

TRZY KILO GADZINY



www.sxc.hu

Na początku grudnia 2008 roku wydany został Python 3.0, zwany również – ze względu na zakres zmian i unowocześnień – Pythonem 3000 (w skrócie: Py3k). Sami autorzy tego wszechstronnego języka programowania podkreślają, że dawka nowości jest bezprecedensowa.

JAN KALISZEWSKI

By uporządkować Pythona, a zarazem otworzyć nowe możliwości jego rozwoju, świadomie zerwano wsteczną zgodność w wielu elementach języka – jeśli chodzi zarówno o składnię, jak i o obszerną bibliotekę standardową. Tak głębokie zmiany wywołały szereg dyskusji, przy okazji których jak bumerang wraca kwestia: „kiedy i jak przenosić projekty do Pythona 3.0?”. Wydaje się, że rozsądna odpowiedź na pytanie „kiedy?” brzmi: „gdy przemiegrowane zostaną potrzebne nam biblioteki i frameworki”. Wyczerpującą odpowiedź na pytanie „jak?” (włącznie z opisem narzędzia 2to3, które automatyzuje szereg czynności) znajdziemy zaś w oficjalnej dokumentacji [1].

Rozwiemy – ujawniające się czasem w dyskusjach – obawy o los projektów bazujących na linii 2.x języka: jej rozwój będzie kontynuowany przez co najmniej kilka lat. Prawie równoległe z Py3k wypuszczono Pythona 2.6 (wprowadzającego te cechy Py3k, które nie powodują istotnych niezgodności), a już trwają prace nad wersją 2.7. Warto też przypomnieć, że obecnie wciąż wypuszczane są poprawki do wersji 2.4 (z 2004 roku) oraz 2.5 (z 2006 roku).

Przyrządzanie

By posmakować tego młodego gada, musimy go (cóż za niespodzianka!) zainstalować. Jak na razie w większości dystrybucji nie znaj-

dziemy gotowych pakietów dla wersji 3.0, pozostaje więc instalacja ręczna. Do dzieła!

Z oficjalnego serwera ftp [2] pobieramy plik *Python-3.0.tgz*, rozpakowujemy go, wpisując w terminalu:

```
tar xzf Python-3.0.tgz
```

...i patrzymy – co my tu mamy?

```
cd Python3.0
./configure -h
```

Po zapoznaniu się z możliwymi parametrami instalacji uruchamiamy:

```
./configure
```

(zdając się na ustawienia domyślne lub dostosowując opcje wedle życzenia), następnie:

```
make
make test
```

...a wreszcie:

```
sudo make altinstall
```

By uczynić Py3k główną wersją Pythona w naszym systemie (ostrożnie – możemy pospuścić sobie system przez „wykolejenie” programów zależnych od Pythona 2.x!), zamiast *altinstall* wpisujemy *install*.

Jeżeli wszystko poszło dobrze, gad jest już na naszym talerzu. Wywołujemy więc komendę:

```
python3.0
```

Degustacja

Aperitif. Zaczniemy od drobiazgów. Ile wynosi 2^{80} ?

```
>>> 2 ** 80
1208925819614629174706176
```

Ładny drobiazg! Dla bardzo dużych liczb całkowitych Python 2.x miał osobny typ: *long* – taką liczbę mogliśmy poznać po doklejonej na końcu literze *L*. Ale tu niczego podobnego nie widać. Py3k oferuje bowiem tylko jeden typ liczb całkowitych: zwykły *int* – tyle że bez ograniczenia zakresu! Prawda, że prościej?

```
>>> 35/10
3.5
```

Nareszcie wynik dzielenia liczb całkowitych to liczba rzeczywista (bez konieczności wpisywania dziwactw w rodzaju *1./2* czy *1/2.0*).

A co z dzieleniem z resztą?

```
>>> 35//10
3
```

Skoro już mowa o liczbach... Dodano literały dwójkowe, zmieniono też notację literałów ósemkowych (tak, by były podobne do dwójkowych i szesnastkowych):

```
>>> 0b1010
10
>>> bin(10)
'0b1010'
>>> 0o12 # dawniej 012
10
```

Coś ostrzejszego

```
>>> u'Człowiek o 10 nogach'
File "<stdin>", line 1
u'Człowiek o 10 nogach'
^
SyntaxError: invalid syntax
```

Cóż to! Nie ma typu *unicode*? Nie ma, bo nie jest już potrzebny – teraz wszystkie napisy (*str*) są unikodowe.

```
>>> 'Człowiek o 10 nogach'
'Człowiek o 10 nogach'
```

Ponadto dodano dwa zupełnie nowe typy sekwencyjne do operowania danymi binarnymi: *bytes* (niezmiennie i haszowalne, literał ma postać *b"ascii"*) i *bytearray* (niehaszowalne, za to udostępniające metody analogiczne do operacji na listach).

```
>>> 'Człowiek'.encode()
b'Cz\x05\x82owiek'
>>> b'Cz\x05\x82owiek'[2:].decode()
'łowiek'
>>> a = bytearray(b'papuga')
>>> a.extend(b'Martw') # bytes
>>> a.append(97) # int
>>> a
bytearray(b'papugaMartwa')
>>> del a[3:11]
>>> a.decode('utf-8')
'papa'
```

Jak widać, możemy konwertować dane tekstowe na binarne i z powrotem, ale musimy to robić jawnie (opcjonalnie podając kodowanie).

```
>>> print a
File "<stdin>", line 1
print a
^
SyntaxError: invalid syntax
```

Zła składnia? Owszem, bo – uwaga – *print* nie jest już słowem kluczowym, lecz zwykłą funkcją!

```
>>> print(a)
bytearray(b'papa')
>>> print('2 i 2 daje', 11*2)
2 i 2 daje 22
>>> print("print coś,", end=" ")
print coś, >>>
>>> print(192, 168, 0, 1, sep=".")
192.168.0.1
>>> def print(*ni, **niu): pass
...
>>> print('2 i 2 daje', 11*2)
>>> print(a)
>>>
>>> del print # przywracamy
```

...co, jak widać, ma prawie same zalety i tylko jedną wadę: musimy przewyciężyć bardzo silny (oj, przekonamy się, jak bardzo) nawyk wpisywania *print* bez nawiasów.

Stare kontenery – nowe smaki

Typ zbioru (*set*) doczekał się własnego literału:

```
>>> set([1, 2, 3, 4])
{1, 2, 3, 4}
```

Podobieństwo do literału słownika (*dict*) nie jest przypadkowe – w końcu o zbiorze możemy myśleć jako o słowniku z samymi kluczami (na słownikach zresztą opierała się pierwotna implementacja zbiorów w nieistniejącym już module *Sets*).

```
>>> set()
set()
```

Tu drobna niespodzianka: pusty zbiór nie posiada analogicznej formy „literałowej”. Oznaczenie `{}` jest już zarezerwowane dla słowników (i tego autorzy języka postanowili nie zmieniać).

Do znanych z poprzednich wersji Pythona *generator expressions* i *list comprehensions* (zachowanie tych ostatnich zostało zresztą subtelnie zmodyfikowane – w celu ujednolicenia ze sposobem działania tych pierwszych) dołączyły analogiczne konstrukcje tworzące zbiory i słowniki:

```
>>> {2**i for i in range(6)}
{32, 1, 2, 4, 8, 16}
>>> {i: 2**i for i in range(6)}
{0:1, 1:2, 2:4, 3:8, 4:16, 5:32}
```

Skoro już jesteśmy przy *range()*, musimy odnotować, że (w ramach czyszczenia z nadmiarowych elementów zestawu funkcji wbudowanych) dawne *range()* wyrzucono, zaś *xrange()* przemianowano na *range()*:

```
>>> range(10) # py2: xrange
range(0, 10)
>>> list(range(10)) # py2: range
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Jest to zresztą przejaw ogólniejszego trendu: ograniczania użycia list na rzecz iteratorów lub podobnych do nich obiektów. Dobrym tego przykładem są *widoki* (*views*) zwracane, zamiast list, przez słownikowe metody: *dict.keys()*, *dict.items()* oraz *dict.values()* (przy czym *dict.iterkeys()*, *dict.iteritems()* i *dict.itervalues()* zostały usunięte jako już niepotrzebne).

```
>>> d1 = {'j':1, 'd':2, 't':3}
>>> d2 = {'b':2, 'c':4, 'd':5}
>>> d1.keys()
<dict_keys object at 0xb7ae2f40>
>>> d2.values()
<dict_values object at 0xb7ae2ea0>
```

Widoki są iterowalne:

```
>>> for i in d1.items(): print(i)
...
('j', 1)
('d', 2)
('t', 3)
```

Powinniśmy przy tym pamiętać, że zachowują się dynamicznie: operacja zmieniająca słownik zmienia również jego widoki.

```
>>> 1 in d1.values()
True
>>> del d1['j']
>>> 1 in d1.values()
False
>>> len(d1.values())
2
```

Jak widzimy, działa również sprawdzanie obecności i długości.

Dla widoków kluczy (*keys*) bądź elementów (*items*) możemy też wykonywać podstawowe operacje właściwe zbiorom:

```
>>> d1.keys() & d2.keys()
{'d'}
>>> d1.items() | d2.items()
{('b', 2), ('d', 5), ('t', 3), ('d', 2), ('c', 4)}
```

Listing 1: Poprawiacz.py

```

#!/usr/bin/env python3.0
# Domyślne kodowanie źródeł to utf-8 (więc nie trzeba go deklarować)

import optparse, os, re, sys
from collections import namedtuple # Krotki z nazwanymi polami
                                   # dostępnymi jako atrybuty

Zamiana = namedtuple("Zamiana", "rexp, znak")
zamiany = tuple(Zamiana(rexp=re.compile(r), znak=z) for r, z in (
    ("\.{3}", "\u2026"), ("\B-{1,3}\B", "\u2013"),
    ("(?=[\(\[\w])'", "\u201e"), ("", ",", "\u201e"),
    ("'", "\u201d"), ("'", "\u201d"), ("'", "\u2019")))

# Adnotacje dot. argumentów i wyniku funkcji (np. do kontroli typów):
def zamieniaj(napis: str, zamiany: "co zamienić na co") -> str:
    for zamiana in zamiany:
        napis = re.sub(zamiana.rexp, zamiana.znak, napis)
    # ^ Można by też jak dla zwykłych krotek: for (r, z) in zamiany...
    return napis

o = optparse.OptionParser(usage="%prog [-h|--help] | [-f|--force] +
    " INPUT_FILE1 [INPUT_FILE2 ...] OUTPUT_FILE")
o.add_option("-f", "--force", action="store_true", help="never prompt")
# Identyfikatory mogą zawierać znaki spoza zbioru ASCII:
опції, файлу = o.parse_args()

if not файлу: o.error("No files given!")
*ścieżki_wej, ścieżka_wyj = файлу # Rozszerzone rozpakowywanie
if not ścieżki_wej: o.error("No input file(s) given!")
if not опції.force and os.access(ścieżka_wyj, os.F_OK):
    # Metoda format() - nowy sposób (a wręcz mini-język) formatowania:
    pytanie = "File {0} exists! Overwrite? [y/n] ".format(ścieżka_wyj)
    odpowiedź = input(pytanie) # input() zamiast raw_input()
    if not odpowiedź.startswith(("y", "Y")):
        sys.exit()

try:
    # with: prościej niż try...finally... (nie tylko dla plików...)
    with open(ścieżka_wyj, "w") as plik_wyj:
        for ścieżka in ścieżki_wej:
            # Otwarcie pliku: tylko open(), nie file()
            with open(ścieżka) as plik:
                plik_wyj.writelines(zamieniaj(wiersz, zamiany)
                                    for wiersz in plik)
except IOError as błąd: # KlasaBł as bł - a nie mylące: KlasaBł, bł
    # Znów metoda format(): nazwane argumenty, odwołania do atrybutów:
    info = "A problem with file {błąd.filename}!".format(błąd=błąd)
    # Wypisywanie na wskazane wyjście bez składniowych ekstrawagancji:
    print(info, file=sys.stderr)
    sys.exit(1)

```

```

>>> d1.keys() - {'d'}
{'t'}
>>> {'d'} ^ d2.keys()
{'c', 'b'}

```

Większe danie

Jeżeli nabraliśmy już apetytu na odrobinę większy kawałek kodu, zobaczymy, co zaserwowała nam kuchnia na Listingu 1. Ten nieskomplikowany programik czyta jeden lub więcej plików tekstowych, poprawia wielokropki, myślniki, cudzysłowy i apostrofy zgodnie z regułami polskiej typografii, a rezultat zapisuje do jednego pliku wynikowego.

Warto uważnie przeczytać kod – komentarze wskazują kolejne, nieomówione wyżej elementy języka.

By sprawdzić program w działaniu, wpisujemy w terminalu:

```
python3.0 Poprawiacz.py
```

Całość (dzięki zastosowaniu bardzo przyjemnego, choć wcale nie nowego modułu *optparse*) obsługuje się jak typowe narzędzie uniksowe; na początek przyda nam się opcja *--help*.

Deser?

W ramach tak krótkiego obiadu... *pardon*: artykułu – nie sposób zaprezentować wszystkich nowości w Py3k, jakie warto spróbować. Choćby te nowe smaczki w systemie typów i klas... Ślinka cieknie na myśl o klasach abstrakcyjnych (*ABCs*), dekoratorach klas czy (ledwo tylko liźniętych w załączonym programiku) adnotacjach (*annotations*) do parametrów oraz wartości wynikowych funkcji.

Ale dość już robienia smaku! Kto ciekaw, w te pędy udaje się na stronę dokumentacji Pythona. Ten gad naprawdę nie gryzie. ■

INFO

[1] Dokumentacja Pythona 3.0:
<http://docs.python.org/3.0/>

[2] Oficjalne zasoby FTP: <http://www.python.org/ftp/python/3.0/>

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

[4] Poprawiacz.py: <http://linuxmagazine.pl/index.php/issues/61/poprawiacz.py>

AUTOR

Jan Kaliszewski jest kompozytorem, działaczem pozarządowym, a także administratorem sieci. Pełni funkcję prezesa stowarzyszenia Jeunesses Musicales Polska. Pythona używa od kilku lat. Kontakt: jan.kaliszewski@jm.org.pl