Laboratorium programistyczne — Python: Napisy i wyrażenia regularne

Projekt „Matematyka dla ciekawych świata”
Robert Paciorek

2019-03-21

1 Napisy

Do tej pory używaliśmy zmiennych do przechowywania liczb i operowania na nich. Zmienne mogą również jako wartości przyjmować litery, słowa, a nawet całe zdania:

```
x = 'A'
a, b, c = 'Ala', "ma", " kota i psa"
d = """ ... a co ma ...
 "kotek"?"""
print(x, a[2])
print(c[1], c[-1], c[-3])
print(a + b)
print(3 * a)
print(a + " " + b + c + d)
```

```
A a
o a p
Alama
AlaAlaAla
Ala ma kota i psa ... a co ma ...
 "kotek"?
```

Zwróć uwagę na następujące rzeczy:

- Napisy muszą być otoczone pojedynczymi apostrofami lub podwójnym cudzysłowami (nie ma znaczenia, którą wersję wybierzemy), w przypadku napisów wieloliniowych używamy trzykrotnie apostrofu lub cudzysłowowa na początku i końcu napisu.
  Nie przypisane do żadnej zmiennej napisy wieloliniowe mogą być stosowane jako komentarze wieloliniowe.
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery napisu (***numeracja rozpoczyna się od 0***).
- Ujemny indeks oznacza odliczanie liter od końca napisu: ostatnia litera napisu `c` to `c[-1]`, przedostatnia to `c[-2]`, itd.
- Przy użyciu znaku dodawania możemy sklejać (*konkatenować*) napisy.
- Przy użyciu znaku gwiazdki możemy mnożyć napisy (czyli sklejać same ze sobą).

Innymi przydatnymi operacjami na napisach jest sprawdzanie długości napisu poleceniem **len**() oraz wycinanie podnapisu przy użyciu nawiasów kwadratowych z dwukropkiem w sposób analogiczny do wyboru podlisty:

```
tekst = '123456789'
dlugosc = len(tekst)
print(dlugosc, tekst[2:5], tekst[3:], tekst[:3])
print(tekst[::-1], tekst[1::2], tekst[::-3])
print(tekst[::-1][::3], tekst[::3][::-1])
```

```
9 345 456789 123
987654321 2468 963
963 741
```

W powyższym przykładzie:

- komenda `tekst[2:5]` zwraca podnapis od znaku nr 2 (**włącznie**) do znaku nr 5 (**wyłącznie**),
- komenda `tekst[3:]` zwraca podnapis od znaku nr 3 (**włącznie**) do końca,
- komenda `tekst[:3]` zwraca podnapis od początku do znaku nr 3 (**wyłącznie**).
- komenda `tekst[::-1]` zwraca napis od tyłu,
- komenda `tekst[1::2]` zwraca co drugi znak od znaku nr 1,
- komenda `tekst[::-3]` zwraca co 3 znak z napisu od tyłu (warto zauważyć że nie zawsze jest to równoważne wypisaniu napisu złożonego z co 3 znaku od tyłu).

1.1 Napis jako lista

Zapewnet (przy wybieraniu znaków i podnapisów) zauważyłeś już pewne podobieństwo napisów do list. W Pythonie napisy mogą być traktowane jako lista, a dokładniej listą liter. Oznacza to, że po napisie można przejść przy użyciu pętli **for**, tak samo jak przechodziliśmy po liście liczb:

```
for l in 'Abc':
    print('litera', end = ' ')
    print(l)
```

```
litera A
litera b
litera c
```

1.1.1 Modyfikowalność napisów

Python pozwala odwoływać się do poszczególnych znaków w napisie jak do elementów listy, jednak nie pozwala na ich modyfikowanie:

```
s = "abcdefgh"
s[2] = "X"
print(s)
```

```
Traceback (most recent call last):
  File "python", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Zwróć uwagę na komunikat błędu, który został wyświetlony, podaje on informacji o tym co wywołało błąd (opis błędu) i w której linii programu on wystąpił. **Czytanie ze zrozumieniem komunikatów o błędach ułatwia naprawianie niedziałającego programu.**

Jeżeli zachodzi potrzeba modyfikowania napisu konkretnych znaków w napisie możemy użyć poznanej wcześniej metody uzyskiwania podnapisów:

```
s = "abcdefgh"
s = s[:2] + "X" + s[3:5] + s[6:]
print(s)
```

```
abXdegh
```

Powyższy przykład w miejsce znaku nr 2 wstawia napis ”X” oraz usuwa znak nr 5 z napisu. Przy konieczności modyfikacji znak po znaku możemy użyć iteracji po napisie i budować nowy napis znak po znaku:

```
s, ns = "abcdefgh", ""
for z in s:
    if z in "cf":
        ns = ns + "X"
    else:
        ns = ns + z
print(ns)
```

```
abXdeXgh
```

Jako narzędzie do modyfikowania napisów mogą służyć także poznane wcześniej listy. W tym celu można skorzystać np. z listy złożonej z liter oryginalnego napisu:

```
s = "abcdefgh"
l = list(s)
l[2] = "X"
del(l[5])
s = "".join(l)
print(s)
```

```
abXdegh
```

Zadanie 1.1.1

Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy wspak. Np. dla listy `['Ala', 'ma', 'kota']` funkcja powinna wypisać: `alA am atok`

Zadanie 1.1.2

Napisz funkcję `wyiksuj(napis)`, która wypisze dany `napis`, zastępując każdą małą literę polskiego alfabetu małą literą x i każdą wielką literę polskiego alfabetu wielką literą X, natomiast resztę znaków pozostawi bez zmian. Np. dla napisu `'Python 3.6.1 (default, Dec 2015, 13:05:11)'` program powinien wypisać: `Xxxxxx 3.6.1 (xxxxxxx, Xxx 2015, 13:05:11)`

Zadanie 1.1.3

Napisz funkcję, która dla danej listy słów wypisze w kolejnych wierszach ich skróty w postaci `<pierwsza litera>-<ostatnia litera> (<dlugosc slowa>)`.
Np. dla listy `['Interdyscyplinarne', 'Centrum', 'Modelowania']` powinna wypisać:

```
I-e (18)
C-m (7)
M-a (11)
```

Wskazówka: wynik funkcji **len**`()` mierzącej długość napisu jest liczbą. Do rozwiązania tego zadania może Ci się przydać konwersja tej liczby na napis (aby dało się ją skleić z innymi napisami), z użyciem funkcji **str**`()`

Zadanie 1.1.4

Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy powtarzając każdą małą literę dwukrotnie. Np. dla `['Ala', 'ma', 'kota', 'i PSA']` funkcja powinna wypisać:

```
Allaa
mmaa
kkoottaa
ii PSA
```

1.2 Kodowania znaków

Python używa Unicode dla obsługi napisów, jednak przed przekazaniem napisu do świata zewnętrznego konieczne może być zastosowanie konwersji do określonej postaci bytowej (zastosowanie odpowiedniego kodowania). Służy do tego metoda `encode()` np.:

```
a = "aąbcć ... ⇄"
inUTF7 = a.encode('utf7')
inUTF8 =  a.encode() # lub a.encode('utf8')
print("'" + a + "' w UTF7 to: " + str(inUTF7) + ", w UTF8: " + str(inUTF8))
```

Zmienne typu 'bytes' oprócz przekazania na zewnątrz (np. zapisu do pliku lub wysłania przez sieć) mogą zostać także m.in. zdekodowane do napisu z użyciem metody `decode()` lub poddane dalszej konwersji np. kodowaniu base64:

```
print("zdekodowany UTF7: " + inUTF7.decode('utf7'))

import codecs
b64 = codecs.encode(inUTF8, 'base64')
print("napis w UTF8 po zakodowaniu base64 to: " + str(b64))
```

W powyższym przykładzie należy zwrócić uwagę na instrukcję **import**, która służy do załączania bibliotek pythonowych do naszego programu. W tym wypadku załączamy fragment standardowej biblioteki Pythona o nazwie `codecs`.

Base64 jest jednym z kodowań pozwalających na zapis danych binarnych w postaci ograniczonego zbioru znaków drukowalnych, co pozwala m.in. na osadzanie danych binarnych (np. obrazki) w plikach tekstowych (np. dokumenty html, pliki źródłowe programów).

1.2.1 Konwersja pomiędzy znakiem a jego numerem

Możliwe jest także konwertowanie pomiędzy liczbowym numerem znaku Unicode, a napisem go reprezentującym i w drugą stronę — służą do tego odpowiednio funkcje **chr()** i **ord()**. W ramach napisów można też użyć `\uNNNN`, gdzie `NNNN` jest (czterocyfrowym) numerem znaku lub po prostu umieścić dany znak w pliku kodowanym UTF8[1].

```
print(chr(0x221e) + " == \u221e == ∞")
print(hex(ord("∞")), hex(ord("\u221e")), hex(ord(chr(0x221e))) )
```

Zadanie 1.2.1

Napisz program dekodujący napis kodowany w UTF8 zakodowany przy pomocy base64 mający postać: `b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'`.
Wskazówka: dane wejściowe funkcji `decode()` muszą być typu "bytes", można to uzyskać poprzedzając napis prefiksem `b`, tak jak powyżej.

1.3 Wyrażenia regularne

W przetwarzaniu napisów bardzo często stosowane są wyrażenia regularne służące do dopasowywania napisów do wzorca który opisują, wyszukiwaniu/zastępowaniu tego wzorca. Do typowej, podstawowej składni wyrażeń regularnych zalicza się m.in. następujące operatory:

```
.      - dowolny znak
[a-z]  - znak z zakresu
[^a-z] - znak z poza zakresu (aby mieć zakres z ^ należy dać go nie na początku)
^      - początek napisu/linii
$      - koniec napisu/linii
```

[1] Użyty w przykładzie symbol nieskończoności można uzyskać na standardowej polskiej klawiaturze pod Linuxem przy pomocy kombinacji AltGr + Shift + M

```
*       - dowolna ilość powtórzeń
?       - 0 lub jedno powtórzenie
+       - jedno lub więcej powtórzeń
{n,m}   - od n do m powtórzeń

()      - pod-wyrażenie (może być używane dla operatorów powtórzeń,
          a także dla referencji wstecznych)
```

Python umożliwia korzystanie z wyrażeń regularnych za pomocą modułu `re`:

```python
import re

y = "aa bb cc bb ff bb ee"
x = "aa bb cc dd ff gg ee"

if re.match(".*[dz]", y):
  print("y zawiera d lub z")

if re.match(".*[dz]", x):
  print("x zawiera d lub z")

if re.match(".* ([a-z]{2}) .* \\1", y):
  print("y zawiera dwa razy to samo")

if re.match(".* ([a-z]{2}) .* \\1", x):
  print("x zawiera dwa razy to samo")

# zastępowanie
print (re.sub('[bc]+', "XX", y, 2))
print (re.sub('[bc]+', "XX", y))

# zachłanność
print (re.sub('bb (.*) bb', "X \\1 X", y))
print (re.sub('.*bb (.*) bb.*', "\\1", y))
print (re.sub('.*?bb (.*) bb.*', "\\1", y))
```

```
x zawiera d lub z
y zawiera dwa razy to samo
aa XX XX bb ff bb ee
aa XX XX XX ff XX ee
aa X cc bb ff X ee
ff
cc bb ff
```

Zwróć uwagę na:

- Działanie funkcji `match`, która dopasowuje wyrażenie do początku napisu (czyli tak jakby zaczynało się od `^`).
- Odwołania wsteczne do pod-wyrażeń (fragmentów ujętych w nawiasy) postaci `\\x`, gdzie `x` jest numerem pod-wyrażenia.
- „Zachłanność” (*greedy*) wyrażeń regularnych:
  - w pierwszym wypadku `bb (.*) bb` dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`,
  - w drugim przypadku gdy zostało poprzedzone `.*` dopasowało tylko `ff`, gdyż `.*` dopasowało najdłuższy możliwy fragment czyli `aa bb cc`,
  - w trzecim wypadku `bb (.*) bb` mogło i dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`, gdyż było poprzedzone niezachłanną odmianą dopasowania dowolnego napisu, czyli: `.*?`.

Po każdym z operatorów powtórzeń (`. ? + {n,m}`) możemy dodać pytajnik (`.? ?? +? {n,m}?`) aby wskazać że ma on dopasowywać najmniejszy możliwy fragment, czyli ma działać nie zachłannie.

Zadanie 1.3.1

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest słowem (tzn. nie zawiera spacji).

Zadanie 1.3.2

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest liczbą (tzn. jest złożony z cyfr i kropki, a na początku może wystąpić + albo -).

Zadanie 1.3.3

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis kończy się *xyz*.

Zadanie 1.3.4

Jak wiemy język złożony ze słów postaci `aa..aabb..bbaa..bb` (gdzie ilość liter a przed ciągiem liter b jest równa ilości liter a po tym ciągu) nie jest regularny. Jednak programistyczne wyrażenia regularne są rozszerzone w stosunku co do tych spotykanych w matematyce i umożliwiają opis takiego języka. Napisz funkcję, korzystającą z dopasowywania wyrażeń regularnych, która będzie sprawdzała czy podane słowo należy do tego języka.

1.4 Zadania dodatkowe

Zadanie 1.4.1

Zapoznaj się z dokumentacją klasy odpowiedzialnej za napisy (**str**), zwróć szczególną uwagę na metody `split`, `find`, `replace`. Korzystając z metod klasy **str** napisz funkcję `parse` która dla napisu będącego jej argumentem wykona zamianę wszystkich ciągów "XY" na spację oraz dokona rozbicia napisu złożonego z pól rozdzielanych dwukropkiem na listę napisów odpowiadających poszczególnym polom. Funkcja powinna działać w następujący sposób:

```
 > l = parse("Ala:maXYkota:i inne:zwierzeta")
 > print(l)
['Ala', 'ma kota', 'i inne', 'zwierzeta']
```

Zadanie 1.4.2

Napisz funkcję `toStr(liczba, podstawa)`, która konwertuje podaną liczbę do reprezentacji napisowej w systemie o podanej podstawie.
Wskazówka: do testowania poprawności działania możesz użyć funkcji **int**`(napis, podstawa)`, możemy przyjąć że podstawa jest mniejsza od 37 tak aby starczyło liter alfabetu łacińskiego.

Zadanie 1.4.3

Na poprzednich zajęciach mogłeś spotkać się z zadaniem polegającym na rysowaniu *trójkąta z iksów*, takiego jak poniżej:

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
```

Zmodyfikuj rozwiązanie tego zadania **??** tak aby wypisać taki trójkąt na ekranie, korzystając tylko z jednej pętli (oraz bez rekurencji).

2 Głębiej w Pythona (cz. 3)

2.1 Pliki

Do tej pory wszystkie dane, z których korzystały nasze programy, wprowadzaliśmy bezpośrednio do kodu programu. W realnych zastosowaniach bardzo często użyteczniejsze jest korzystanie z danych zapisanych w osobnych plikach.

2.1.1 Zapisywanie tekstu do pliku

Zapis do pliku tekstowego możemy zrealizować w sposób następujący:

```
plik = open('dane.txt', 'wt', encoding='utf8')
plik.write("teskt1\n")
plik.write("teskt2\nteskt3")
plik.close()
```

Jak to działa?

- Polecenie z pierwszej linijki otwiera plik `dane.txt` i zapewnia dostęp do niego poprzez zmienną `plik`. Opcja `'w'` oznacza, że plik jest otwarty „do zapisu” (od angielskiego *write*). Opcja `'t'` oznacza, że plik traktowany jako plik tekstowy[2]. Argument `encoding` pozwala na określenie kodowania użytego do zapisu pliku tekstowego, jest on opcjonalny i gdy nie zostanie podany kodowanie pliku zależne jest od ustawień systemowych.
- Druga i trzecia komenda zapisuje podany jako argument tekst do pliku `dane.txt` (zwróć uwagę na wstawianie nowej linii przy pomocy `'\n'`)
- Ostatnie polecenie zamyka dostęp do pliku `dane.txt`.

Po uruchomieniu powyższego kodu powinien zostać utworzony plik „dane.txt”, zawierający 3 linie tekstu. Jeżeli plik taki wcześniej istniał zostanie on nadpisany.

2.1.2 Wczytywanie tekstu z pliku

```
plik = open('dane.txt', 'rt', encoding='utf8')
for linia in plik:
  print(linia, end="")
plik.close()
```

[2] Tekst możemy zapisywać także do plików otwieranych jako binarne, w takim wypadku argument funkcji write musi mieć typ *bytes* a nie *str*, czyli być jawnie zakodowany w jakimś standardzie.

Zauważ, że została używa opcja `'r'` do otwarcia pliki co oznacza otwarcie do odczytu. Jeżeli chcemy wczytać cały plik do zmiennej napisowej możemy, zamiast pętli czytającej kolejne linie, użyć metody `read()`:

```
plik = open('dane.txt', 'rt', encoding='utf8')
napis = plik.read()
plik.close()
```

2.2 Zmienna, obiekt i referencja

W Pythonie każda zmienna jest nazwą wskazującą na jakiś obiekt w pamięci. Podobnie każdy element listy czy słownika wskazuje na jakiś obiekt[3]. Na jeden obiekt może wskazywać wiele zmiennych i/lub elementów innych obiektów (takich jak listy czy słowniki). Jeżeli zmienna nie ma na co wskazywać (np. został do niej przypisany wynik funkcji, która nie zwraca wartości) wskazuje na obiekt `None` (typu `NoneType`). Zatem na wszystkie zmienne pythonowe możemy patrzeć jak na referencje do obiektów istniejących gdzieś w pamięci.

Do uzyskania identyfikatora obiektu związanego z daną nazwą, lub elementem innego obiektu służy funkcja **id** (w przypadku standardowej implementacji Pythona jest to po prostu adres w pamięci).

2.2.1 Usuwanie i czas życia zmiennych

Instrukcja **del**, której używaliśmy już do usuwania elementów z listy lub słownika może być wykorzystana także do usuwania innych zmiennych. Należy jednak pamiętać iż w Pythonie usunięcie zmiennej nie wiąże się z natychmiastowym zwolnieniem zajmowanej przez nią pamięci z kilku powodów:

- na pojedynczy obiekt może wskazywać kilka zmiennych
- to Python decyduje o tym kiedy zwalniać / ponownie użyć pamięć pozostałą po obiektach na które nie wskazuje już żadna nazwa

2.2.2 Kopiowanie obiektów

Python w momencie przypisania wartości jednej zmiennej do innej nie tworzy kopii obiektu na który wskazuje zmienna, zamiast tego przypisuje referencję do istniejącego obiektu. Jest to szczególnie zauważalne w obiektach, które mogą być wewnętrznie modyfikowalne (takich jak listy czy słowniki)[4]:

```
a = [1, 2, 3]
b = a
print(a, b, "\n", hex(id(a)), hex(id(b)))
a[1] = 0
print(a, b, "\n", hex(id(a)), hex(id(b)))
del a
print(b,    "\n", hex(id(b)))
```

```
[1, 2, 3] [1, 2, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3] [1, 0, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3]
 0x7f50d76b2bc8
```

Jak widać `a` i `b` posiadają taki sam identyfikator obiektu zwracany przez funkcję **id**, modyfikacja `a[1]` wpłynęła na zawartość `b`, natomiast usunięcie `a` nie ma wpływu na `b` (usunęliśmy tylko jedną z dwóch referencji na wspólny obiekt). Jeżeli chcemy uzyskać kopię listy lub słownika musimy skorzystać z metody `copy()` odpowiedniego obiektu:

[3] Zasadniczo wszystkie definiowane przez nas zmienne czy funkcje są elementem słownika związanego z danym kontekstem. Do słowników tych można uzyskać dostęp poprzez funkcje **globals**() (słownik zawierający elementy zdeklarowane w kontekście globalnym) i **locals**() (słownik zawierający elementy zadeklarowane w kontekście lokalnym).

[4] Zauważ że jedyną możliwością modyfikacji liczby czy napisu jest przypisanie wartości wyrażenia do zmiennej, a dla list czy słowników możemy je modyfikować bez operacji przypisania całej listy czy słownika do nowej czy tej samej zmiennej.

```
a = [1, 2, 3]
b = a.copy()
b[1] = "X"
print(a, b, "\n", hex(id(a)), hex(id(b)))
```

```
[1, 2, 3] [1, 'X', 3]
 0x7f50d76b2bc8 0x7f50d57a7088
```

Zauważ że tak utworzone `b` ma inny identyfikator obiektu niż `a`. Należy mieć także na uwadze że nawet argumenty funkcji przekazywane są jako referencje na obiekty a nie kopie obiektów, natomiast dopiero operacja przypisania nowej wartości do zmiennej związanej z argumentem powoduje że zaczyna ona wskazywać na nowo utworzony (w wyniku wyrażenia po prawej stronie znaku równości) obiekt.

2.2.3 Dla jeszcze bardziej dociekliwych

Osobom jeszcze bardziej dociekliwym w temacie wnętrzności Pythona możemy polecić lekturę artykułu omawiającego te zagadnienia http://www.rwdev.eu/articles/objectthinking oraz samodzielne eksperymenty.

2.3 Klasy i struktury

Inną metodą grupowania zmiennych i funkcji jest definiowanie własnych klas:

```
class NazwaKlasy:
  # pola składowe
  a, d = 0, "ala ma kota"
  # metody składowe
  def wypisz(self):
    print(self.a + self.b)
  # metody statyczna
  def info():
    print("INFO")
  # konstruktor (z jednym argumentem)
  def __init__(self, x = 1):
    # i kolejny sposób na utworzenie pola składowego klasy
    self.b = 13 * x
```

Warto zauważyć jawny argument metod składowych klasy w postaci obiektu tej klasy. Możliwe jest także dziedziczenie po jednej lub kilku klasach bazowych, w tym celu definicje klasy rozpoczynamy:

```
class NazwaKlasy(Bazowa1, Bazowa2):
```

Tworzenie obiektu klasy i używanie go:

```
k = NazwaKlasy()
k.a = 67
k.wypisz()
```

```
80
```

Obiekty można rozszerzać o nowe składowe i funkcje:

```
k.c = k.a + 10
print(k.c)
```

```
77
```

W ten sposób można też tworzyć całe struktury:

```
class Pusta():
  pass

x = Pusta()
x.a = 3
x.b = 4
```

Od strony implementacyjnej są one trzymane w słowniku związanym z danym obiektem o nazwie `__dict__`.
Spróbuj wypisać zawartość `x.__dict__` oraz `k.__dict__`.

Do metod klasy możemy odwoływać się także z podaniem nazwy klasy a nie obiektu, w takim wypadku jeżeli nie są to metody statyczne należy przekazać jako argument obiekt danej klasy lub go udający[5]:

```
NazwaKlasy.info()
NazwaKlasy.wypisz(k)
NazwaKlasy.wypisz(x)
```

```
INFO
80
7
```


[5] Wystarczy żeby taki obiekt miał metody i składowe używane przez dana metodę, nie musi to być obiekt tej klasy.