odstraní první výskyt (zleva) příslušného prvku:

```
>>> xb = bytearray(b'Ahoj, svete! Jak se mas?')

>>> xb.remove(101)
>>> xb
bytearray(b'Ahoj, svte! Jak se mas?')

>>> xb.remove(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: value not found in bytearray
```

→ Opět: Nenechte se zmást tím, že ukázky ukazuji na *cestine*. Je to jenom proto, že se to čte líp než nějaké sekvence netisknutelných bajtů.

# Binární soubory I

Binární soubor (povinně identifikovaný příznakem 'b') můžeme otevřít v několika módech:

- **br** – soubor je otevřen pouze pro čtení
- **bw** – soubor je otevřen pro zápis (již existující neprázdný soubor bude smazán)
- **ba** – soubor je otevřen pro přidávání (zapsaná data budou přidána na konec)
- **br+** – soubor je otevřen pro čtení i zápis
- **bw+** – soubor je otevřen pro čtení i zápis (již existující neprázdný soubor bude smazán)

Mějte zvláště na paměti následující rozdíly mezi binárními a textovými soubory:

1. *Binární* soubory jsou čteny (a zapisovány) **po bajtech**, *textové* jsou zpracovávány **po znacích** (přičemž jeden znak zabírá podle použitého kódování místo jednoho či několika bajtů).
2. V textových souborech je automaticky prováděna konverze konců řádků různých platforem (`\n` na Unixu, `\r\n` na Windows, `\r` na Mac OS) na jednotné pracovní `\n` (konverze je samozřejmě obousměrná).

# Binární soubory II

S binárními soubory se pracuje prakticky stejně jako se soubory textovými. Jen nemá žádný smysl uvádět kódování (protože se týká textových souborů) a také ukazatel aktuální pozice ve streamu `tell()` vždy souhlasí s počtem bajtů, které jsme už načetli.

```
with open('obrazek.png', mode='rb') as f:
    data = f.read()
```

Vybrané metody dostupné na binárním streamu:

- `read([N])` – načti všechny nebo uvedený počet bajtů (ukazatel do streamu se posouvá)
- `peek([N])` – načti jeden nebo uvedený počet bajtů (ukazatel do streamu se NEposouvá)
- `write(b)` – zapiš uvedený byte-objekt *b* a vrať počet skutečně zapsaných bajtů
- `seek(offset[, 0|1|2])` – přesuň se na daný bajt ve streamu; a to buď od začátku (výchozí hodnota), aktuální pozice nebo konce
- `tell()` – vrať aktuální pozici ve streamu
- `flush()` – vynuť si uložení dat do streamu před jeho zavřením
- `close()` – uzavři otevřený stream (čímž se uloží veškerá případně dosud neuložená data)
- …