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 – Kódování, Unicode

ASCII

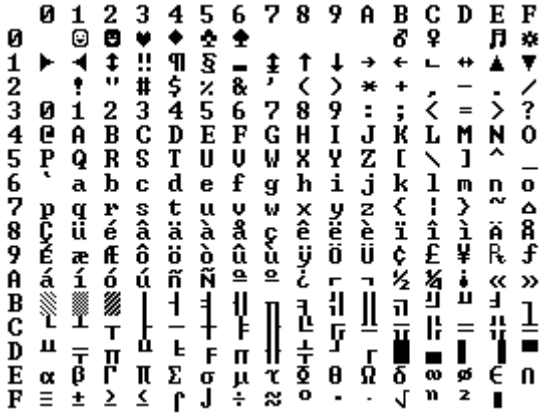
„Prvním“ počítačům (nebo spíše anglicky hovořícím uživatelům necítícím potřebu psát slova s cizími písmeny :) stačila tabulka znaků o 128 položkách, tzv. ASCII:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Jednotlivé prvky tabulky mají přiřazená čísla 0-127 decimálně (aneb 0-7F hexadecimálně nebo 0-177 osmičkově) – k jejich zachycení v počítači tudíž stačí pouhých 7 bitů z každého bajtu.

Nadstavby ASCII

Počítače však už standardně používaly *bajty*, tj. bitů 8. To dávalo k dispozici dalších 128 možných znaků. Není ale divu, že každý si do volného místa vymyslel něco úplně jiného, kupříkladu i grafické prvky:



Kódové stránky

Pro rozlišení mezi různými rozšířeními v osmém bitu se zavedly tzv. *kódové stránky* (*code pages*). Příkladem budiž CP 852 (Latin 2), Windows 1250 (Central Europe), ISO/IEC 8859-5 (Cyrillic).

Alespoň že se všichni shodli, že prvních 128 znaků bude staré dobré ASCII – pokud jste nevěděli, v jaké CP byl dokument uložen, pořád jste přečetli alespoň anglickou abecedu a další základní znaky z ASCII-rozsahu. Ovšem všechno ostatní … škoda mluvit.

→ Tedy pokud jste nepsali texty editorem WordStar, který prý osmý bit používal pro označení posledního znaku slova *_*

Intermezzo

Mimochodem – nejen že co jazyk, to vlastní CP. Spíš co výrobce nebo program to vlastní CP *_* Pamětníkům dodnes vstávají vlasy hrůzou na hlavě, když si vzpomenou na různá kódování češtiny.

A nezapomeňme na východní jazyky, kterým pouhých 128/256 znaků na zaznamenání jejich obrázkových písem o desítkách tisíc znaků v žádném případě stačit nemohlo! Ty svůj problém vyřešily použitím tzv. DBCS-systému – něco se zapsalo bajtem jedním, něco (většina) dvěma.

→ A co takhle zkusit vytvořit dokument s alespoň dvěma různými písmy, co vy na to, he?

Enter Unicode!

Dlouho přehlížený problém, který se provalil na světlo s příchodem Internetu (a tedy do té doby nevídanou volnou výměnou dokumentů mezi různými počítači a systémy), naštěstí pár osvícenců řešilo už v osmdesátých letech. Výsledkem jejich snažení byla univerzální množina znaků, tzv.

UNICODE

→ Kromě samotné tabulky znaků řeší Unicode ale i mnoho dalších věcí, zvláště pro implementátory – namátkou třebas

řazení nebo vykreslování.

Code Points

V Unicode'u má každý znak (jehož přesná definice není ani trochu triviální!) přiřazeno jedinečné konkrétní číslo, tzv. *code point*. Tak kupříkladu latinské *A* má číslo U+0041 a řecká *α* zase U+03B1.

Přiřazených znaků je v dané chvíli již přes sto tisíc, tudíž je jasné, že pro jejich zachycení si už dávno nevystačíme ani se dvěma bajty.

Kódování

Nejjednodušším způsobem jak uložit text v Unicode'u bude asi prostě zapsat jeho *code point* přímo jako bajty (pro prvních 65536 znaků případ UTF-16). Číhá na nás ale několik zádrhelů:

- Ne všechny systémy ukládají vícebajtové hodnoty ve stejném pořadí (viz *big-endian* a *little-endian*)!
- Vidíte, kolik zbytečných nul bude obsahovat text, na který by bývalo stačilo ASCII?

První problém (pořadí bajtů) byl nuceně vyřešen (téměř) povinným zápisem jisté sekvence bajtů na začátek každého unicodového řetězce, tzv. *BOM* (*byte order mark*). (Podle jejich pořadí pak program pozná, o kterou z obou variant jde.)

> → Konkrétně je BOM sekvencí `FE FF` pro UTF-16 a `00 00 FE FF` pro UTF-32. V UTF-8 BOM (který pak je `EF BB BF`) spíše překáží – za prvé je tam zbytečný (UTF-8 je sekvence jednotlivých bajtů, takže žádné problémy s pořadím zápisu vícebajtových hodnot nemá) a za druhé s ním spousta programů nepočítá.

Druhý problém si ovšem vyžádal hodně přemýšlení: Jak jednoznačně a výhodně zaznamenat oněch více jak sto tisíc znaků, aby se všechny vešly, ale přitom jsme nespotřebovávali zbytečně moc místa?

Enter UTF-8!

Naprosto geniálním řešením (i když východní Asie naše nadšení sdílet nemusí) je kódování UTF-8.

- Znaky s *code points* 0 až 127, tj. staré dobré ASCII, se vlezou do jednoho bajtu a jsou zaznamenány úplně stejně.
- Znaky s vyššími *code points* se zaznamenávají rozdílným počtem bajtů (dvěma až čtyřmi).

UTF-8 algoritmus

oktet 1	**oktet 2**	**oktet 3**	**oktet 4**	***code point***
0xxxxxxx				0xxxxxxx
110zzzyy	10xxxxxx			00000zzz yyxxxxxx
1110wwww	10zzzzyy	10xxxxxx		wwwwzzzz yyxxxxxx
11110uuu	10vvwwww	10zzzzyy	10xxxxxx	000uuuvv wwwwzzzz yyxxxxxx

→ Podle: "Unicode Explained" by Jukka K. Korpela (O'Reilly 2006), ISBN 0-596-10121-X & http://en.wikipedia.org/wiki/UTF-8

Poznámka k UTF-8

Zatímco české znaky s nabodeníčky v libovolném osmibitovém kódování zabírají samozřejmě právě jeden bajt, v kódování UTF-8 zabírají bajty dva. Podobně asijská písma typicky zapisovaná pomocí dvoubajtových kódování zabírají na každý znak bajty tři.

V reálných textech se ale ukazuje, že rozdíl je zanedbatelný.

→ Ad Asie: Alespoň to tvrdí Wikipedie, neověřoval jsem to.

A co na to Python?

Odpověď je dvojí a nehezká:

- Python 3 – pokud není řečeno jinak, je vše pokládáno za UTF-8.

nirvána

- Python 2 – vše je pokládáno za ASCII a pokud vám to nestačí, připravte se na škrábání levou rukou za pravým uchem s nohama svázanýma do kozelce.

  → U Python'u 2 vás budou zajímat metody řetězce *decode()* a *encode()* plus systémová knihovna *codecs*.