

Obróbka tekstu w Pythonie

WĄŻ W STRINGACH

http://www.sxc.hu

W bieżącym numerze „Linux Magazine” rozpoczynamy cykl artykułów na temat Pythona.

Pierwszy z nich poświęcony będzie pracy z tekstem.

ROBERT JANUSZKIEWICZ

Dlaczego Python?

Dokonując wyboru języka programowania dla danego projektu, zawsze idziemy na kompromis, dokonawszy swobodnego rachunku zysków i strat. Bardzo często decyzja jest prosta – determinuje ją doświadczenie programistów, charakter projektu, zewnętrzne wymagania i ograniczenia itp. Często dany język wybieramy ze względu nie na jego cechy, lecz – jak na przykład w przypadku PHP – z uwagi na łatwość tworzenia mniej lub bardziej złożonych aplikacji webowych i mnogość dostępnych bibliotek i rozszerzeń.

Kiedy możemy sobie pozwolić na wybór Pythona? Zawsze tam, gdzie zależy nam na szybkim tworzeniu aplikacji, łatwości wprowadzania zmian i czytelności kodu. Do dyspozycji mamy bogaty zestaw bibliotek wspomagających tworzenie wszelkiego typu programów. Python sprawdza się przede wszystkim w sytuacjach, gdy tworzymy oprogramowanie na wewnętrzne potrzeby naszej organizacji: aplikacje do pobierania, przetwarzania i analizy danych itp. Jeśli jednak pracujemy nad aplikacją, która ma być rozpowszechniana wśród użytkowników, czeka nas wiele wyzwań – zwłaszcza związanych z niedopracowaniem odpowiednich bibliotek odpowiedzialnych za graficzny interfejs użytkownika. Dlatego na przykład Python jest intensywnie wykorzystywany wewnątrz Google, jednak firma ta, udostępniając popularne oprogramowanie dla zwykłych użytkowników (takie jak Earth), wybiera bardziej tradycyjne rozwiązania (nawiasem mówiąc, kontrola Google Earth z poziomu Py-

thona jest bardzo przyjemnym i prostym zadaniem). Warto o tym pamiętać, bowiem wielu użytkowników poznając nowy, interesujący język, ma tendencję do wykorzystywania go do wszystkiego, bez względu na okoliczności.

Python znajduje się w praktycznie każdej współczesnej dystrybucji Linuksa. Oprócz głównego pakietu do dyspozycji mamy zwykle dodatkowe moduły – ich liczba i nazewnictwo zależy od konkretnej dystrybucji. W Debianie (a więc i w Ubuntu) jest to *python-NazwaModulu-NumerWersji_Architektura.deb*, przy czym po ciągu *python* może opcjonalnie wystąpić numer wersji Pythona, z którą współpracuje dany moduł. Ze względu na popularność zastosowań Pythona pod Linuksem, najprawdopodobniej jest on już zainstalowany w naszej dystrybucji.

Skrypty Pythona możemy zapisywać w plikach z rozszerzeniem *.py* i uruchamiać poleceniem:

```
python nazwa_skryptu.py
```

Możemy też nadać im prawo do wykonywania i w pierwszym wierszu umieścić ciąg `#!/usr/bin/env python`. Jednak eksperymentując z nową funkcjonalnością, możemy użyć samego interpretera Pythona, dysponującego znakomitym trybem interaktywnym. Jest on tak wygodny, że wielu użytkowników Linuksa korzysta z niego w roli podręcznego kalkulatora. Istotna uwaga: w skryptach Pythona (w 2., ewentualnie w 1. wierszu) powinniśmy umieścić ciąg `#!/usr/bin/env python` w prze-

ciwnym wypadku napotkamy na problemy przy przetwarzaniu znaków spoza zestawu ASCII.

Łańcuchy – podstawy

Praca z łańcuchami znakowymi („napisami”) w Pythonie to czysta przyjemność. W poniższym przykładzie zmiennej *a* przypisujemy wartość *Python – elegancki gad*:

```
>>> a = "Python – elegancki gad"
>>> print a
Python – elegancki gad
```

W roli ogranicznika łańcucha znakowego możemy użyć zarówno zwykłego cudzysłowu (”), jak i apostrofu ('). Dwa napisy możemy dodać na przykład za pomocą operatora +:

```
>>> a = a + "!"
>>> print a
'Python – elegancki gad!'
```

Napisy możemy nie tylko dodawać, ale i mnożyć:

```
>>> a = a + 3 * "\nI zwinny!"
>>> print a
Python – elegancki gad!
I zwinny!
I zwinny!
I zwinny!
```

Dostęp do poszczególnych znaków napisu uzyskujemy poprzez indeks zawarty w na-

Python – podstawy

Python stawia na prostotę. Jedną z cech składni jest korzystanie z wcięć zamiast nawiasów klamrowych do wyróżniania bloków logicznych. Przyjrzyjmy się poniższemu przykładowi:

```
f a == 1:
    print "a=1!"
else:
    print "Niemożliwe"
```

Jest to typowy kod Pythona, łatwy do odczytania, przejrzysty i prosty. Funkcje definiujemy słowem kluczowym *def*:

```
def dodaj (a, b):
    return a+b
```

Standardowej iteracji dokonujemy na przykład za pomocą pętli *for*, przekazując jej explicite poszczególne elementy:

```
for n in [1,2,3]:
    print n
```

Możemy też przekazać jej zakres słowem kluczowym *range()*:

```
for i in range (10):
    print i
```

Pętli *while* używamy równie prosto:

```
a = 1
while a<10:
    print a
    a = a+1
```

wiasach kwadratowych. Pierwszy znak napisu to *napis[0]*, drugi to *napis[1]* itd.:

```
>>> print a[0], a[2]
P t
```

Nie musimy bynajmniej ograniczać się do pojedynczych znaków – do dyspozycji mamy całe zakresy, które oznaczamy dwukropkiem. Poniższa instrukcja wyświetli ciąg od trzeciego do siódmego znaku napisu:

```
>>> print a[2: 6]
thon
```

W roli indeksów mogą też występować liczby ujemne: *napis[-1]* to przedostatni znak ciągu, *napis[-2]* – drugi od końca itd. Wyświetli pięć ostatnich znaków napisu:

```
>>> print a[-5:]
inny!
```

Równie prosta jest zamiana małych liter na wielkie:

```
>>> a = "Python"
>>> a.upper()
'PYTHON'
```

W analogiczny sposób zamieniamy wielkie litery na małe metodą *lower()*. Możemy też zamienić na wielką tylko pierwszą literę danego napisu funkcją *capitalize()*. Nasz łańcuch możemy z łatwością przekształcić w listę:

```
>>> a = "Python"
>>> list (a)
['P', 'y', 't', 'h', 'o', 'n']
```

Czasem zależy nam na eleganckim wyrównaniu danego napisu – do prawej (*rjust*), do lewej (*ljust*), bądź na wyśrodkowaniu go (*center*):

```
>>> print "|", a.center (20), "|"
| Python |
```

Czasem przeciwnie – chcemy pozbyć się zbędnych spacji, które znajdują się na po-

czątku lub na końcu napisu:

```
>>> a = " Python "
>>> a.strip()
'Python'
```

Łańcuchy możemy traktować też jak szablony, wstawiając w nich odnośniki do zmiennych oznaczane – podobnie jak w C – operatorem formatującym % (na przykład %s dla zmiennych łańcuchowych, %d dla zmiennych całkowitych):

```
a = "%s (%d) Pythony%s" %
("Dwa", 2, "!")
>>> a
'Dwa (2) Pythony!'
```

W przykładzie powyżej kolejne wystąpienia ciągów %s, %d i %s zostały zastąpione sekwencją (w nawiasach okrągłych).

Nieco dłużej trwa odwrócenie napisu – łańcuchy znakowe są bowiem w Pythonie typem niezmiennym. Musimy więc najpierw przekształcić łańcuch do postaci listy metodą *list()*, odwrócić kolejność poszczególnych elementów listy funkcją *reverse()*, a następnie połączyć elementy listy, przekształcając je z powrotem w łańcuch:

```
>>> a = "Python"
>>> lista = list (a)
>>> lista.reverse()
>>> lista = "".join (lista)
>>> print lista
nohtyP
```

Proste wyszukiwanie

Powyższe przykłady powinny dać pewne wyobrażenie o prostocie manipulacji napisami. Możemy teraz przejść do wyszukiwania ciągów znaków. Mamy tu do wyboru kilka możliwości; jeśli chcemy tylko sprawdzić, czy dany ciąg znajduje się w określonym napisie, możemy skorzystać operatora przynależności, *in*:

```
>>> a = "Python"
>>> "a" in a
False
>>> "Py" in a
True
```

Możemy też policzyć, ile razy dany ciąg występuje w napisie:

```
>>> a = "Python przez duże 'P'"
>>> a.count ("P")
```

Wyrażenia regularne

Wyrażenie	Opis
<code>a a</code>	<code>a</code> lub <code>a</code>
<code>.</code>	jakikolwiek znak z wyjątkiem nowego wiersza
<code>^</code>	początek wiersza
<code>\$</code>	koniec wiersza
<code>*</code>	0 lub więcej wystąpień poprzedzającego wyrażenia
<code>+</code>	1 lub więcej wystąpień poprzedzającego wyrażenia
<code>?</code>	0 lub 1 wystąpienie poprzedzającego wyrażenia
<code>{n}</code>	<code>n</code> wystąpień poprzedzającego wyrażenia
<code>{m, n}</code>	od <code>m</code> do <code>n</code> wystąpień poprzedzającego wyrażenia
<code>\d</code>	cyfra
<code>\w</code>	znak alfanumeryczny
<code>\s</code>	spacja, tabulator, znak końca linii/nowego wiersza

Generalnie jednak podnapiisy znajdujemy metodą `find()`. Przyjmuje ona trzy argumenty, z których dwa ostatnie (początek i koniec) są opcjonalne, zaś pierwszy jest wyszukiwanym ciągiem znaków. Funkcja `find()` zwraca tylko pierwsze wystąpienie danego ciągu; jeśli jest ich więcej, musimy użyć jej ponownie, zaczynając wyszukiwanie od kolejnego znaku. `-1` oznacza, że ciąg nie został znaleziony:

```
>>> a.find("P")
0
>>> a.find("P", 1)
19
>>> a.find("x")
-1
```

Wyrażenia regularne

Proste wyszukiwanie tekstu sprawdza się jedynie w trywialnych sytuacjach. W praktyce znacznie częściej potrzebujemy pomocy wyrażeń regularnych. W tym przypadku wyzwaniem jest takie skonstruowanie wyrażenia regularnego, by idealnie odpowiadało wyszukiwanym ciągom znaków; kwestia wykorzystania do tego celu Pythona czy Perla jest drugorzędna. Konstruowanie wyrażeń regularnych jest sztuką samą w sobie, której poświęcono kilka książek. W tabeli *Wyrażenia regularne* znajdziemy kilka najprostszych przykładów. Pamiętajmy, że jeśli chcemy dopasować jeden ze znaków specjalnych bądź ciągów wykorzystywanych do konstrukcji wyrażeń regularnych (na przykład kropkę), musimy poprzedzić go znakiem `\`.

Za obsługę wyrażeń regularnych w Pythonie odpowiada moduł `re`, który importujemy poleceniem `import re`. Po wydaniu tego polece-

nia mamy do dyspozycji cztery główne metody: `compile()`, `match()`, `search()` i `findall()`. Pierwsza z nich kompiluje wyrażenie regularne i zwraca odpowiedni obiekt. Druga próbuje odszukać dany wzorzec na początku ciągu; jeśli go znajdzie, zwraca znaleziony obiekt, w przeciwnym wypadku zwraca `None`. Trzecia różni się od drugiej tym, że odnajduje pierwsze wystąpienie bez względu na to, gdzie się znajduje. Z kolei `findall()` zwraca wszystkie (nie nachodzące na siebie) wystąpienia wzorca w danym łańcuchu. Oprócz tego dysponujemy użyteczną funkcją podmiany, `sub()`, która zamienia wszystkie wystąpienia wzorca w napisie na podany ciąg, oraz `split()`, dzielącą napis według podanego wzorca.

Przyjrzyjmy się, jak w praktyce wygląda korzystanie z funkcji `match()`:

```
>>> import re
>>> r = re.match
("P.t", "Python")
>>> r.group()
'Pyt'
```

Przede wszystkim importujemy moduł odpowiedzialny za obsługę wyrażeń regularnych, `re`. Następnie wywołujemy funkcję `match` należącą do tego modułu i sprawdzamy, czy na początku ciągu `Python` znajdzie się wyrażenie regularne `P.t`. Funkcja `group()` zwraca pasujący ciąg („grupe”). W analogiczny sposób używamy funkcji `search`, która odnajdzie wyrażenie regularne w napisie bez względu na miejsce jego wystąpienia:

```
>>> r = re.search ("
.t.", "Python")
>>> r.group()
```

```
'yth'
```

Powyższe można zapisać prościej jako:

```
>>> re.search(".t.", "Python").
group()
'yth'
```

Kolejny przykład:

```
>>> wyrazenie = "hun|han|hon"
>>> napis = "Python"
>>> re.search(wyrazenie, napis).
group()
'hon'
```

Użyliśmy w nim wyrażenia regularnego, które dopasuje jeden z trzech wzorców (`hun`, `han`, `hon`).

Metoda `findall()` działa identycznie, z tym że odnajduje wszystkie wystąpienia, nie tylko pierwsze:

```
>>> wyrazenie = "hun|han|hon"
>>> napis = "Python dla
pythonistów!"
>>> re.findall(wyrazenie, napis)
['hon', 'hon']
```

Bardziej interesująca jest podmiana, którą wykonujemy funkcją `sub()`:

```
>>> napis = "Python dla
pythonistów!"
>>> wyrazenie = "ython"
>>> print re.sub(wyrazenie,
"erl", napis)
Perl dla perlistów!
```

Podsumowanie

Mamy nadzieję, że tym krótkim i z konieczności dość uproszczonym przedstawieniem możliwości Pythona w zakresie pracy z napisami zachęciliśmy do bliższego przyjrzenia się temu interesującemu językowi. W kolejnym odcinku pokażemy, jak wykorzystacie zdobytą wiedzę w konkretnych zastosowaniach. ■

INFO

Polska strona poświęcona Pythonowi:
<http://www.python.org.pl>

Polskie forum Pythona: <http://forum.python.org.pl/>

Polska grupa dyskusyjna poświęcona Pythonowi: <http://groups.google.pl/group/pl.comp.lang.python>