ŘETĚZCE

7

V této kapitole probereme standardní řetězce a řetězcové operace v Pythonu.

V této kapitole:

Řetězce jakožto sekvence znaků

Řetězce jsou sekvencemi znaků.

Pro účely vyjmutí znaků nebo podřetězců můžeme řetězce považovat za posloupnosti znaků, což znamená, že pro ně můžeme používat indexy nebo zápis řezu:

```
>>>
```

```
>>> x = "Ahoj"
>>> x[0]
⇒   'A'
>>> x[-1]
⇒   'j'
>>> x[1:]
⇒   'hoj'
```

Indexování (včetně řezů) lze využít k extrakci znaků nebo podřetězců z řetězce.

Jedním z možných použití zápisu formou řezu je u řetězců možnost odseknout znak nového řádku od konce řetězce, obvykle od řádku, který byl načten ze souboru:

```
>>>
```

```
>>> x = "Sbohem\n"
>>> x = x[:-1]
>>> x
⇒   'Sbohem'
```

Funkce *len* vyhledá počet znaků v řetězci.

Můžete také pomocí funkce `len` určit, z kolika znaků se řetězec skládá, stejně jako při zjišťování počtu prvků v seznamu:

```
>>>
    >>> len("Sbohem")
    ⇒  6
```

Řetězce nelze modifikovat.

Řetězce však přesto nejsou seznamy znaků. Nejviditelnějším rozdílem mezi řetězci a seznamy je ten, že na rozdíl od seznamů *řetězce nelze modifikovat*. Pokus např. o `retezec.append('c')` nebo `retezec[0]= 'H'` vyvolá výjimku. V předchozím příkladu si všimněte, že jsme oddělili znak nového řádku (`\n`) od řetězce tím, že byl vytvořen nový řetězec, který je řezem původního řetězce, a ne přímo změnou předchozího řetězce. To je jedno ze základních omezení Pythonu, vynucené požadavky na jeho efektivitu.

Základní řetězcové operace

Operátor + zřetězí řetězce.

Nejjednodušším (a pravděpodobně nejběžnějším) způsobem spojování řetězců v Pythonu je použití řetězcového operátoru řetězení `+`:

```
>>>
    >>> x = "Sbohem " + "světe"
    >>> x
    ⇒  'Sbohem světe'
```

K němu je analogický multiplikační řetězcový operátor `*`, jenž, jak jsem zjistil, je někdy, i když ne zrovna často, také užitečný:

```
...
    >>> 8 * "x"
    ⇒  'xxxxxxxx'
```

Speciální znaky a escape sekvence

Escape sekvence jsou (pod)řetězce začínající znakem `\`. Obecně představují speciální znaky.

Viděli jste již několik sekvencí znaků, které Python považuje, pokud jsou použity uvnitř řetězců, za speciální: "`\n`" představuje přechod na nový řádek, "`\t`" ztělesňuje tabulátor. Sekvence znaků, které začínají zpětným lomítkem a které slouží k tomu, aby představovaly jiné znaky, jsou nazývány *escape sekvencemi*. Escape sekvence se obecně používají k reprezentaci *speciálních znaků*, to znamená takových znaků (jako je tabulátor a znak nového řádku), které nemají standardní jednoznakovou tištěnou podobu. V tomto oddíle knihy se budeme podrobněji zabývat problematikou escape sekvencí, speciálních znaků a příbuznými tématy.

Základní escape sekvence

Základními escape sekvencemi jsou `\n`, `\t`, `\\`, `\'`, `\"`, `\a`, `\b`, `\f`, `\r`, a `\w`.

Python poskytuje menší zásobu escape sekvencí tvořených dvěma znaky pro použití v řetězcích (viz tab. 7.1):

Tabulka 7.1 Escape sekvence

Escape sekvence	Představuje znak
`\'`	jednoduché uvozovky
`\"`	dvojité uvozovky
`\\`	zpětné lomítko
`\a`	znak „zvonek"
`\b`	o jeden znak zpět (backspace)
`\f`	posuv papíru (nová stránka)
`\n`	nový řádek
`\r`	návrat vozíku (není stejný jako `\n`)
`\t`	tabulátor
`\v`	svislý tabulátor

Znaková sada ASCII, kterou používá Python a standardní znakové sady na většině počítačů, definuje několik speciálních znaků. Pracuje se s nimi prostřednictvím numerických escape sekvencí, které jsou popsány v dalším oddíle.

Numerické (osmičkové a šestnáctkové) escape sekvence

Numerickou escape sekvencí může být vyjádřen libovolný znak.

Do řetězce můžete zařadit libovolný znak sady ASCII pomocí osmičkové (se základem 8) nebo šestnáctkové (se základem 16) escape sekvence odpovídající tomuto znaku. Osmičková escape sekvence je tvořena znakem zpětného lomítka a trojčíselným kódem definujícím osmičkové číslo; odpovídající ASCII znak je nahrazen osmičkovou escape sekvencí. Šestnáctková (hexadecimální) escape sekvence je podobná, ale začíná znaky „\x", nikoli pouhým „\" a skládá se z libovolného šestnáctkového čísla. Escape sekvence je ukončena v okamžiku, kdy je nalezen znak, který není šestnáctkovým číslem. Například, v tabulce ASCII znaků má znak 'm' desítkovou hodnotu 109. To je 155 v osmičkové soustavě a 6D v šestnáctkové soustavě, takže:

```
>>>
    >>> 'm'
    ⇒   'm'
    >>> '\155'
    ⇒   'm'
    >>> '\x6D'
    ⇒   'm'
```

Všechny tři výrazy jsou vyhodnoceny jako řetězec, který obsahuje jediný znak „m". Nicméně tyto formy budou za normálních okolností využity tak, jako by představovaly znaky, které nemají tištěnou podobu. Znak nového řádku "\n" má osmičkovou hodnotu 012 a šestnáctkovou 0A.

```
>>>
    >>> '\n'
    ⇒   '012'
```

[8] Poznámka českého vydavatele: V novější verzi Pythonu se namísto `'\012'` zobrazuje `'\n'`

```
>>> '\012'
⇒   '\012'
>>> '\x0A'
⇒   '\012'
```

Tisk versus vyhodnocení řetězce se speciálními znaky

Hovořili jsme o rozdílu mezi pythonovým interaktivním vyhodnocením výrazu a tiskem výsledku stejného výrazu pomocí příkazu `print`. Ačkoliv se jedná o stejný řetězec, tyto dvě operace mohou vytvořit na obrazovce výstupy, které vypadají velmi odlišně. Řetězec, který je pouze vyhodnocen na vyšší úrovni interaktivního režimu Pythonu, se předvede se všemi jeho speciálními znaky s osmičkovými escape sekvencemi, které objasňují, z čeho se řetězec skládá. Zato příkaz tisku předá řetězec přímo na terminálový program, který může interpretovat speciální znaky zvláštním způsobem. Zde je příklad toho, co se stane s řetězcem, který se skládá ze znaku 'a', následovaného znakem nového řádku, tabulátorem a znakem 'b':

```
>>> 'a\n\tb'
⇒   'a\012\011b'          # 'a\n\tb' v novější verzi Pythonu
>>> print 'a\n\tb'
a
    b
```

Znak čárky za řetězcem zabrán příkazu *print*, v přidání nového řádku.

V prvním případě se explicitně vytisknou znaky nového řádku a tabulátoru; ve druhém se tyto speciální znaky interpretují jako nový řádek a tabulátor.

Normální příkaz `print` také automaticky přidá na konec řetězce znak nového řádku. Nastávají i situace (např. v okamžiku, kdy jsou řádky načteny z externího souboru a již končí novými řádky), jaké si nepřejeme. Čárka za řetězcem způsobí, že příkaz `print` znak nového řádku nepřipojí.

```
>>> print "abc\n"
⇒   abc
⇒
>>> print "abc\n",
⇒   abc
>>>
```

Modul string

Protože *string* je modul, je nutno ho před použitím importovat.

Většina dalších řetězcových funkcí Pythonu (spolu s některými užitečnými konstantami) je součástí standardního modulu `string`.[9]

Moduly podrobněji probereme v dalších kapitolách. Pro účely tohoto oddílu vám postačí vědět, že každý program v Pythonu, který používá funkce modulu `string`, musí mít uveden řádek:

```
import string
```

[9] Python ve verzi 1.6 bude obsahovat řetězcové metody, které budou mít funkčnost tohoto modulu. Nové verze kódu, v němž jsou již použity, můžete vidět (a najít popisy těchto metod) v *Python Library Reference*. Nicméně, kvůli velikosti základního kódu jazyka, bude i nadále tento modul také k dispozici. Kód, který jste napsali bude nadále fungovat a vy jej budete budete moci nadále používat. V nové verzi budou i dvě nové metody: `startswith` a `endswith`.

na jeho začátku nebo poblíž, a že na funkce a hodnoty dat v modulu `string` se vždy odkazujete zápisem `string.`*`funkce`* nebo `string.`*`data`*; to znamená, že jsou uvozeny slovem `string` následovaným tečkou.

Žádná z funkcí v modulu *`string`* ve skutečnosti neměnířetězec.

Protože jsou řetězce neměnitelné, jsou funkce definované v modulu `string` funkcemi v nejčistším smyslu tohoto slova, to znamená, že jejich výstupem je vždy hodnota a ony samy žádným způsobem nemění jejich argumenty.

Začneme těmi operacemi z modulu `string`, které jsou nejužitečnější a běžně používané, a potom probereme některé méně běžné, ale stále užitečné operace. Nakonec probereme několik rozmanitých bodů, které souvisí s modulem `string`. Z modulu `string` zde není zdokumentováno všechno. Nahlédněte do příručky na konci knihy, která obsahuje kompletní seznam modulu `string`.

Funkce string.split a string.join

Je téměř jisté, že každý, kdo pracuje s řetězci, shledá funkce `string.split` a `string.join` neocenitelnými. Obě jsou navzájem inverzní: funkce `string.split` vezme řetězec a vrátí seznam podřetězců, funkce `string.join` vezme seznam řetězců a spojí je do jediného, nového řetězce. Tyto dvě funkce používají typicky pro oddělování řetězců, které mají spojit nebo rozdělit mezery, což však lze změnit nepovinným argumentem.

Funkce *`string.join`* spojí více řetězců do jednoho.

Zřetězení řetězců pomocí '+' je užitečné, avšak neefektivní pro spojování velkého množství řetězců do jednoho řetězce, neboť při každém použití operátoru '+' dojde k vytvoření nového objektu. Náš předchozí příklad „Ahoj, světe" vytvořil dva řetězcové objekty, kde jeden z nich můžeme okamžitě odložit. Lepší volbou je použití funkce `join`.

```
>>> import string
>>> string.join(["funkce", "join", "vloží", "mezi", "prvky",
... "standardně", "mezery"])
⇒   'Funkce join vloží mezi prvky standardně mezery'
```

Předáním druhého nepovinného argumentu do funkce `join` můžete mezi spojované řetězce vložit libovolné znaky:

```
...
>>> string.join(["Oddělení", "prvku", "dvojtečkami"], "::")
⇒   'Oddělení::prvku::dvojtečkami'
```

Funkce *`string.split`* rozdělí řetězce na části.

Nejběžnějším použitím funkce `string.split` je pravděpodobně velmi jednoduchý dělicí mechanismus oddělených řetězcových záznamů uložených v textových souborech. Standardně funkce `split` rozděluje podle prázdných míst, tedy nikoli na základě znaku mezery, ale při předání druhého nepovinného argumentu i na základě konkrétní sekvence:

```
>>>
>>> import string
>>> string.split("Můžete \t\t smíchat \t\n \t tabulátory a " \
                 nové řádky \n\n")
```

```
⇒  ['Můžete', 'smíchat', 'tabulátory', 'a', 'nové', 'řádky']
>>> string.split("Mississippi", "ss")
⇒  ['Mi', 'i', 'ippi']
```

Funkce `string.join` a `string.split` nejsou úplně přesně inverzní; pokud je druhý argument vynechán, funkce `string.split` rozdělí řetězec za každým prázdným místem (ne jen nutně za jednoduchou mezerou), avšak funkce `string.join` spojí řetězce a mezi ně vsune znak jediné mezery.

Funkcí *string.split* lze rozdělit řetězec na předem určený počet podřetězců.

Někdy je užitečné umožnit, aby poslední pole spojovaného řetězce obsahovalo libovolný text třeba i včetně podřetězců, které by se mohly shodovat s tím, na základě čeho funkce `string.split` při načítání těchto dat řetězce rozděluje. To lze provést určením, kolik dělení řetězce má funkce `string.split` provést, a to předáním třetího nepovinného argumentu. Pokud zadáte *n* rozdělení, potom funkce `string.split` postupně zpracovává vstupní řetězec a provede buď *n* dělení vstupního řetězce (vytvoří seznam obsahující *n+1* podřetězců), nebo skončí vyčerpáním řetězce. Zde je několik příkladů:

```
>>> import string
>>> string.split('a b c d', ' ', 1)
⇒  ['a', 'b c d']
>>> string.split('a b c d', ' ', 2)
⇒  ['a', 'b', 'c d']
>>> string.split('a b c d', ' ', 9)
⇒  ['a', 'b', 'c', 'd']
```

Použijete-li třetí nepovinný argument funkce, musíte zadat i druhý. Aby se dosáhlo dělení po prázdných místech pomocí třetího argumentu, potom jako druhý argument použijte hodnotu „`None`“.

Funkce `string.join` a `string.split` používají autoři této knihy osobně velice intezivně – obvykle při práci s textovými soubory vytvořenými v jiných programech. Měli byste však vědět, že jste-li schopni nadefinovat svůj vlastní souborový formát dat pro použití výhradně v programech v Pythonu, pak existuje mnohem lepší alternativa pro ukládání dat v podobě textových souborů. Tuto možnost probereme později v oddíle, v němž budeme hovořit o modulu `cPickle`.

Převod řetězců na čísla

Funkce *string.atoi*, *string.atol* a *string.atof* mohou převádět řetězce na čísla, ale...

Funkce `string.atoi`, `string.atol` a `string.atof` lze využít k převádění řetězců na celá čísla, dlouhá celá čísla nebo čísla s pohyblivou desetinnou čárkou. Pokud je těmto funkcím předán řetězec, který nelze interpretovat jako číslo, vyvolají výjimku `ValueError`. Výjimky jsou podrobněji rozebrány v kapitole 14. Kromě toho může být funkcím `string.atoi` nebo `string.atol` předán třetí nepovinný argument, který udává, jakou číselnou soustavu použít při interpretaci vstupního řetězce.

```
>>>
>>> string.atof('123.456')
⇒  123.456
>>> string.atof('xxyy')
```

```
Traceback (innermost last):
   File "<stdin>", line 1, in ?
ValueError: invalid literal for atof(): xxyy
>>> string.atoi('3333')
⇒  3333
>>> string.atoi('123.456')
Traceback (innermost last):
   File "<stdin>", line 1, in ?
ValueError: invalid literal for atof(): 123.456
>>> string.atoi('10000', 8)
⇒  4096
>>> string.atoi('123456', 6)
Traceback (innermost last):
   File "<stdin>", line 1, in ?
ValueError: invalid literal for atoi: 123456
```

- **Celá čísla nemohou mít desetinnou čárku.**
- **Intepretuje číslo 10000 jako osmičkové číslo.**
- **Nelze intepretovat číslo 123456 v šestkové soustavě.**

Pochopili jste důvod poslední výjimky? Chtěli jsme, aby byl řetězec převeden na číslo v šestkové soustavě, jenže v ní se číslice 6 nemůže nikdy objevit! Jak rafinované!

... *int*, *long* a *float* jsou pravděpodobněji vhodnější.

Operace konverze řetězců na čísla potřebujete znát, protože musíte-li pracovat s řetězci v osmičkové, šestnáctkové nebo jiných soustavách, představují pro tyto účely správnou cestu. Mám-li však převádět řetězce na čísla, dávám přednost použití vestavěných funkcí Pythonu: `int`, `long` a `float`. Jsou snazší na zapamatování a poskytují stejnou funkčnost jako funkce `string.atoi`, `string.atol` a `string.atof`. Funkce `int` a `long` ovšem pracují pouze s čísly v desítkové soustavě.

Jak se zbavit přebytečných prázdných míst

Funkce *strip* odstraní přebytečné mezery na koncích řetězců.

Existuje trojice jednoduchých funkcí, které jsou překvapivě užitečné: `string.strip`, `string.lstrip` a `string.rstrip`. Funkce `string.strip` vezme řetězec jako argument a vrátí nový řetězec shodný s originálem, s tím rozdílem, že z něj odstraní všechny přebytečné mezery na jeho konci i na začátku. Funkce `string.lstrip` a `string.rstrip` fungují stejně s tím rozdílem, že odstraní přebytečné mezery buď na začátku, nebo konci řetězce (tedy na jeho levé nebo pravé straně):

```
...
>>> import string
>>> string.strip("  Ahoj,    svete\t\t ")
⇒  'Ahoj,    světe'
>>> string.lstrip("  Ahoj,    svete\t\t ")
⇒   'Ahoj,    světe\011\011 '
>>> string.rstrip("  Ahoj,    svete\t\t ")
⇒   '  Ahoj,    světe'
```

Python považuje za „přebytečné mezery" všechny znaky, uložené v konstantě *string.whitespace*.

V tomto příkladu jsou znaky tabulátoru považovány za přebytečné mezery. Přesný význam, co je považováno za přebytečnou mezeru, se může lišit podle druhu používaného operačního systému, ale co za ně považuje Python, to lze zjistit zpřístupněním konstanty `string.whitespace`. Na mém počítači s Windows jde například o následující znaky:

```
...
>>> string.whitespace
⇒  '\011\012\013\014\015 '
>>> "\t\n\v\f\r "
⇒  '\011\012\013\014\015 '
```

Znaky dané zpětným lomítkem a trojicí osmičkových čísel (\nnn) představují znak pro tabulátor, nový řádek a tak dále. Mezera je představována sama sebou, hned za číslicí '5' v řetězci \015. Možná vás to bude svádět k měnění těchto hodnot, abyste viděli, jak to ovlivní funkci `string.split` a další. Nedělejte to. Je jisté, že výsledkem takového pokusu bude něco zcela jiného, než to, co hledáte.

Nejběžnějším použitím těchto funkcí je rychlé vyčištění řetězců, jež byly právě načteny. To je konkrétně užitečné při načítání řádku z externího souboru (bude probráno v dalších kapitolách), protože Python vždy načte obsah celého řádku najednou, včetně případného znaku nového řádku, vyskytne-li se. Poté, co se dostanete ke zpracování načtených řádků, nebudete typicky tento ukončovací řádek potřebovat. Funkce `string.rstrip` je nejvhodnější způsobem, jak se ho zbavit.

Hledání řetězců

Modul `string` obsahuje řadu funkcí, které provádějí jednoduché hledání řetězců. Než je popíšeme, dovolte, abychom krátce pohovořit o dalším modulu Pythonu, o modulu `re`, který podrobně popíšeme v kapitole „Regulární výrazy" (kapitola 21).

Modul *re* poskytuje mnohem výkonnější funkce než modul '*string*'.

Modul `re` také provádí hledání řetězců, ale mnohem pružnějším způsobem. Místo hledání jednoho zadaného podřetězce hledá modul `re` řetězcové vzory. Umožňuje například hledat podřetězce, které se skládají výhradně z číslic.

Proč se zde o tom zmiňujeme, když se budeme modulem `re` zabývat později? Dle mých zkušeností mnoho použití funkcí pro prohledávání řetězců, hlavně těch v modulu `string`, je nevhodných. Programátor by měl prospěch z těchto mnohem výkonnějších prohledávacích mechanismů, ale vůbec neví, že takové existují, a proto ani nehledá něco lepšího. Výjimku lze připustit snad jedině v případě, že řešíte naléhavý projekt, ve kterém se pracuje s řetězci a nemáte čas číst celou knihu. Měli byste si však být vědomi, že takové mnohem výkonnější alternativy existují.

V modulu `string` jsou čtyři základní prohledávací funkce, které jsou si velmi podobné: `string.find`, `string.rfind`, `string.index` a `string.rindex`. K nim ještě patří příbuzná funkce `string.count`, která jednoduše spočítá, kolikrát se daný podřetězec vyskytuje v jiném řetězci. Funkci `find` rozebereme podrobněji, a pak si popíšeme, jak se od ní ostatní funkce liší.

Pozici podřetězce v řetězci naleznete funkcí *string.find*.

Funkce `string.find` má dva argumenty: první z nich, *string*, je prohledávaný řetězec, a druhý *substring*, je podřetězec, jehož výskyt hledáme. Funkce `string.find` vrací pozici prvního znaku podřetězce *substring* v řetězci

string nebo vrací hodnotu –1, jestliže se podřetězec *substring* v řetězci *string* nenachází.

```
...
>>> import string
>>> string.find("Mississippi", "ss")
⇒  2
>>> string.find("Mississippi", "zz")
⇒  -1
```

Funkce `find` může také mít jeden nebo dva další nepovinné argumenty. Prvním z nich, pokud je uveden, je celé číslo *start*, díky němuž funkce `find` bude ignorovat všechny znaky, které leží před pozicí *start* v řetězci *string*. Druhý argument, pokud je zadán, je celé číslo *end*, které způsobí, že funkce `find` ignoruje všechny znaky, které leží na nebo za pozicí *end* v řetězci.

```
>>>
>>> string.find("Mississippi", "ss", 3)
⇒  5
>>> string.find("Mississippi", "ss", 0, 3)
⇒  -1
```

Funkce `string.rfind` je takřka stejná jako funkce `find` s tím rozdílem, že prohledávání začíná na konci řetězce *string*, takže vrací pozici prvního znaku posledního výskytu podřetězce *substring* v řetězci *string*.

```
...
>>> string.rfind("Mississippi", "ss")
⇒  5
```

Funkce `string.rfind` může také přijímat jeden nebo dva nepovinné argumenty, které mají stejný význam jako argumenty funkce `string.find`.

Funkce `string.index` a `string.rindex` jsou identické s funkcemi `string.find` a `string.rfind`, tedy až na jedinou výjimku: pokud funkce `string.index` a `string.rindex` nenajdou výskyt podřetězce *substring* v řetězci *string*, nevrátí hodnotu –1, ale vyvolají výjimku `ValueError`. Co to přesně znamená, se dozvíte po přečtení kapitoly „Výjimky“.

Použití funkce `string.count` je identické s předchozími čtyřmi funkcemi s tím rozdílem, že vrací počet výskytů nepřekrývajících se podřetězců v daném řetězci:

```
>>> string.count("Mississippi", "ss")
⇒  2
```

Úpravy řetězců

Řětezce jsou neměnitelné, avšak modul `string` nabízí řadu funkcí, jejichž argumentem je řetězec a které vrací nový řetězec, jenž je modifikovanou verzí řetězce v argumentu. To má ve většině případů stejný efekt jako přímá modifikace řetězce. Kompletnější popis těchto funkcí naleznete v referenční části na konci knihy.

Další funkce modulu *string* provádějí náhradu podřetězce,...

Funkci `string.replace` lze použít k nahrazení podřetězce *substring* (její druhý argument) v řetězci *string* (její první argument) podřetězcem *newstring* (její třetí argument). Má také čtvrtý nepovinný argument. Podrobnosti viz referenční část na konci knihy.

```
>>>
    >>> string.replace("Mississippi", "ss", "+++")
    ⇒   'Mi+++i+++ippi'
```

Stejně jako v případě funkcí pro vyhledávání poskytuje modul `re` mnohem výkonnější metody pro nahrazování částí řetězce.

... překlad znaků na základě překladové tabulky,...

Funkce `string.maketrans` a `string.translate` lze společně použít k překladu znaků v řetězci na jiné znaky. Ačkoliv se používají zřídka, mohou vám tyto funkce, přijde-li na ně řada, zjednodušit život.

Řekněme, že pracujete na programu, který překládá řetězcové výrazy z jednoho počítačového jazyka do druhého. První jazyk používá pro logickou negaci operátor '~', zatímco druhý používá '!'; první jazyk používá pro logický součin operátor '^', kdežto druhý jazyk používá '&'; a první jazyk používá '(' a ')' , kdežto druhý jazyk používá '[' a ']'. V dané řetězcovém výrazu tedy potřebujete nahradit všechny znaky '~' znakem '!', všechny výskyty znaku '^' znakem '&' a , všechny výskyty znaku '(' znakem '[' a všechny znaky ')' znakem ']'. To lze provést opakovaným voláním funkce `string.replace`, ale mnohem snazším a jednodušším způsobem je:

```
...
    >>> table = string.maketrans("~^()", "!&[]")
    >>> string.translate("~x^(y % z)", table)
    ⇒   '!x & [y % z]'
```

V prvním řádku jsme použili funkci `string.maketrans`, která ze svých dvou argumentů vytvoří převodní tabulku. Oba argumenty musí obsahovat stejný počet znaků a tabulka musí být uspořádána tak, aby vyhledáním *n*-tého znaku prvního argumentu v této tabulce funkce vracela *n*-tý znak druhého argumentu.

Poté je tato tabulka vytvořená funkcí `string.maketrans` spolu s překládaným řetězcem předána funkci `string.translate`, ta projde každý znak řetězce v jeho prvním argumentu a zkontroluje, zda má tento znak svůj protějšek v tabulce dané řetězcem v druhém argumentu. Pokud je znak v převodní tabulce nalezen, pak funkce `string.translate` nahradí tento znak odpovídajícím znakem nalezeným v překladové tabulce a vytvoří tak přeložený řetězec.

Funkci `string.translate` může být předán nepovinný argument, určující znaky, které mají být z řetězce zcela odstraněny.

...a konverze velkých a malých písmen, rozšíření pomocí tabulátorů a zarovnání textu.

Další funkce v modulu `string` provádějí mnohem specializovanější úlohy. Funkce `string.lower` převede řetězec alfanumerických znaků na řetězec obsahující pouze malá písmena, zatímco funkce `string.upper` provádí její opak, převádí malá písmena na velká. Funkce `string.capitalize` převede první písmeno řetězce na velké, funkce `string.capwords` převede znaky všech slov řetězce na velké a funkce `string.swapcase` zamění v řetězci všechna malá

písmena za velká a velká na malá. Funkce `string.expandtabs` vypustí z řetězce všechny znaky tabulátoru zadaným počtem mezer, funkce `string.ljust`, `string.rjust` a `string.center` doplní řetězec mezerami tak, aby jeho velikost dosáhla požadované délky. Funkce `string.zfill` doplní číselný řetězec nulami zleva. Podrobnosti k těmto funkcím najdete v referenční části.

Úpravy řetězců prostřednictvím seznamů

Speciálnější operace s řetězci vyřešíte konverzí na seznamy

Protože řetězce jsou nemodifikovatelnými objekty, neexistuje způsob, jak s nimi přímo manipulovat, jako třeba se seznamy. Zatímco operace, které z řetězců vytvářejí řetězce nové (a ponechají přitom originální řetězce nezměněny) jsou užitečné k mnoha účelům, budete někdy chtít manipulovat s řetězci tak, jako kdyby šlo o seznamy znaků. V takovém případě jen převeďte řetězec na seznam znaků, proveďte s ním, co potřebujete a výsledný seznam převeďte zpět na řetězec:

```
...
>>> import string
>>> text = "Ahoj, světe"
>>> slovaSeznamu = list(text)
>>> slovaSeznamu[5:] = []
>>> slovaSeznamu.reverse()
>>> text = string.join(slovaSeznamu, "")
>>> print text
⇒   ,johA
```

- **Odstraň vše za čárkou**
- **Spoj dohromady znaky bez mezer.**

Ačkoliv k převedení řetězce na seznam znaků lze použít funkci `string.split`, jednodušší, nejen k zapamatování, je použít funkci pro konverzi typu **`list`** (a co je ještě užitečnější, lze navíc převést řetězec na n-tici pomocí interní funkce **`tuple`**). Pro zpětný převod seznamu na řetězec použijte funkci `string.join`.

Operace s řetězci prováděné tímto způsobem jsou z hlediska výpočetního výkonu relativně drahé, takže byste tyto metody neměli využívat přehnaně. Zpracování stovek či tisíců řetězců tímto způsobem pravděpodobně nebude mít na váš program velký dopad, ovšem zpracování milionů řetězců už ano.

Užitečné konstanty

Některé užitečné řetězcové konstanty

V modulu `string` jsou také definovány určité užitečné konstanty. Již jsme viděli konstantu `string.whitespace`, která je tvořena řetězcem znaků, jež Python považuje za prázdný prostor. Konstanta `string.digits` je řetězec '`0123456789`', konstanta `string.hexdigits` zahrnuje všechny znaky řetězce `string.digits` doplněné o znaky '`abcdefABCDEF`', tedy znaky, které jsou používány pro zápis šestnáctkových čísel. Konstanta `string.octdigits` obsahuje řetězec '`01234567`', osmičková čísla používají jen tyto znaky. Konstanta `string.lowercase` obsahuje všechna malá písmena, `string.uppercase` obsahuje všechna velká písmena; konstanta `string.letters` obsahuje všechny znaky konstant `string.lowercase` a `string.uppercase`.

Zapamatujte si, že řetězce jsou sekvencemi znaků, takže u nich můžete vhodně použít pythonový operátor `in`, jímž zjistíte, zda se daný znak nachází v některém z těchto řetězců.

Možná budete v pokušení přiřadit něco těmto konstantám, a tím i změnit chování jazyka. Python vám to sice umožní, ale bude to asi opravdu nevhodný nápad.

Převod objektů na řetězce

!!!

Na řetězec lze převést cokoliv.

V Pythonu lze převést na nějaký typ řetězce pomocí vestavěné funkce `repr` prakticky cokoliv. Seznamy jsou jediným složitým objektem Pythonu, se kterým jsme se zatím nesetkali, takže nyní převeďme nějaké seznamy na řetězce:

```
>>>
    >>> repr([1, 2, 3])
    ⇒   '[1, 2, 3]'
    >>> x =[1]
    >>> x.append(2)
    >>> x.append[(3, 4)]
    >>> repr(x)
    ⇒   '[1, 2, [3, 4]]'
```

Funkce '*objekt*' vrací objekt ve formě řetězce.

Schopnost jazyka získat řetězcovou reprezentaci takřka libovolného objektu je natolik užitečná, že jazyk Python pro ni zavádí zkratku, tzv. *zpětný apostrof*. Vložení výrazu mezi zpětné apostrofy funguje stejně jako volání funkce `repr`:

```
>>>
    >>> x = [1, 2, 3]
    >>> retezec = "Seznam " + `x` + " má " + `len(x)` + " prvky."
    >>> retezec
    ⇒   'Seznam [1, 2, 3] má 3 prvky.'
```

Všimněte si rozdílu mezi zpětnými (`x`) a normálními apostrofy ('x'). Uzavřením položky do normálních apostrofů vznikne řetězec tvořený pouze jeho jménem.

V uvedeném příkladu jsme použili ` ` pro převod seznamu `x` a funkce zjišťující délku řetězce `len(x)` na řetězec, který byl zřetězen s ostatními řetězci do konečného řetězce. Bez použití znaků ` ` by to nefungovalo. Výrazem, jako `"string" + [1, 2] + 3` se snažíme přidat řetězec, přidat seznam nebo přidat čísla? Python v takovém případě netuší, co chcete udělat a udělá to, co uzná za nejbezpečnější, a raději vyvolá výjimku než by se něco domníval. V předešlém případě bylo nutno všechny prvky před jejich spojením zkonvertovat na řetězce.

Zatím byly seznamy jediným „složitým" objektem, který jsme si dosud popsali, ale použitím `` lze obdržet určitý druh řetězcové reprezentace skoro každého pythonového objektu. Abyste to viděli, zkuste do zpětných apostrofů `` sami

uzavřít nějaký složitý vestavěný objekt jazyka – konkrétní funkci Pythonu. Vyzkoušejte třeba toto:

```
>>>
    >>> `len`
    ⇒   '<built-in function len>'          # interní funkce len
```

Python nevytvořil řetězec obsahující kód, jímž je funkce `len` implementována, ale vrátil alespoň řetězec, `<built-in function len>`, který popisuje, o jakou funkci se jedná. Budete-li si operátor `` pamatovat a vyzkoušíte jej na každém datovém typu Pythonu (slovníky, n-tice, třídy apod.) tak, jak jsme to udělali v knize, uvidíte, že bez ohledu na typ objektů Pythonu vždy obdržíte řetězec, který vám o tomto objektu něco řekne.

To je skvělé pro ladění programu. Pokud si nejste jisti, jaký je v určitém okamžiku běhu programu obsah té které proměnné, stačí její jméno uzavřít do `` a vytiskne se její aktuální obsah.

Funkce *repr* vrací úplnou, formální reprezentaci objektu; to se vám nemusí pro ladění hodit

Již jsme zjistili, jak Python umí převádět objekty na řetězce, které popisují tento objekt. Pravdou je, že Python to může provést jedním ze dvou různých způsobů. Funkce `repr` vždy vrací hodnotu, kterou můžeme nepřesně nazvat jako *formální řetězcovou reprezentaci* objektu Pythonu. Konkrétněji řečeno, metoda `repr` vrací řetězcovou reprezentaci objektu, z níž lze původní objekt znovu obnovit. U velkých, složitých objektů to není zrovna věc, kterou byste chtěli ve výpisech debuggingu a hlášeních o stavu vidět.

Funkce *str* vrací reprezentaci objektu v čitelnější formě.

Pro získání lidštější reprezentace objektu nabízí Python vestavěnou funkci `str`, jež vrací řetězec, který bychom mohli nazvat *neformální řetězcovou reprezentací*. Řetězec, který vrátí funkce `str`, nemusí definovat objekt jako celek. Je určen pro uživatele, ne pro Python.

Mezi funkcemi `repr` a `str` si při jejich prvním použití nepovšimnete žádného rozdílu, dokud nezačnete využívat objektově orientovaných vlastností Pythonu, protože do té doby mezi nimi opravdu žádné rozdíly nejsou. Ve skutečnosti funkce `str` aplikovaná na libovolný vestavěný objekt Pythonu volá funkci `repr`, aby vypočítala její výsledek. Rozdíl mezi nimi začne být důležitý až v okamžiku, kdy začnete definovat své vlastní třídy. Tím se budeme zabývat v knize později.

Proč tedy o tom hovoříme již nyní? V podstatě jsme jen chtěli, abyste si byli vědomi, že se „za scénou“ děje mnohem více věcí, než je na první pohled patrné, a že metoda `str` umí při testování a hledání chyb v programu mnohem více než prostý příkaz výstupu `print`. Je jen otázkou dobrého vkusu, zda budete při vytváření řetězců pro výpis informací dávat přednost funkci `str` před funkcí `repr` nebo znaky ``.

Formátování řetězců

Znak % slouží k přesnému formátování textu pro výstup.

V tomto oddíle probíráme jednoduchý řetězcový operátor Pythonu, operátor *řetězcový modulus* %. Slouží ke spojování pythonových hodnot do formáto-

vacích řetězců pro tisk nebo pro jiné použití. Uživatelé jazyka C jistě najdou podobnosti se skupinou příkazů `printf`.

Příklad:

```
>>>
    >>> "%s je %s %s" % ("Ambrosie", "pokrmem", "bohů")
    ⇒   'Ambrosie je pokrmem bohů'
```

Operátor % nahrazuje podřetězce pro formátovací sekvence v delším řetězci.

Operátor řetězcový modulus reprezentovaný znakem procento (znak % vytištěný tučně, ne znaky % na levé straně v předchozím příkladu) se skládá ze dvou částí; z levé strany, kterou tvoří řetězec, a pravé strany, která je tvořena proměnnou typu n-tice. Operátor řetězcového modulu prozkoumá řetězec na levé straně, zda se v něm vyskytují speciální *formátovací sekvence,* a vytvoří nový řetězec, v němž budou tyto formátovací sekvence nahrazeny hodnotami z jeho pravé strany. V tomto příkladu jsou formátovacími sekvencemi na levé straně tři znaky „%", které mající význam „zde vlož řetězec".

Předáním různých hodnot na pravou stranu se vytvoří různé řetězce:

```
>>>
    >>> "%s je %s %s" % ("Nektar", "napojem", "bohů")
    ⇒   'Nektar je napojem bohů'
    >>> "%s %s je %s pro %s" % ("Bruselský", "pstruh", "jídlem", "šílence")
    ⇒   'Bruselský pstruh je jídlem pro šílence'
```

Na prvky n-tice na pravé straně je díky zápisu "`%s`" automaticky aplikována funkce `str`, což znamená, že ony samy už nemusí být řetězci:

```
>>> x = [1, 2, "tři"]
>>> "%s obsahuje: %s" % ("Seznam", x)
⇒   "Seznam obsahuje: [1, 2, 'tři']"
```

Použití formátovacích sekvencí

Formátovací sekvence mohou určovat šířku pole, zarovnání textu a počet desetinných míst.

Všechny formátovací sekvence jsou vlastně podřetězce obsažené v řetězci na levé straně od ústředního operátoru „%". Každá formátovací sekvence začíná znakem procenta a je doplněna jedním nebo více znaky, které udávají, čím bude sekvence nahrazena a jak bude nahrazení provedeno. Výše použitá formátovací sekvence „%s" je nejjednodušším typem sekvence a říká, že namísto řetězce zastoupeného symbolem „%s" bude dosazen odpovídající řetězec z prvku n-tice na pravé straně od ústředního operátoru „%".

Jiné formátovací sekvence mohou být podstatně složitější. Tato udává, aby délka (celkový počet znaků) tištěného řetězce byla 6, počet čísel za desetinnou čárkou (tečkou) byl 2 a výsledek byl zarovnán vlevo. Do řetězce jsme vložil znaky pro ostré závorky (znaky < >), abyste viděli, kam byly při formátování řetězce vloženy mezery:

```
>>> "Pí je <%-6.2f>" % 3.14159 # použití formátovací sekvence: %-6.2f
⇒   'Pí je <3.14  >'
```

Všechny možnosti práce se znaky, které jsou ve formátovacích sekvencích dovoleny, jsou uvedeny v referenci na konci knihy. Najdete jich tam několik, použití žádné z nich není obtížné. Pamatujte, že si formátovací sekvence můžete v Pythonu vždy vyzkoušet v interaktivním režimu a sami se přesvědčit, zda program dělá to, co jste očekávali.

Pojmenované parametry a formátovací sekvence

Nahrazení formátovacích sekvencí lze provést pojmenováním.

Nakonec se dostáváme ještě k jedné vlastnosti jazyka dostupné pomocí operátoru řetězcového modulu (%), která může být za jistých okolností docela užitečná. Abychom si ji mohli popsat, budeme bohužel nuceni použít jednu z dalších vlastností jazyka Python, o které jsme se ještě nezmínili, a to slovníky v jiných jazycích, běžně nazývané hashovacími tabulkami, nebo též asociativními poli. Můžete přejít na další kapitolu „Slovníky“, abyste se o slovnících dozvěděli více, nebo tento odstavec přeskočit a vrátit se k němu později, nebo si jej pročíst a důvěřovat příkladům, které vám mají tuto problematiku osvětlit.

Formátovací sekvence nemusejí udávat, co se má čím nahradit, jen svou pozicí, ale také jménem. Při použití této formy má každá formátovací sekvence jméno uzavřené do závorek hned za znakem %, jak ukazuje příklad:

```
>>> "%(pi).2f"  # Povšimněte si jména v závorce, níže uvedený text je vytištěn tučně.
```

Kromě toho argument na pravé straně operátoru % již není pouhou hodnotou nebo n-ticí hodnot, které mají být vytištěny, ale celým slovníkem hodnot, kde každá formátovací sekvence má ve slovníku svůj odpovídající pojmenovaný klíč. Pomocí výše uvedené formátovací sekvence s řetězcovým modulus operátorem % lze tedy vytvořit kód jako je tento (jeho nové části – jméno formátovací sekvence a slovník – jsou zde uvedeny tučně):

```
>>> dict = { 'e' : 2.718, 'pi' : 3.14159 }
>>> print "%(pi).2f - %(pi).4f - %(e).2f" % dict
⇒   3.14 - 3.1416 - 2.72
```

Jde o zvláště užitečné použití řetězcového operátoru modulus s řetězci, které provedou velký počet záměn, protože se již nemusíme starat o sledování pozic formátovacích sekvencí a prvků n-tic na pravé straně. Na pořadí, ve kterém jsou prvky ve slovníku `dict` definovány, nezáleží, v šabloně řetězce lze používat hodnoty ze slovníku `dict` více než jednou (jak je tomu i s položkou '`pi`').