Rafał Korzeniewski

Wstęp do języka Python. Część 6

Krotki i listy to kolejne narzędzia, które pozwolą nam na poszerzenie naszych programistycznych możliwości. Jak się przekonamy, te wbudowane w Pythona struktury danych kryją swoje tajemnice.

KROTKA/TUPLA

W poprzednim numerze (PJR 3/2020) poznaliśmy napisy. Wiele z informacji, które tam się pojawiło, wykorzystamy także i dzisiaj, tak więc warto je sobie odświeżyć.

Wiemy już, że na napis możemy spojrzeć jak na kolekcję znaków, które występują w pewnej kolejności. Jest to struktura, w której każdy element ma swoją pozycję. A czy istnieje jakiś sposób na to, by w podobny sposób przechowywać informacje na przykład o liczbach albo o szeregu różnych napisów? Oczywiście są na to sposoby. Jako pierwszy z nich przedstawimy krotki – często też nazywane ich angielską nazwą: *tuple*. Taka tupla to zbiór elementów rozdzielonych przecinkami, dodatkowo umieszcza się je zazwyczaj w nawiasach okrągłych. Weźmy na przykład wszystkie liczby pierwsze, mniejsze od 10 (o liczbach pierwszych przeczytaj w ciekawostce).

Listing 1. Przykładowa tupla

```
In [1]: (2, 3, 5, 7)
Out[1]: (2, 3, 5, 7)

In [2]: 2, 3, 5, 7
Out[2]: (2, 3, 5, 7)

In [3]: liczby = 2, 3, 5, 7

In [4]: liczby
Out[4]: (2, 3, 5, 7)

In [5]: liczby = (2, 3, 5, 7)

In [6]: liczby
Out[6]: (2, 3, 5, 7)
```

Możemy sprawdzić, jakiego typu jest zmienna `liczby`. By to zrobić, wykorzystamy znaną nam już funkcję `type`:

Listing 2. Sprawdzenie typu

```
In [7]: type(liczby)
Out[7]: tuple
```

Jak widzimy, została zwrócona nazwa `tuple`. Jest to też funkcja[1], która potrafi zamienić różne rzeczy na tuplę – na przykład napis – co zobaczymy w Listingu 3.

Listing 3. Działanie funkcji „tuple" i jej dokumentacja

```
In [8]: tuple("abc")
Out[8]: ('a', 'b', 'c')

In [9]: tuple?
Init signature: tuple(iterable=(), /)
Docstring:
Built-in immutable sequence.

If no argument is given, the constructor returns
an empty tuple.
If iterable is specified the tuple is initialized
from iterable's items.

If the argument is a tuple, the return value is
the same object.
Type:           type
Subclasses:     int_info, float_info,
UnraisableHookArgs, hash_info,
version_info, flags, thread_info, asyncgen_hooks,
ExceptHookArgs, stat_result, ...
```

W Listingu 3 pokazano też pomoc dla tej funkcji. Jak widać, przyjmuje ona jako argument coś nazywanego „iterable". Argument ten ma też wartość domyślną `()`.

1. Jest to pewne uproszczenie. W tym i podobnych przypadkach (`list`, `dict`, `set`, `str`) z technicznego punktu widzenia jest to klasa, a nie funkcja. O klasach i funkcjach będzie jeszcze mowa w kolejnych częściach naszej serii

Funkcję można więc wywołać bez podawania żadnych argumentów – zwracana jest wtedy pusta krotka. Jeśli podamy argument i będzie on iterowalny, to zostanie zwrócona krotka zbudowana z elementów tego obiektu. Słowo „iteracja" pochodzi od łacińskiego słowa *iteratio*, które oznacza powtarzanie. Określamy tym słowem powtarzanie tej samej operacji w pętli z góry określoną liczbę razy lub aż do spełnienia określonego warunku. Ta wiedza przyda nam się jeszcze, kiedy będziemy omawiać tak zwane pętle. Teraz zauważmy jedynie, że napisy i same tuple też są takimi obiektami. Tupla w samej dokumentacji nazywana jest też *sekwencją*. Słowo to oznacza uporządkowany ciąg. Może to być ciąg liczb, znaków, nazw. Zauważmy też, że użyte jest tu określenie *immutable sequence*. To *immutable* oznacza, że coś jest niezmienialne, mówimy też często po polsku, że jest to *niemutowalne*. Raz utworzonej tupli nie możemy zmienić (mutować). Oznacza to, że nie możemy do niej dodać ani usunąć żadnych nowych elementów, zmienić tych istniejących. Możemy oczywiście na podstawie istniejącej tupli utworzyć nową, ale to będzie z punktu widzenia Pythona inny obiekt.

Listing 4. Tuple możemy łączyć, tworząc nowe. Ale nie możemy zmienić istniejących

```
In [1]: liczby = (2, 3, 5, 7)

In [2]: id(liczby)
Out[2]: 4594589760

In [3]: liczby = liczby + (11, 13)

In [4]: liczby
Out[4]: (2, 3, 5, 7, 11, 13)

In [5]: id(liczby)
Out[5]: 4596639200
```

Niemutowalne są w Pythonie także napisy. Czy są tu jeszcze jakieś podobieństwa? Są. Na przykład tupla oferuje nam metody znane z napisów: `index`, `count`.

Listing 5. Metody dostępne dla krotek

```
In [9]: liczby.count(2)
Out[9]: 1

In [10]: liczby.count(1)
Out[10]: 0

In [11]: liczby.index(11)
Out[11]: 4
```

Czy to wszystkie metody dostępne dla krotek? Możemy przyjąć, że tak. Możemy to sprawdzić za pomocą funkcji `dir`, jak w Listingu 6.

Listing 6. Użycie funkcji „dir" do sprawdzenia, jakie atrybuty i metody dostępne są dla tupli

```
In [12]: dir(liczby)
Out[12]:
['__add__',

  ...

 '__subclasshook__',
 'count',
 'index']
```

Widzimy tu szereg różnych wpisów ujętych w podwójne znaki podkreślenia. To są specjalne atrybuty i metody, których przeznaczenie poznamy w przyszłości. Do takich zwykłych zastosowań służą te, których nazwy nie mają tych otaczających podkreśleń. Pamiętasz, jak sprawdzić pomoc dla tych metod? Możesz użyć funkcji `help`, a jeśli używasz IPythona, to po nazwie metody możesz dodać znak zapytania. Przykłady pokazano w Listingu 7.

Listing 7. Korzystanie z pomocy. Sposób ze znakiem zapytania zadziała, tylko jeśli używasz IPythona

```
In [13]: help(liczby.count)

Help on built-in function count:

(value, /) method of builtins.tuple instance
Return number of occurrences of value.
(END)

(autor: Tu trzeba nacisnąć q by wyjść z pomocy)
In [14]: liczby.index?
Signature: liczby.index(value, start=0,
stop=9223372036854775807, /)
Docstring:
Return first index of value.

Raises ValueError if the value is not present.
Type:      builtin_function_or_method
```

Jakie jeszcze podobieństwa z napisami mają krotki? Całkiem sporo. Możemy na przykład sprawdzić długość krotki za pomocą funkcji `len`. Spójrzmy w Listing 8.

Listing 8. Działanie funkcji „len" dla tupli jest analogiczne jak dla napisu

```
In [16]: tekst = "W Brańsku żyje się po pańsku"

In [17]: len(tekst)
Out[17]: 28

In [18]: znaki = tuple(tekst)

In [19]: znaki
Out[19]:
('W', ' ', 'B', 'r', 'a', 'ń', 's', 'k', 'u',
 ' ', 'ż', 'y', 'j', 'e', ' ', 's', 'i', 'ę',
' ', 'p', 'o', ' ', 'p', 'a', 'ń', 's', 'k', 'u')

In [20]: len(znaki)
Out[20]: 28
```

Każdy element tupli ma swoją pozycję – indeks. Indeks ten liczony jest od zera.

Pamiętasz wybieranie z artykułu o napisach? W tuplach działa to identycznie – może tylko za wyjątkiem tego, że zwrócony obiekt będzie w przypadku krotki nową krotką.

Listing 9. Wybieranie działa analogicznie jak w przypadku napisów

```
In [20]: len(znaki)
Out[20]: 28

In [21]: znaki[0]
Out[21]: 'W'

In [22]: znaki[27]
Out[22]: 'u'

In [23]: znaki[-1]
Out[23]: 'u'

In [24]: znaki[2:12:2]
Out[24]: ('B', 'a', 's', 'u', 'ż')
```

Zobacz, jakie to fajne. Nauczyliśmy się czegoś z artykułu o napisach i od razu to procentuje. Zdobyta wiedza przydaje się nam w pracy z nowym typem danych. A to, czego do tej pory nauczyliśmy się o tuplach, za chwilę będziemy mogli wykorzystać także przy kolejnej sekwencji – a więc przy listach.

LISTY

Listy to sekwencje bardzo podobne do tupli. Do ich tworzenia służą nawiasy kwadratowe albo funkcja `list`. To, co funkcja `tuple` robi w przypadku krotek, funkcja `list` robi dla list. Mają one także metody `count` i `index`. Zadziała też dla nich funkcja `len` czy tak zwany *slicing* (wybieranie)

Listing 10. Tworzenie list

```
In [25]: parzyste = [2, 4, 6, 8]

In [26]: pierwsze = list(liczby)

In [27]: pierwsze
Out[27]: [2, 3, 5, 7, 11, 13]

In [28]: parzyste
Out[28]: [2, 4, 6, 8]

In [30]: parzyste.index(6)
Out[30]: 2
```

Czym więc te listy różnią się od naszych tupli? Zerknijmy w dokumentację:

Listing 11. Dokumentacja typu list. Taka dokumentacja w świecie Pythona określana jest też jako docstring (czytaj: dokstring)

```
In [31]: list?
Init signature: list(iterable=(), /)
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor
creates a new empty list.
The argument must be an iterable if specified.
Type:           type
Subclasses:     _HashedSeq, StackSummary, SList,
_ImmutableLineList, FormattedText, NodeList,
 _ExplodedList, Stack, _Accumulator
```

Jak widzimy, jest to wbudowana sekwencja, ale mutowalna. Oznacza to, że lista może być zmieniana. Do takich zmian nie wystarczą dwie znane nam metody – potrzebujemy ich więcej. I tak na przykład do listy możemy dodawać (na przykład metody: `append`, `insert`) i to nadal będzie ta sama lista. Możemy z niej też usuwać elementy (na przykład: `remove`, `pop`), a nawet rozszerzać o elementy z innej listy (`extend`). Zachęcam cię gorąco do przeczytania dokumentacji tych metod. Kryją one różne dodatkowe, przydatne parametry. W Listingu 12 pokażę najprostsze zastosowanie.

Listing 12. Inne metody list

```
In [33]: parzyste.append(14)

In [34]: parzyste.insert(1, 10)

In [35]: parzyste
Out[35]: [2, 10, 4, 6, 8, 14]

In [36]: parzyste.remove(4)

In [37]: parzyste
Out[37]: [2, 10, 6, 8, 14]

In [38]: parzyste.pop()
Out[38]: 14

In [39]: parzyste
Out[39]: [2, 10, 6, 8]

In [40]: parzyste.extend([2, 12, 10])

In [41]: parzyste
Out[41]: [2, 10, 6, 8, 2, 12, 10]
```

Te nowe metody mają różne zachowania. Niektóre z nich, jak na przykład `append`, `insert`, `remove`, `extend`, po prostu modyfikują nasza listę, nie zwracają jednak żadnej wartości, a właściwie, by być precyzyjnym – zwracają wartość `None`. Metoda `pop` trochę się różni. Zauważ, że nie tylko zmienia naszą listę, ale też zwraca wartość. Domyślnie usuwa ona z listy ostatni element i go zwraca.

Jeśli więc zechcemy dodać do naszej listy element, to użyjemy metody `append` – po polsku oznacza to „dołącz". Przyjmuje ona jako argument dowolny obiekt i dodaje go na końcu listy. No właśnie – zarówno tuple, jak i listy mogą mieć w sobie elementy różnych typów. Zobaczymy to wszystko w Listingu 13:

Listing 13. Lista i „tuple" mogą zawierać w sobie jednocześnie elementy różnych typów

```
In [8]: elementy = []

In [9]: elementy.append(100)

In [10]: elementy.append("a")

In [11]: elementy.append(1.23)

In [12]: elementy.append(("to", "jest", "tupla"))

In [13]: elementy.append(print)

In [14]: elementy
Out[14]: [100, 'a', 1.23, ('to', 'jest', 'tupla'),
<function print>]
```

Dodaliśmy do listy `elementy` liczbę całkowitą, liczbę zmiennoprzecinkową, napis, tuplę, a nawet funkcję `print`. Moglibyśmy też do naszej listy dodać inną listę jako jej kolejny element. Tak samo jak dodaliśmy tuplę w `In [12]`.

Czy wiesz, w jaki sposób moglibyśmy się dostać do elementów tej tupli z poziomu naszej listy? Zobaczmy. Sama tupla jest czwartym elementem naszej listy. Ma więc indeks = `3`. Możemy ją wybrać tak jak w Listingu 14.

Listing 14. Wybieramy krotkę z listy

```
In [21]: elementy[3]
Out[21]: ('to', 'jest', 'tupla')
```

Gdybyśmy chcieli wybrać z niej słowo `jest`, to możemy to zrobić, dodając kolejne nawiasy i wskazując tam odpowiedni indeks. Tak jak w Listingu 15.

Listing 15. Wybieranie elementu z krotki zagnieżdżonej w liście

```
In [22]: elementy[3][1]
Out[22]: 'jest'
```

Istniejące elementy w liście możemy zmieniać przy użyciu wybierania i przypisania – tak jak w Listingu 16.

Listing 16. Zmienianie elementów listy za pomocą wybierania

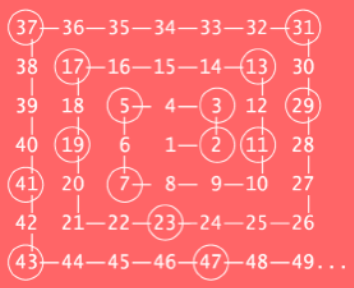
```
In [42]: parzyste[0] = 222

In [43]: parzyste
Out[43]: [222, 10, 6, 8, 2, 12, 10]
```

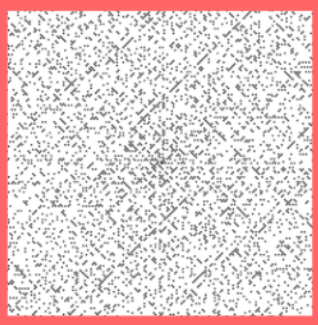
A gdybyśmy chcieli do naszej listy dodać elementy innej listy – ale tak pojedynczo, a nie jako całą listę – to moglibyśmy użyć metody `extend` – co po polsku oznacza „rozszerz". Taki przykład znajdziesz w Listingu 17:

CIEKAWOSTKA

Liczbą pierwszą nazywamy taką liczbę naturalną większą od 1, która dzieli się bez reszty tylko przez 1 i przez samą siebie. A więc 2 jest na przykład najmniejszą liczbą pierwszą. I jedyną parzystą liczbą pierwszą. Każda inna liczba parzysta dzieli się bowiem zawsze także przez 2, a więc nie tylko przez siebie i przez 1. Nie znamy wszystkich liczb pierwszych, ale ich cały zbiór ma w matematyce specjalne oznaczenie, symbol: ℙ. Nie znamy też dokładnych sposobów na to, by wyznaczać kolejne liczby pierwsze, choć wydaje się, że ich rozkład wiąże się z pewnymi prawidłowościami. Jedną z takich prawidłowości wskazał w 1963 roku wybitny polski matematyk – Stanisław Ulam.

Narysował on sobie na kartce spiralę (zobacz Ilustracja 1, górna grafika) i zaznaczył na niej liczby pierwsze.





Ilustracja 1. Spirala Ulama (źródło: https://pl.wikipedia.org/wiki/Spirala_Ulama)

Jeśli taka spirala będzie miała bardzo dużo liczb, to wystąpienia liczb pierwszych wydają się układać w pewne wzory. Widać to na dolnej grafice Ilustracji 1, gdzie czarne punkty oznaczają liczbę pierwszą. Spirala ta ma wymiary 200x200.

Znajomość tych liczb jest w informatyce bardzo istotna. Wykorzystywane są one na przykład w kryptografii.

Listing 17. Dodawanie elementów do listy

```
In [23]: elementy.append([100, 200, 300])

In [24]: elementy
Out[24]: [100, 'a', 1.23,
('to', 'jest', 'tupla'), <function print>,
[100, 200, 300]]

In [25]: elementy.extend([100, 200, 300])

In [26]: elementy
Out[26]:
[100,
 'a',
 1.23,
 ('to', 'jest', 'tupla'),
 <function print>,
 [100, 200, 300],
 100,
 200,
 300]
```

Usuwać elementy z listy możemy, jak już wspomnieliśmy, przy użyciu metod `pop`, `remove`, ale też służyć do tego może specjalne polecenie Pythona: `del` – przykłady znajdziesz w Listingu 18:

Listing 18. Usuwanie elementów z listy

```
In [27]: lista = [1, 2, 3, 1, 2, 3]

In [28]: lista.remove(2)

In [29]: lista
Out[29]: [1, 3, 1, 2, 3]

In [30]: lista.pop()
Out[30]: 3

In [31]: del lista[-1]

In [32]: lista
Out[32]: [1, 3, 1]

In [33]: lista.remove(100)
---------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-33-e279e2ae38ed> in <module>
----> 1 lista.remove(100)
```

Jak widać, do metody `remove` przekazujemy element, który chcemy usunąć. Metoda ta usuwa pierwsze wystąpienie tego elementu. Po takich zmianach, to znaczy jeśli usuniemy jakiś element z początku lub środka listy, indeksy elementów zmieniają się. Po prostu indeksy zawsze biegną od zera i nie ma tam żadnych przeskoków. Zerknijmy na znaną nam już tabelkę z indeksami:

P	r	o	g	r	a	m	i	s	t	a
0	1	2	3	4	5	6	7	8	9	10
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Jeśli usuniemy pierwsze „r", to taka tabelka będzie wyglądać następująco:

P	o	g	r	a	m	i	s	t	a
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Jak widać, literka „o" poprzednio miała indeks = 2, a teraz jest to indeks = 1.

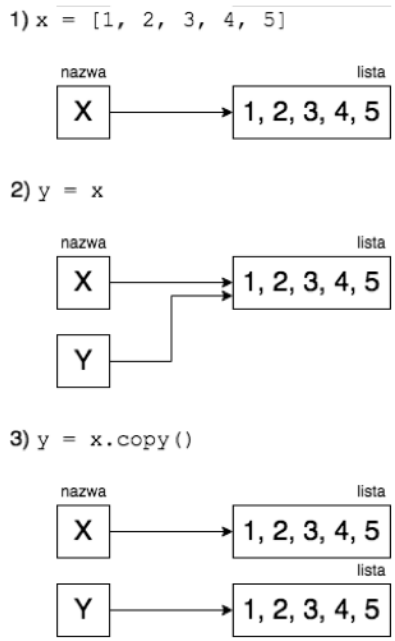
Warto wspomnieć jeszcze o tej mutowalności obiektów. Mogą się z nią bowiem wiązać różne dziwne zachowania. Spójrzmy w Listing 19.

Listing 19. Referencja i mutowalność mogą powodować dziwne zachowania

```
In [1]: x = [1, 2, 3]

In [2]: y = x

In [3]: x.append(4)

In [4]: y
Out[4]: [1, 2, 3, 4]

In [5]: x
Out[5]: [1, 2, 3, 4]

In [6]: y = x.copy()

In [7]: x.append(5)

In [8]: x
Out[8]: [1, 2, 3, 4, 5]

In [9]: y
Out[9]: [1, 2, 3, 4]
```

Wspominaliśmy kiedyś, że zmienna w Pythonie nazywana jest też czasem referencją (w pythonowym świecie czasem też mówimy o zmiennej: nazwa). Wiele referencji (nazw) może wskazywać na ten sam obiekt. Ale nie zawsze o to nam chodzi. Tak mogłoby być i w tym przypadku. Jeśli zrobimy po prostu `y = x`, to y zaczyna wskazywać na ten sam obiekt co x. Stąd każda modyfikacja tego obiektu – w tym przypadku listy – będzie widoczna przez obie referencje. Nam może chodzić o to, by w y były po prostu te same elementy co w x, ale by zmiany x nie wpływały na y. Do tego właśnie służy metoda copy.

Aby sobie to zwizualizować, zerknijmy na Ilustrację 2.



Ilustracja 2. Referencja do obiektu

W 1) po prostu tworzymy nową listę i przypisujemy ją do nazwy x. W 2) mówimy, że y ma wskazywać na tę samą listę co x. Widać więc, że wszelkie modyfikacje, czy to przez referencję x, czy y, wpłyną na ten sam obiekt. W 3) robimy kopię tego, na co wskazuje x, czyli tworzymy nowy obiekt i mówimy, że będzie na ten nowy obiekt wskazywać y.

Metoda copy to tak zwana płytka kopia. Jeśli w naszych listach będą zagnieżdżone referencje do innych obiektów, to wywołanie tej metody nie utworzy kopii tamtych obiektów – referencje pozostaną te same. Wtedy warto skorzystać z funkcji deepcopy dostępnej w module copy.

WARTO WIEDZIEĆ

W informatyce, tak i w życiu, możemy w różny sposób organizować różne zadania.

Powiedzmy, że mamy do umycia talerze. Te talerze najpierw trafiają do zlewu czy jakiejś miski. Pierwszy talerz ląduje tam na samym dole, kolejny kładziemy na nim i tak dalej. Kiedy zaczynamy zmywanie, to pierwszym zmytym talerzem będzie ten, który trafił do zlewu jako ostatni. Nie chcemy przecież wyciągać za każdym razem tego talerza z samego spodu. Byłoby to niepraktyczne. Takie podejście nazywane jest LIFO – *Last In First Out* (ostatni wszedł, pierwszy wyszedł). A taką strukturę, która korzysta z tego podejścia, nazywamy *stosem* (ang. *stack*) – zupełnie jak stos talerzy. W Pythonie bardzo łatwo uzyskamy taki stos przy użyciu listy i metod append i pop. Dodajemy do listy jakąś wartość za pomocą append, a metodą pop zrzucamy ostatnio dodaną wartość.

Inna sytuacja to kolejka. Kiedy klienci przychodzą do sklepu czy do lekarza, to pierwszy, który przyszedł, będzie pierwszym, który zostanie obsłużony. Takie podejście nazywamy FIFO – *First In First Out* (pierwszy wszedł, pierwszy wyszedł). Taką strukturę nazywamy właśnie *kolejką* (ang. *queue*). W Pythonie to podejście także da się zrealizować za pomocą list i wspomnianych metod. Może spróbujesz samodzielnie wykonać takie ćwiczenie?

Kiedyś zrobimy taki przykład, a póki co po prostu zerknij w dokumentację: https://docs.python.org/3/library/copy.html.

Warto też wiedzieć, że jest jeszcze jeden sposób na utworzenie kopii. Jeśli x jest listą, to kopię tej listy możemy wykonać następująco:

```
y = x[:]
```

Czyli do nowej nazwy y przypisujemy listę składającą się z wszystkich elementów x.

Rafał Korzeniewski

Z wykształcenia muzyk-puzonista i fizyk. Z zamiłowania i zawodu Pythonista. Trener Pythona, współorganizator PyWaw (http://pywaw.org) – warszawskiego meetupu poświęconego Pythonowi. W wolnych chwilach uczy się gry na nowych instrumentach, udziela się społecznie i dużo czyta.

KORZENIEWSKI@GMAIL.COM