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

brainfuck – Programování v jazyce brainfuck

Principy programování I

I. Jelikož *brainfuck* umí vytisknout znak, jehož ASCII-kód se nachází v aktuální paměťové pozici, princip vypsání např. slova `AHOJ` by mohl vypadat třeba takto (nezapomeňte, že po spuštění programu jsou v „paměti“ všude nuly):

1. inkrementuj aktuální buňku na hodnotu 65 (ASCII-hodnota písmene `A`) a vytiskni ji
2. posuň se o buňku dál, tu inkrementuj na hodnotu 72 (aneb `H`) a vytiskni ji
3. posuň se o buňku dál, tu inkrementuj na hodnotu 79 (aneb `O`) a vytiskni ji
4. posuň se o buňku dál, tu inkrementuj na hodnotu 74 (aneb `J`) a vytiskni ji

Takový kód bude pochopitelně příšerně neefektivní a doslova šílený:

```
# Kód pro zápis písmene A:
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.
# Kód pro zápis písmene H:
>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Kód pro zápis písmene O:
>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Kód pro zápis písmene J:
>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

II. Velmi rychle vás asi napadne, jak program zjednodušit – místo přičítání jedničky vždy v nové buňce od nuly budeme postupně přičítat a odčítat potřebné hodnoty v rámci buňky jediné:

1. inkrementuj aktuální buňku na hodnotu 65 (ASCII-hodnota písmene `A`) a vytiskni ji
2. inkrementuj aktuální buňku z hodnoty 65 na hodnotu 72 (aneb `H`) a vytiskni ji
3. inkrementuj aktuální buňku z hodnoty 72 na hodnotu 79 (aneb `O`) a

vytiskni ji

4. dekrementuj aktuální buňku z hodnoty 79 na hodnotu 74 (aneb `J`) a vytiskni ji

Takový kód už bude mnohem „hezčí“, přestože pořád žádná sláva ^_^

```
# Kód pro zápis písmene A:
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.
# Upravený kód pro zápis písmene H:
+++++++.
# Upravený kód pro zápis písmene O:
+++++++.
# Upravený kód pro zápis písmene J:
-----.
```

III. Další zjednodušení umožňují pochopitelně smyčky, ale na ty se musíme nejdříve podívat.

Smyčky

Podstatu smyček si ukážeme na vytisknutí znaku `A`, jehož ASCII-hodnota je – jak už víme – 65:

I. Jelikož např. `65 = 8*8 + 1`, můžeme evidentně spoustu práce ušetřit osminásobným opakováním inkrementace o osm (nebo třeba šestkrát po deseti a vůbec). Jak na to? Následujícím kódem:

```
++++++++[>++++++++<-]
```

Rozeberme si ho podrobně:

1. Nejdříve jsme aktuální paměťovou buňku nastavili z hodnoty 0 (předpokládáme, že takovou právě měla; stejně jako všechny ostatní) na hodnotu 8: `++++++++`
2. Nyní následuje „magická“ část kódu:
    1. Příkaz `[` se podívá na aktuální buňku – jelikož je v ní 8 (právě jsme je tam zanesli), předá vykonávání programu na další znak.
    2. Zde natrefíme na příkaz `>`, který přesune ukazatel o jednu buňku dále. Na tomto místě je zatím 0, ale sekvence navazujících příkazů `++++++++` tam zanese také 8.
    3. Následuje příkaz `<`, který vrátí ukazatel o jednu buňku zpět. Následující příkaz `-` tuto buňku dekrementuje, takže z původních 8

dostaneme 7. Stav těchto dvou sousedních buněk je v dané chvíli tedy: `7 8`

4. Poslední příkaz `]` zkontroluje stav aktuální buňky – jelikož je tam 7 (právě jsme provedli dekrementaci), zařídí příkaz `]` přesun vykonávání kódu na znak následující po odpovídajícím začátku smyčky `[`, tj. opět na začátek sekvence příkazů `++++++++<-`.

3. Další běh kódu už je nyní jasný: Ukazatel se díky `>` posune o buňku dále (kde je zatím 8), kde se pomocí `++++++++` přičte k aktuálnímu obsahu dalších 8 (takže tam bude už 16), poté se ukazatel díky `<` vrátí na předchozí buňku (kde je zatím 7) a odtud je za pomoci `-` odečtena jednička (takže se zde objeví 6). Před dalším vyhodnocením podmínky konce cyklu `]` je tedy stav obou paměťových buněk: `6 16`

4. Kód takto pokračuje v postupné inkrementaci druhé buňky o 8 v každém kroku, přičemž počet kroků určuje první buňka, na jejíž hodnotu se ptá podmínka cyklu. Následující posuny ve zdrojovém kódu na příkaz `]` tak naleznou obě buňky postupně v následujících stavech: `5 24` → `4 32` → `3 40` → `2 48` → `1 56` → `0 64`

5. Až do stavu `1 56` se na běhu programu nic nezmění. Ale po poslední inkrementaci druhé buňky o 8 je první buňka příkazem `<-` dekrementována na nulu a tak se změní chování následujícího příkazu `]`, který až dosud vracel cyklus na začátek – jelikož na aktuální buňce je 0, předá příkaz `]` v tuto chvíli běh programu na další znak zdrojového kódu po sobě. A jelikož tam v tomto programu již žádný další příkaz není, končí běh programu se stavem buněk `0 64`.

II. Nyní už stačí doplnit předchozí smyčku..

```
++++++++[>++++++++<-]
```

..o přesun (z buňky cyklu na následující buňku písmene) a jednu inkrementaci..

```
++++++++[>++++++++<-] > +
```

..aby se ve druhé buňce nacházela požadovaná hodnota 65, kterou pomocí příkazu `.` (tečka)..

```
++++++++[>++++++++<-] > + .
```

..vytiskneme na výstup jako znak `A`. Jak jednoduché, milý Watsone ^_~

Principy programování II

Náš ukázkový program pro výpis slova `AHOJ`..

```
# Kód pro zápis písmene A:
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.
# Upravený kód pro zápis písmene H:
+++++++.
# Upravený kód pro zápis písmene O:
+++++++.
# Upravený kód pro zápis písmene J:
-----.
```

..tak můžeme podstatně zjednodušit nahrazením první velké inkrementace o 65 smyčkou:

```
# Kód pro zápis písmene A pomocí smyčky:
++++++++[>++++++++<-] > + .
# Upravený kód pro zápis písmene H:
+++++++ .
# Upravený kód pro zápis písmene O:
+++++++ .
# Upravený kód pro zápis písmene J:
----- .
```

Bez komentářů a v jedné řádce tak celkově dostaneme krásný ^_^ výsledný program:

```
++++++++[>++++++++<-]>+.+++++++.+++++++.-----.
```

„Hello World!“ podruhé

Pochopit dříve uvedený program pro výpis *Hello World!* je nyní, navíc za pomoci komenentářů v kódu, určitě snadné – jeho základní trik spočívá v tom, že si buněk pro tisk připraví více a operuje s hodnotami na tom paměťovém místě, kde je to zrovna jednodušší z hlediska požadovaného výstupu:

```
+++++ +++++             initialize counter (cell #0) to 10
[                       use loop to set the next four cells to 70/
    > +++++ ++              add  7 to cell #1
    > +++++ +++++           add 10 to cell #2
    > +++                   add  3 to cell #3
    > +                     add  1 to cell #4
    <<<< -                  decrement counter (cell #0)
]
> ++ .                  print 'H'
> + .                   print 'e'
+++++ ++ .              print 'l'
.                       print 'l'
+++ .                   print 'o'
> ++ .                  print ' '
<< +++++ +++++ +++++ .  print 'W'
> .                     print 'o'
+++ .                   print 'r'
----- - .               print 'l'
----- --- .             print 'd'
> + .                   print '!'
> .                     print '\n'
```

→ Zdroj: http://en.wikipedia.org/wiki/Brainfuck

Programování – základní konstrukce

Ukažme si několik užitečných konstrukcí, které nám – snad – usnadní čtení (a možná i psaní :) skutečných brainfuckovských programů:

→ Volně podle různých zdrojů na internetu.

I. Nastavení aktuální buňky na 0:

```
[-]
```

→ Tady není co řešit – aktuální paměťová buňka se bude dekrementovat o jedničku tak dlouho, dokud na ní nebude nula.

II. Mírná modifikace předchozího algoritmu – vynulování všech předchozích buněk:

```
[
    [-]
    <
]
```

→ Je-li aktuální paměťová buňka nenulová, „pustí“ nás vnější

smyčka dovnitř. Tamější kód pak provede vynulování aktuální buňky (podle předchozího postupu) a provede přesun doleva o jedno místo. Je-li tato buňka také nenulová, celý proces se zopakuje. Vše poběží tak dlouho, dokud nenatrefíme na první nulovou buňku.

III. Ve velmi podobném duchu zvládneme i operace „rewind“ a „fast-forward“, tedy přesun na první nulovou buňku vlevo či vpravo od aktuální:

```
# rewind
[<]

# fast/forward
[>]
```

→ V obou případech se obsah smyčky, tj. posun daným směrem po paměťové pásce, provádí tak dlouho, dokud se nepřesuneme na buňku obsahující 0.

IV. Velmi podobným způsobem můžeme hledat první buňku dané hodnoty. Např. pro nalezení první buňky s hodnotou 1 směrem doprava použijeme kód:

```
-[+>-]+
```

Pro totéž doleva vyměníme příkaz posunu > za opačný <. A pro vyhledání jiné hodnoty než právě 1 patřičně „zmnožíme“ všechny plusy a mínusy.

V. S trochou disciplíny zvládneme i rozhodovací konstrukci if-then-else. Nejde o nic jiného, než o určení, zda se provede aktuální smyčka nebo až následující kód:

```
# nastavení podmínky
kód vyhodnocení podmínky (tedy nastavení aktuální buňky na 0 nebo
# IF
[kód větve IF]
# ELSE
kód větve ELSE
```

Příklad na přepis vstupních malých písmen na výstupní velká, upraveně podle Wikipedie:

```
,----------[----------------------.,----------]
```

- Za předpokladu, že začínáme s vynulovanou buňkou, způsobí první část ,---------- *ne*vykonání následující smyčky, pokud stiskneme *ENTER*: Solidní implementace *brainfuck*u by totiž v tomto případě měla vrátit číslo 10 (viz následující slajdy), které bude kódem vynulováno.
- Nestiskli-li jsme *ENTER*, smyčka naopak *bude* vykonána: To způsobí

odečtení dalších 22 od jejího aktuálního obsahu, celkem tedy 32, což je právě rozdíl mezi stejnými písmeny ve velkém a malém provedení v ASCII-tabulce. Výsledné velké písmeno bude vytištěno.

- Kód uvnitř smyčky dále načte další vstupní hodnotu, od které opět odečte 10, a na konci s takto získanou hodnotou provede test na ukončení smyčky.

  → Nečeká-li kód konkrétního překladače *brainfuck*u při vstupu na ENTER (v Python'u např. pomocí `msvcrt.getch()`), bude se smyčka neustále opakovat, dokud *ENTER* nestiskneme. Čeká-li zadání na vstup (v Python'u např. pomocí `input()` nebo `sys.stdin.read()`), smyčka proběhne pouze jednou, protože s každým zadaným písmenem bude načten zároveň i konec řádku `\n`, tedy 10.

Programování – aritmetické operace

→ Volně podle různých zdrojů na internetu.

I. Přesun (přičtení) obsahu aktuální buňky do buňky následující:

```
[>+<-]
```

→ V první buňce tedy bude po provedení smyčky 0.

→ Ve druhé buňce bude buď kopie obsahu první buňky (to tehdy, byla-li v ní původně 0), nebo součet jejího vlastního původního obsahu s obsahem buňky první.

→ Změnou počtu posunů snadno určíme, o kolik buněk dále se má obsah aktuální buňky přesunout. Například pro tři by kód byl:

```
[>>>+<<<-]
```

Nedestruktivní kopie (přičtení) obsahu aktuální buňky do buňky následující je poněkud složitější:

```
[>+>+<<-]>>[<<+>>-]
```

→ V první buňce tedy zůstane zachován původní obsah: První smyčka zajistí přesun první buňky do druhé a třetí, přičemž je první buňka vynulována. Druhá smyčka pak přesune obsah třetí buňky zpátky do první, aby bylo zadání učiněno zadost. Celé to takto funguje samozřejmě pouze tehdy, byla-li v obou následujících buňkách původně nula.

→ Pro kopii do vzdálenějšího místa platí podobná poznámka

jako u předchozího bodu. Jen si musíme hlídat, kam umístíme buňku dočasnou.

II. Podobně se dá implementovat i odčítání – operaci `+` na příslušném místě zaměníme za operaci `-`. Obsah aktuální buňky odečteme od buňky následující destruktivním způsobem následujícím kódem:

```
[>-<-]
```

→ Tedy: Z následující buňky se odečítá 1 tak dlouho, dokud je aktuální buňka (při postupné dekrementaci o jedničku) nenulová.

III. Nejsnadnější násobení je násobení dvěma – „vem sám sebe a zdvoj se“. Implementace může použít „triku“ – buňku zkopíruj do vedlejšího paměťového místa, které poté přesunutím „přičti“ k první buňce. Přibližně tedy:

```
# A) Zkopíruj buňku 1 do buněk 2 a 3 (buňka 1 se přitom smaže):
[>+>+<<-]

# B) Přesuň se na buňku 3 a tu přičti k buňce 2 (buňka 3 se přitom
>>[<+>-]
# Nyní je v buňce 2 dvojnásobek původního obsahu buňky 1.

# C) Přesuň se na buňku 2 a její obsahu přesuň do buňky 1 (buňka 2
<[<+>-]
# Nyní už je v buňce 1 dvojnásobek její původní hodnoty.
```

→ Celé to takto funguje samozřejmě pouze tehdy, byla-li v obou následujících buňkách původně nula.

Násobení jinými čísly (a případně i umocňování) jsou vcelku názorné operace – v principu jde o přičítání různých čísel různým počtem krát.

IV. Dělení je ovšem netriviální. V principu jde o něco podobného jako u kombinace odčítání a násobení – od obsahu jedné buňky budeme opakovaně odečítat obsah buňky jiné. Musíme si při tom ovšem ještě někde jinde počítat, kolikrát jsme dané odčítání provedli. A navíc nezapomenout na to, že čísla začínají v ASCII-tabulce až na pozici 48 decimálně.

Příklad na odečtení dvou jednociferných čísel, upraveně podle Wikipedie:

```
,>,>                                    # Ulož 2 čísla ze vstupu v buňká
++++++[-<--------<--------->>]          # a od každého z nich odečti 48

<<                                      # Přesuň se zpátky na buňku (0)
[                                       # Hlavní smyčka; opakuje se, dok
    >                                       # Přesuň se na buňku (1)
    [->+>+<<]                               # Destruktivně zkopíruj děli
    >                                       # Přesuň se na buňku (2)
    [
        -<<-                                    # Odečti dělitele (divis

        [>]>>>                                  # Připrav test: Je-li dě
        [< [>>>-<<<[-]] >>]
        <<
    ]
    >>>+                                    # Přidej jedničku k podílu (
    <<[-<<+>>]                              # Destruktivně zkopíruj děli
    <<<                                     # Přesuň paměťový ukazatel n
]                                       # vrať se na začátek hlavní smyč

>[-]>>>>[-<<<<<+>>>>>]                  # Destruktivně zkopíruj podíl (q
<<<<++++++[-<++++++++>]<.               # Přičti 48 a vypiš výsledek
```

→ Protrasujete-li si kód, zjistíte, že nejvnitřnější smyčka `[< [>>>-<<<[-]] >>]` se prakticky celou dobu výpočtu ignoruje. Nastoupí teprve ve chvíli, kdy se se postupným odčítáním konečně vynuluje buňka číslo 0. Její jedinou funkcí je zajistit, že kód následující po smyčce již znovu nezmění výsledek dělení (on ho totiž o jedničku zvýší, ale tato smyčka ho právě proto preventivně o jedničku sníží).

Programování – vstup a výstup

Vstup do *brainfuck*ovských programů zajišťuje příkaz , (tedy *čárka*), výstup pak příkaz . (tedy *tečka*). Jsou celkem pochopitelným zdrojem problémů, zvláště při konstrukci přenositelných programů, protože mimo jiné se každý systém ke konci řádku chová jinak. Souhrnně se to dá sepsat do dvou bodů:

- Při zpracování konce řádky se většina *brainfuck*ovských programů řídí nepsaným pravidlem, že při načtení konce řádky uloží 10 (ať už obdržely `CRLF`, tedy `\r\n`, tedy 13 a 10, nebo cokoliv jiného) a podobně při jeho zápisu také 10 zapíše.

  → V Python'u máme situaci ulehčenou – jelikož je tento problém poměrně rozšířený, řeší ho Python interně automatickým překladem čehokoliv ze vstupu na `\n` (čili právě požadovaných 10) a zase zpět.

- Podobným problémem je určení chování příkazu , pro čtení, pokud už není odkud číst (tj. nastala situace EOF, *end of file*). Zde bohužel žádný konsenzus není – některé programy nedělají nic, jiné do aktuálního paměťového místa zapíší 0 a ještě jiné -1.

  → Některým programům to možná bude vadit, ale doporučením asi bude nedělat nic (tj. ponechat na aktuální paměťové buňce původní hodnotu).

PS: Druhý bod v praxi znamená, že implementace programu *cat*, který přepisuje svůj vstup na výstup, je právě trojí:

```
# EOF returns 0
,[.,]

# EOF returns minus 1
,+[-.,+]

# no change on EOF
,[.[-],]
```

Poznámky k implementaci

Snad je zřejmé, že narozdíl od běžných programovacích jazyků, u kterých už letmý pohled do libovolného místa programu vám dá alespoň přibližnou představu, co se zde bude dít, je *brainfuck* úplně jiný – pokud neznáte do nejmenších podrobností aktuální stav programu (tj. obsazení jednotlivých paměťových buněk), nemáte prakticky žádnou šanci dobrat se byť mizivého chápání funkce programu.

Naštěstí vás to nemusí zas až tak moc trápit, protože *psát* programy v *brainfuck*u po vás nikdo nechce, vy je máte pouze vykonat ^_~ Následuje několik poznámek, které by vám mohly s implementací pomoci:

- Díky automatickému překladu konců řádek na \n, tj. 10, máte část práce ulehčenou. Použijete-li pro vstup `sys.stdin.read(1)`, bude se skoro vše chovat tak, jak má.
- S výstupem je to podobné. Jen si při použití `print()` dejte pozor, abyste toho netiskli víc, než chcete, a při použití `sys.stdout.write()` zase nezapomeňte vždy hned zavolat i `sys.stdout.flush()`, aby se zapsaný řetězec skutečně objevil i na obrazovce (Python bafruje řádky, případně

bloky kódu po 8 kB).

- Smyčky `[]` můžete „hardcorově“ řešit prohledáváním zdrojového souboru za běhu, ale spousta implementací si je připraví dopředu proskenováním zdrojového kódu před jeho vlastním vykonáním.
- Omezení paměťové pásky na původních 30 000 míst vám může být u dynamického programovacího jazyka a na dnešním hardwaru vcelku ukradené. Spíše si dejte pozor na to, že některé programy vyžadují na první buňce pevnou zarážku (tj. páska se nemůže rozšiřovat doleva od nulté buňky).
- Pokud nebudete *brainfuck*ovskými programy zpracovávat binární data nebo něco podobného, budou vám stačit osmibitové paměťové buňky (tj. schopné pojmout pouze čísla 0 až 255). Spousta programů s tím dokonce počítá a pokud se tomu nepřizpůsobíte, nebudou chodit správně.
- Docela velké množství implementací *brainfuck*u tradičně rozšiřuje základní sadu příkazů o následující dva: `#` („vytiskni debugovací informace“) a `!` („odděl kód od jeho vstupu“). Můžete se s nimi v některých programech setkat.