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 – Poznámky k datovým typům

Základní datové typy – konstruktory

proměnné (*mutable*)	**neproměnné (*immutable*)**	**typ**
—	int()	číselný
—	float()	číselný
—	complex()	číselný
—	str(), '', ""	sekvenční
bytearray()	bytes(), b'', b""	sekvenční
list(), []	tuple(), (), ,	sekvenční
—	range()	sekvenční
set()	frozenset()	množinový
dict(), {}	—	mapovací

Přiřazení (či spíše „pojmenovávání“)

```
>>> xs = ['a', 'h', 'o', 'j']
>>> ys = xs
>>> ys
['a', 'h', 'o', 'j']

>>> xs.remove('o')
>>> xs
['a', 'h', 'j']
>>> ys
['a', 'h', 'j']
```

Proměnné (*mutable*) typy jsou v Python'u předávány odkazem, neproměnné (*immutable*) prozměnu zase hodnotou! Na začátku si na tom asi párkrát nabijete nos, ale časem na tuhle nakonec celkem logickou podivnost (zvanou, pokud vůbec nějak, *předávání objektem*) zvyknete.

S tím souvisí i rozdíl mezi *mělkou* a *hlubokou kopií* u složitějších objektů. Ukažme si to na slovnících a seznamech:

```
>>> xs = ['a', 'b', 'c']

# a) Zaveďme seznam a slovník, které obsahují podseznam..
>>> ys = [1, xs]
>>> zs = { 1: xs }
# ..a mělce je zkopírujme:
>>> ys2 = ys[:]
>>> zs2 = zs.copy()
# Výsledek je zatím nepřekvapivý:
>>> ys
[1, ['a', 'b', 'c']]
>>> ys2
[1, ['a', 'b', 'c']]
>>> zs
{1: ['a', 'b', 'c']}
>>> zs2
{1: ['a', 'b', 'c']}

# b) Odstraňme nyní z podseznamu poslední prvek:
>>> xs.pop()
'c'
>>> xs
['a', 'b']
# Nyní už se děje něco, co ne každý očekával:
>>> ys
[1, ['a', 'b']]
>>> ys2
[1, ['a', 'b']]
>>> zs
{1: ['a', 'b']}
>>> zs2
{1: ['a', 'b']}
```

Potřebujeme-li obejít uvedený problém, dává nám Python k dispozici metodu `code.deepcopy()`:

```
>>> xs = ['a', 'b', 'c']

# a) Zaveďme seznam a slovník, které obsahují podseznam..
>>> ys = [1, xs]
>>> zs = { 1: xs }
# ..a hluboce je zkopírujme:
>>> import copy
>>> ys2 = copy.deepcopy(ys)
>>> zs2 = copy.deepcopy(zs)
# Výsledek je zatím nepřekvapivý:
>>> ys
0: [1, ['a', 'b', 'c']]
>>> ys2
1: [1, ['a', 'b', 'c']]
>>> zs
2: {1: ['a', 'b', 'c']}
>>> zs2
3: {1: ['a', 'b', 'c']}

# b) Odstraňme nyní z podseznamu poslední prvek:
>>> xs.pop()
4: 'c'
>>> xs
5: ['a', 'b']
# Nyní už se to vše chová úplně jinak:
>>> ys
6: [1, ['a', 'b']]
>>> ys2
7: [1, ['a', 'b', 'c']]
>>> zs
8: {1: ['a', 'b']}
>>> zs2
9: {1: ['a', 'b', 'c']}
```

Hešovatelnost

Proměnné (*mutable*) a neproměnné (*immutable*) datové typy se od sebe liší tím, že druhé z nich představují objekty s fixní hodnotou (nemohou být tedy změněny), zatímco prvé nikoli. Prakticky to znamená, že **neproměnné typy jsou hešovatelné** (*hashable*), což obnáší:

1. dají se převést na svoji heš-hodnotu, která se během jejich existence nikdy nezmění
2. dají se porovnávat s dalšími objekty

   → Zatímco porovnatelnost zjišťují porovnávací operátory, na pozadí stojí „magická“ metoda `__eq__()`. Podobně hešovatelnost, kterou si můžeme otestovat pomocí

> vestavěné funkce `hash(OBJEKT)`, zajišťuje na pozadí metoda `__hash__()`.

Hešovatelnost tak umožňuje použít neproměnné typy jako prvky *množin* a klíče *slovníků*.

Několik poznámek:

- Heš-hodnoty jsou celá čísla.
- Rozdílné číselné typy stejné hodnoty mají samozřejmě stejnou heš (aby se porovnávaly stejně):

```
>>> x, y = 1, 1.0
>>> type(x)
<class 'int'>
>>> type(y)
<class 'float'>
>>> hash(x) == hash(y)
True
```

- Zdánlivě velmi podobná funkce `id(OBJEKT)` vrací „identifikátor“ objektu. Ten je po dobu života objektu jedinečný a konstantní, ale dva různé objekty existující v navzájem se nepřekrývajícím čase ho mohou mít stejný.
- Instance uživatelem definovaných tříd jsou z principu hešovatelné – porovnávají se jako nerovné a jejich heš-hodnotou je jejich *id()*.

Náročnost operací

Seznamy, n-tice a slovníky jsou implementovány jako heše, řetězce jako pole. To má mimo jiné následující vliv na náročnost operací prováděných nad příslušnými datovými strukturami:

- seznamy, n-tice, řetězce:
  1. náhodný přístup: O(1)
  2. vložení/smazání prvku, dotaz na prvek (`in`): O(n)
- slovníky:

1. dotaz na prvek (`in`), náhodný přístup: O(1)
2. vložení/smazání prvku: O(1)
3. neexistence lineárního uspořádání

→ Více viz http://www.cis.upenn.edu/~lhuang3/cse399-python/

Čísla

Vynecháme-li rozšiřující moduly, máme v základním Python'u k dispozici tři číselné typy s odpovídajícími konverzními funkcemi:

- `int()` – celá čísla (s libovolnou přesností)
- `float()` – reálná čísla (s přesností aktuální implementace)
- `complex()` – komplexní čísla (jako dvojice reálných čísel)

I. Celá a reálná čísla můžeme zapsat několika různými způsoby. U reálných je to ještě poměrně jednoduché..

```
>>> 3.14
3.14

>>> 3.14e2
314.0

>>> 3.14e-2
0.0314
```

..ale u celých je způsobů mnohem více:

```
# klasicky desítkově
>>> 26
26

# binárně
>>> 0b11010
26

# hexadecimálně
>>> 0x1a   # nebo 0x1A
26

# oktalově
>>> 0o32
26
```

U komplexních čísel není moc co řešit – písmenka `j`, resp. `J`, slouží k označení imaginární části komplexního čísla.

II. Celá a reálná čísla na sebe můžeme převádět, stejně jako se můžeme pokusit vyextrahovat číslo z řetězce:

```
>>> float(1)
1.0
>>> int(1.2)
1

>>> float('1.23')
1.23
>>> int('23')
23
```

→ S komplexními čísly to takhle z pochopitelných důvodů nejde.

III.

```

```

Řetězce vs bajtové objekty

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

```
>>> for x in 'ahoj':
...     print(x)
a
h
o
j
```

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

```
>>> for x in b'ahoj':
...     print(x)
97
104
111
106
```

II. Znakové a bajtové řetězce 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'
```

III. Zatímco řetězce a bajtové řetězce jsou neměnné, tak k bajtovým řetězcům existuje proměnný protějšek – bajtová pole:

```
>>> xs = 'ahoj'
>>> xs[1]
19: 'h'
>>> xs[1] = 'H'
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    xs[1] = 'H'
TypeError: 'str' object does not support item assignment

>>> xs = b'ahoj'
>>> xs[1]
20: 104
>>> xs[1] = 'H'
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    xs[1] = 'H'
TypeError: 'bytes' object does not support item assignment

>>> xs = bytearray(b'ahoj')
>>> xs
21: bytearray(b'ahoj')
>>> xs[1]
22: 104
>>> xs[1] = 72
>>> xs
23: bytearray(b'aHoj')
```

IV. Řetězce i bajtové objekty typu *bytes* jsou neměnitelné a tudíž

hešovatelné. Bajtová pole jako měnitelný protějšek bajtových řetězců nikoli:

```
>>> hash( 'ahoj' )
-1425894204
>>> hash( b'ahoj' )
1425894204

>>> hash( bytearray(b'ahoj') )
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    hash( bytearray(b'ahoj') )
TypeError: unhashable type: 'bytearray'
```

N-tice vs seznamy

I. Přirozeným konstruktorem prázdného seznamu je `[]`, seznam s jedním prvkem se zapíše jednoduše jako `[a,]`, přičemž ukončovací čárka není nutná.

U n-tice to už není tak jednoduché: N-tice totiž „vyrábí“ operátor `,`, nikoli závorky. Prázdnou n-tici tak sice získáme jako `()` a žádná čárka tam dokonce být nesmí, ale na druhou stranu n-tice o jednom prvku jest zapsána jako `(a,)` a zápis `a,` nedává smysl. Delší n-tice jsou pak už klasické: `a, b` stejně jako `(a, b)`.

II. N-tice jsou všude prezentovány jako jakési „zamrzlé“ seznamy. Ve skutečnosti je mezi n-ticemi a seznamy poněkud hlubší rozdíl: Chcete-li kupříkladu zachytit souřadnice konkrétního bodu ve 3D-prostoru jako jeho jakýsi identifikátor (nebudou se tudíž měnit), je zcela přirozené použít pro to n-tici o třech prvcích, která jasně evokuje, že každá z položek n-tice má jistý konkrétní význam (souřadnice x-ová, y-ová a z-ová). Seznam by v tomto případě byl dosti zavádějící – sice by na jednu stranu umožňoval měnit souřadnice objektu, ale na druhou by také umožňoval měnit jejich počet, což má samozřejmě úplně jiné vyznění.

III. N-tice jsou narozdíl od seznamů neměnné a tudíž hešovatelné (a tudíž použitelné jako prvky množin či jako klíče slovníků):

```
>>> ts = (1, 2, 3)
>>> hash(ts)
-378539185

>>> xs = [1, 2, 3]
>>> hash(xs)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    hash(xs)
TypeError: unhashable type: 'list'
```

Nesmíme to ovšem smíchat dohromady:

```
>>> xs = [1, 2, 3]
>>> ts = (1, 2, xs)
>>> ts
(1, 2, [1, 2, 3])
>>> type(ts)
<class 'tuple'>
>>> hash(ts)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    hash(ts)
TypeError: unhashable type: 'list'
```

→ Algoritmus pro „výrobu“ heše jednoduše „cestou“ narazil ve třetí položce n-tice na nehešovatelný typ.

Množiny a slovníky

I. Přirozeným konstruktorem prázdného slovníku je `{}`, slovník s jedním prvkem se zapíše jednoduše jako `{KLÍČ: HODNOTA, }`, přičemž ukončovací čárka není nutná.

Vzhledem k historické volbě konstruktoru slovníku to s množinou už bohužel není tak jednoduché – prázdnou množinu tak získáme pouze pomocí konstruktoru `set()` a jinak to nejde. Delší množiny už jsou od slovníků snadno k rozeznání, protože neobsahují dvojtečku: `{PRVEK, }` (čárka při tom opět není povinná)

II. O množinách můžeme celkem bez problémů uvažovat jako o slovnících bez hodnot – prvky množiny stejně jako klíče slovníku musí být jedinečné a navíc oboje neproměnného (a tudíž hešovatelné) typu.

III. Narozdíl od dychotomie mezi n-ticemi a seznamy, kde souvislost „neměnná n-tice – proměnný seznam“ není zcela košer, u množin je to jednoduché: Potřebujeme-li neproměnnou množinu (a tudíž i hešovatelnou), využijeme konstrukturu `frozenset()`. Tuto „zamrzlou“ množinu je pak samozřejmě možné používat jako prvek jiných množin nebo klíč do slovníků.

IV. Slovníky a množiny jako proměnné typy hešovatelné nejsou, zmražené množiny jako typ neproměnný pak ano:

```
# zmražená množina
>>> hash( frozenset({1,}) )
593349751

# množina
>>> hash( {1,} )
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    hash( {1,} )
TypeError: unhashable type: 'set'

# slovník
>>> hash( {1: 1,} )
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    hash( {1: 1,} )
TypeError: unhashable type: 'dict'
```

Rozsahy

Rozsahy se konstruují pomocí funkce `range()`. Na první pohled se tváří jako sekvence, ale nepodporují výřezy (*slicing*), spojování (*concatenation*) ani opakování (*repetiotion*):

```
>>> xs = range(10)
>>> xs
range(0, 10)

>>> xs[5]
5

>>> xs[:3]
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    xs[:3]
TypeError: sequence index must be integer, not 'slice'

>>> xs + range(3)
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    xs + range(3)
TypeError: unsupported operand type(s) for +: 'range' and 'range'

>>> 2 * xs
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    2 * xs
TypeError: unsupported operand type(s) for *: 'int' and 'range'
```

→ Jinými slovy typ je to sice iterovatelný, ale nikoli sekvenční.

Podobně používat na nich operátorů `in`, `not in` a funkcí `min()`, `max()` sice jde, ale je to pekelně neefektivní, protože rozsah je kvůli nim třeba proiterovat:

```
>>> 6 in range(10)
True

>>> 10 in range(10)
False

>>> max( range(10) )
9

>>> min( range(10) )
0
```

Vzájemné konverze

Prakticky všechny iterovatelné typy se dají převádět navzájem na sebe. Pár příkladů:

```
>>> xs = 'ahoj'
>>> tuple(xs)
('a', 'h', 'o', 'j')
>>> list(xs)
['a', 'h', 'o', 'j']

>>> xs = 'ahoj ahoj'
>>> set(xs)
{'a', 'h', 'j', 'o', ' '}
```

Převod mezi čísly a řetězci jsme už viděli na dřívějším slajdu:

```
>>> float(1)
1.0
>>> int(1.2)
1

>>> float('1.23')
1.23
>>> int('23')
23
```