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 – Čísla

int, float, complex

Přímo v Python'u bez importu dalších knihoven jsou k dispozici následující číselné typy:

- **int** – celá čísla
- **float** – reálná čísla (implementována pomocí Céčkovského *double*)
- **complex** – aneb dvojice `x + y[j|J]`, kde reálná a imaginární část jsou samy o sobě typu *float*

  → Typ *Boolean* je vlastně podtyp celých čísel. Navíc jsou na celých čísel k dispozici ještě bitové operace.

Základní operace na těchto typech jsou celkem nepřekvapivé:

```
>>> 3 + 5
8
>>> 3.1 + 5
8.1

>>> 8 / 2
4.0
>>> 8 / 2 + 3
7.0

>>> (1 + 1j) * (2 + 1j)
(1+3j)
```

Dělení vždy vrací *float*, i když argumenty jsou celá čísla:

```
>>> 4 / 2
2.0
>>> 4.2 / 2
2.1

>>> 4 // 2
2
>>> 4.2 // 2
2.0

>>> 4 % 2
0
>>> 4.2 % 2
0.20000000000000018
```

→ Výslovně na to upozorňuji, protože do Python'u 2.4 to bylo jinak.

Bitové operace

Bitové operace operují nad celočíselnými argumenty na úrovni bitů. Příklady:

```
>>> x = 10    # 1010
>>> y = 3     # 0011

>>> x & y    # 10
2
>>> x | y    # 1011
11
>>> x ^ y    # 1001
9

>>> x >> 1    # 101
5
>>> x >> 2    # 10
2
>>> y << 1    # 1100
12

>>> ~x    # -1011
-11
>>> ~y    # -100
-4
```

Kromě obligátního dělení a násobení mocninami dvojky...

```
>>> x = 7    # 111

>>> x//2 == x>>1    # 11
True
>>> x*4 == x<<2    # 11100
True
```

…se hodí např. pro porovnávání dvou bitových „řetězců“:

```
>>> x = 13    # 1101
>>> y = 9     # 1001

# Jsou všechny bity z x i v y?
>>> x == x&y    # 1100 == 1001
False
# Jsou všechny bity z y i v x?
>>> y == x&y    # 1001 == 1001
True
```

decimal

V případě typu *float* jsme s přesností výpočtů a zaokrouhlovacími chybami ponecháni na pospas aktuální implementaci a architektuře. Pokud chceme s reálnými čísly pracovat tak, jak je běžné v matematice, musíme se obrátit na modul `decimal`. Namátkou:

I. Přesnost výpočtů je volitelná, výchozí hodnotou je **28**:

```
>>> import decimal    # zavedení modulu 'decimal'

>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.14285714285714285714285714429')

>>> decimal.getcontext().prec = 5
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.14286')
```

- → Jak vidíte, tak vybraná přesnost zaokrouhluje podle aktuálně nastaveného zaokrouhlovacího algoritmu (na výběr je jich osm).
- → PS: `decimal.getcontext()` vrací aktuální nastavení, které ve výchozím stavu jest `Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999, capitals=1, flags=[Rounded, Inexact, InvalidOperation], traps=[InvalidOperation, DivisionByZero, Overflow])`
- → Nezapomeňte vrátit přesnost nazpátek.

II. Operace s čísly s neukončeným binárním rozvojem se chovají v rámci

typu *decimal* „normálně“:

```
>>> 1.1 + 2.2
3.3000000000000003

>>> decimal.Decimal('1.1') + decimal.Decimal('2.2')
Decimal('3.3')
```

Díky tomu pak odpadne i standardní problém malých čísel:

```
>>> 3 * 0.1 - 0.3
5.551115123125783e-17

>>> decimal.Decimal( str(3 * 0.1) ) - decimal.Decimal('0.3')
Decimal('0.0')
```

III. Čísla v *decimal* drží nastavený počet významných míst:

```
>>> 1.30 + 1.20
2.5

>>> decimal.Decimal('1.30') + decimal.Decimal('1.20')
Decimal('2.50')
```

IV. V rámci *decimal* máme k dispozici i několik tradičních hodnot, jako jsou `Infinity`, `-Infinity` a `NaN` plus rozlišení mezi `-0` a `+0`:

```
>>> decimal.Decimal('-Infinity') + decimal.Decimal('-Infinity')
Decimal('-Infinity')
```

V. K dispozici je samozřejmě též dedikovaná sada funkcí, včetně matematických, namátkou *exp*, *ln* nebo třeba *sqrt*. Příklad:

```
>>> decimal.Decimal(1).exp()
Decimal('2.71828182845904523536028747 1')
```

Pokud budete *decimal* používat, může se vám pro zpřehlednění zápisu hodit následující „trik“:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

fractions

Budete-li potřebovat pracovat s racionálními čísly, může se vám hodit modul `fractions`. Pár příkladů:

```
>>> import fractions

>>> fractions.Fraction()
Fraction(0, 1)

>>> fractions.Fraction(16)
Fraction(16, 1)
>>> fractions.Fraction(16, -10)
Fraction(-8, 5)
>>> fractions.Fraction('-16/10')
Fraction(-8, 5)

>>> fractions.Fraction('-16/10') + fractions.Fraction('3/5')
Fraction(-1, 1)
```

Kromě toho, že za vás *fractions* automaticky zjednodušuje zlomky, také vám třeba pomůže s hledáním největšího společného dělitele:

```
>>> fractions.gcd(42, 56)
14
```

Když na to přijde, můžete použít podobný „trik“ jako u *decimal*:

```
>>> F = fractions.Fraction
>>> F('3/4') + F('2/16')
Fraction(7, 8)
```