Programowanie
przed 6 wykładem

Andrzej Giniewicz

12.04.2024

W tej części materiałów zajmiemy się napisami. Zaczniemy od przypomnienia i poszerzenia wbudowanych operacji na napisach, aby potem przejść do sprawdzania nawiasowania — jednego z klasycznych algorytmów na napisach — oraz nieco trudniejszego algorytmu szukania najdłuższego wspólnego podnapisu.

1 Podstawowe operacje na napisach

Zacznijmy od przypomnienia niektórych prostych operacji na napisach oraz omówienia kilku nowych, które możemy wykonać w Pythonie bez implementacji dodatkowych algorytmów. Na początek przypomnijmy sobie, że napis możemy zamienić na listę tak, aby podzielić go na znaki.

```
list(”napis”) == [”n”, ”a”, ”p”, ”i”, ”s”]
```

Co, jeśli chcemy podzielić dłuższy napis na słowa? Możemy użyć metody `split`.

```
”czy działa?”.split() == [”czy”, ”działa?”]
```

Metodzie split możemy podać jako argument znaki, na których ma dzielić napisy. Domyślnie dzieli na dowolnym ciągu białych znaków, czyli spacjach, tabulacjach, enterach itp. Do podziału tekstu na listę linijek, służy metoda `napis.splitlines()`, która działa podobnie, jak `napis.split(”\n”)`.

Operacją odwrotną do dzielenia napisów jest scalanie napisów. Robimy to metodą `join`.

```
”+”.join([”1”, ”2”, ”3”]) == ”1+2+3”
```

Zwróćmy uwagę, że Python wstawia napis ”+” pomiędzy każde dwa elementy listy. Zamiast napisu ”+” możemy użyć dowolnego innego napisu, w tym takiego składającego się z więcej niż jednego znaku.

Często na początku lub końcu napisu znajdują się białe znaki, czyli spacje, entery, tabulacje i inne znaki sterujące odstępami. Nie zawsze są nam potrzebne. Do ich usunięcia z lewej strony służy metoda `lstrip` a z prawej `rstrip`. Do usunięcia znaków z obu stron służy natomiast metoda `strip`.

```
” jakiś napis \t\n”.strip() == ”jakiś napis”
```

W napisach możemy też podmieniać ich fragmenty na inne wartości.

```
”tak, tak tak!”.replace(”tak”, ”nie”) == ”nie, nie nie!”
”nie, nie nie!”.replace(”nie”, ”tak”, 2) == ”tak, tak nie!”
```

Pierwszy wariant zamieni wszystkie wystąpienia napisu, drugi tylko określoną ich liczbę.

Mamy też możliwość modyfikowania liter małych na wielkie i odwrotnie.

```
napis = ”this is The day”
napis.lower() == ”this is the day”
napis.upper() == ”THIS IS THE DAY”
napis.swapcase() == ”THIS IS tHE DAY”
napis.capitalize() == ”This is the day”
napis.title() == ”This Is The Day”
```

Oprócz znanego już sprawdzania, czy napis zawiera się w drugim napisie za pomocą operatora `in`

```
”kot” in ”bojkot”
```

możemy sprawdzić, czy napis zaczyna się lub kończy innym

```
”bojkot”.startswith(”kot”) is False
”bojkot”.endswith(”kot”) is True
```

Możemy też sprawdzić, czy napis składa się z liter (`isalpha`), jest ciągiem cyfr (`isdecimal`, `isdigit`, `isnumeric`), jest ciągiem liter lub cyfr (`isalnum`), jest spacją (`isspace`), jest pisany małymi literami (`islower`), wielkimi literami (`isupper`) albo każde słowo zaczyna się wielką literą (`istitle`).

```
”tekst”.isalpha()
”12”.isdecimal()
”12”.isdigit()
”12”.isnumeric()
”tekst12”.isalnum()
” ”.isspace()
```

```
"tak".islower()
"TAK".isupper()
"Tak".istitle()
```

Warunek `isalnum` sprawdza, czy któryś z czterech wcześniejszych warunków zachodzi (`isalpha`, `isdecimal`, `isnumeric` lub `isdigit`). Definicja tych czterech warunków nie jest jednak prosta. Aby obsłużyć wiele różnych języków, korzysta z własności Unicode. I tak na przykład `isalpha` zwraca `True` dla wszystkich symboli, które w Unicode mają ustawioną opcję `General_Category` na „Letter", czyli litera[1]. Oznacza to, że będą rozpoznawać również litery z egzotycznych alfabetów. Różnice pomiędzy `decimal`, `digit` oraz `numeric` również są subtelne[2]. Zasadniczo `decimal` zawiera jedynie znaki, które w różnych językach oznaczają cyfry używane do zapisu dziesiętnego. Typ `digit` zawiera cyfry, których nie można używać do zapisu dziesiętnego, na przykład cyfry w wykładnikach, indeksach i kółkach, które w Unicode stanowią osobne znaki o osobnym kodzie. `numeric` zawiera w sobie również złożone symbole oznaczające liczby, na przykład znak oznaczający ½.

Jeśli dysponujemy listą napisów, możemy je posortować. Niestety, jak już wiemy, Unicode to nie wszystko i wciąż polskie znaki nie muszą dobrze działać. Zobaczmy, że Łucja jest w niespodziewanym miejscu.

```
sorted(["Karol", "Adam", "Łucja", "Zofia"]) == ['Adam', 'Karol', 'Zofia', 'Łucja']
```

Aby uzyskać prawidłowe sortowanie, musimy zrobić kilka rzeczy. Po pierwsze zaimportować moduł `locale` i ustawić w nim język na polski.

```
import locale
locale.setlocale(locale.LC_ALL, "pl_PL")
```

Parametr `LC_ALL` ustawia wszystkie aspekty językowe, natomiast `pl_PL` oznacza język polski, kraj Polska. Dla języka polskiego jest to jedyna możliwość, natomiast znaczenie ma to przy niektórych językach używanych w wielu krajach, przykładowo dla języka angielskiego są zdefiniowane `en_GB` oraz `en_US` (i kilka innych). Po ustawieniu języka, musimy jeszcze powiedzieć funkcji `sorted` lub metodzie `sort`, że ma z niego skorzystać. Robimy to, przekazując jako klucz funkcję `strxfrm` z modułu `locale`.

```
sorted(["Karol", "Adam", "Łucja", "Zofia"], key=locale.strxfrm)
```

zwróci oczekiwaną listę

```
['Adam', 'Karol', 'Łucja', 'Zofia']
```

[1] Patrz `https://unicode.org/reports/tr44/#General_Category_Values`.
[2] Patrz `https://unicode.org/reports/tr44/#Numeric_Type`.

1.1 Ostrzeżenie — niewłaściwe używanie znaków może grozić utratą życia lub zdrowia, przed operacjami na napisach skonsultuj się z typografem lub językowcem

Operacje na napisach nie są oczywiste. Czasem niewielka zmiana w pisowni powoduje, że coś staje się błędem, a czasem zmienia znaczenie, w tragiczny wręcz sposób. Na przykład wizualna forma liter połączonych w jedną, tak zwana ligatura, może powodować, że napis będzie zapisany błędnie. W języku niemieckim, w którym w słowie mogą wystąpić po sobie trzy litery „f", stosowanie ligatury „ff" może wprowadzać błędy do zapisu. Przyjrzyj się dobrze słowu „schifffahrt" oraz „schifffahrt". Pierwsze jest zapisane poprawnie, drugie z błędem.

Uwaga: osoby znające język turecki przepraszam za użycie słowa wulgarnego w tekście poniżej, zostało ono użyte w celu ilustracji potencjalnych problemów, gdy nie zna się zasad typograficznych panujących w danym kraju.

Jeśli sprawa ma się tak jak w tym niemieckim przykładzie, nie jest to tragiczne. O wiele bardziej niebezpieczny przykład możemy znaleźć w języku tureckim, który ma dwie literki „i", jedną z kropką, jak w naszym alfabecie, czyli „i" oraz drugą, bez kropki „ı". Co jest dość logiczne w tej sytuacji, wielka litera od „i" to nie „I" jak w większości języków, tylko „İ", ponieważ tam „I" jest wielką literą odpowiadającą małemu „ı". Łatwo wyobrazić sobie błąd. W Pythonie `"i".upper() == "I"`. Niestety, nie wszystkie systemy obsługują takie litery „i" oraz „ı", a jak się okazuje, nawet nie wszystkie telefony w Turcji obsługują takie zachowanie tekstu. Sprawa robi się o wiele bardziej interesująca, gdy dowiemy się, że słowo „Sıkısınca" oznacza w wolnym tłumaczeniu „przyprzeć do muru" np. w dyskusji, podczas gdy „Sikisinca" jest wulgarnym określeniem na odbycie stosunku seksualnego. Co więc się dzieje, gdy napiszecie w Pythonie `"SIKISINCA".lower()`? Słowo zmieni znaczenie i to bardzo. Tak bardzo, że dwie osoby straciły życie a trzy trafiły do więzienia z powodu SMS'a wysłanego pomiędzy telefonami, które nie miały takiej samej obsługi kodowania litery „i" oraz „ı"[3]. Przykład jest może skrajny, ale przytaczam go, aby pomóc zauważyć, że to, jak posługujemy się znakami, jest bardzo mocno zależne od języka i nigdy nie należy tego lekceważyć. A już na pewno nie należy publikować tekstów w obcym języku, nie znając zasad typograficznych i językowych rządzących danym krajem.

Niestety, trik z modułem `locale`, który rozwiązał problem z sortowaniem polskich znaków, nie pomaga w tej sytuacji. Jeśli tworzymy kod, który chcemy, żeby dobrze działał z wieloma językami, powinniśmy skorzystać z bardziej zaawansowanych bibliotek, na przykład PyICU. Instalacja tej biblioteki nie jest trywialna, ale zyski płynące z jej użytkowania są niekiedy wręcz ratujące życie. Zachęcam, aby poszukać informacji w Internecie na temat tego, jak zainstalować i używać tej biblioteki na waszym systemie operacyjnym (użycie jej na Windowsie może być nieco trudniejsze niż na innych systemach, ale proces wciąż jest wart zachodu, jeśli myślimy poważnie o umiędzynarodowieniu naszego programu).

[3]Osoby zainteresowane przeczytaniem, jak do tego doszło, odsyłam do artykułu w języku angielskim `https://gizmodo.com/a-cellphones-missing-dot-kills-two-people-puts-three-m-382026` lub tureckim `https://www.hurriyet.com.tr/gundem/kucucuk-bir-nokta-tam-5-kisiyi-yakti-8748359`.

2 Sprawdzanie nawiasowania

Sprawdzanie nawiasowania jest jednym z prostych, ale ważnych zadań. Na początek załóżmy, że mamy tylko jeden rodzaj nawiasów — okrągłe. Wtedy na przykład `(2+(3+1))` jest prawidłowo nawiasowane, ale `)2+3` nie jest prawidłowo nawiasowane. Naszym zadaniem będzie napisanie funkcji, która dla argumentu będącego napisem z nawiasami, zwróci wartość logiczną mówiącą, czy nawiasy są poprawnie umieszczone.

Aby sprawdzić jeden rodzaj nawiasów, wystarcza nam sprawdzanie znaków `(` i `)` oraz zliczanie ile nawiasów pozostało otwartych. Jeśli liczba otwartych nawiasów spadnie choć raz poniżej zera lub na końcu napisu pozostanie niezerowa, nawiasy się nie domykają.

```
def zbalansowane(tekst):
    otwarte = 0
    for znak in tekst:
        if znak == '(':
            otwarte += 1
        elif znak == ')':
            if not otwarte:
                return False
            otwarte -= 1
    return not otwarte
```

Spróbuj przeanalizować ten kod, aby przekonać się, że jest poprawny. Strona PythonTutor może Ci w tym pomóc.

Teraz spróbujemy nieco uogólnić tę funkcję, tak aby działała dla dowolnych nawiasów. Niestety nie wystarczy już zliczanie otwartych nawiasów, ponieważ nawias kwadratowy należy zamknąć kwadratowym, a nie innym. Musimy więc pamiętać listę otwartych nawiasów.

```
def zbalansowane(tekst):
    nawiasy = {'(': ')', '[': ']', '{': '}'}
    otwarte = []
    for znak in tekst:
        if znak in nawiasy.keys():
            otwarte.append(znak)
        elif znak in nawiasy.values():
            if not otwarte or nawiasy[otwarte[-1]] != znak:
                return False
            otwarte.pop()
    return not otwarte
```

Ponownie zachęcam do przetestowania funkcji na różnych napisach, na przykład

```
"[2+3]+({2-4}-2)"
```

PythonTutor będzie tutaj jeszcze bardziej pomocny niż przy ostatnim przykładzie.

Lista `otwarte`, którą wykorzystujemy w ten sposób, że doczepiamy elementy na końcu i zdejmujemy z końca, jest wykorzystana w charakterze struktury danych nazywanej stosem. Stosy są często wykorzystywaną strukturą w programowaniu.

3 Najdłuższy wspólny podnapis

Spróbujmy teraz odnaleźć najdłuższy wspólny podnapis dwóch napisów. Podobną funkcję moglibyśmy stworzyć dla najdłuższego wspólnego podciągu dwóch ciągów, ponieważ napisy możemy traktować jako ciągi znaków. Wykorzystamy technikę zapamiętywania wyników pośrednich, podobnie jak w implementacji liczb Fibonacciego, ale tym razem będziemy zapamiętywać wyniki ręcznie. Ta technika programowania nazywa się programowaniem dynamicznym.

Jeśli mamy dwa napisy: jeden długości n i drugi długości m, tworzymy macierz L wymiaru $(n+1)\times(m+1)$. W macierzy tej będziemy przechowywać liczby oznaczające długości wspólnych podnapisów. W miejscu $L_{j,i}$ będzie znajdować się długość wspólnego napisu znajdującego się na miejscu $(j - L_{j,i}, \ldots, j-1)$ w pierwszym napisie oraz $(i - L_{j,i}, \ldots, i-1)$ w drugim napisie. Jeśli napis pierwszy na współrzędnej j jest równy napisowi drugiemu na współrzędnej i, to $L_{j+1,i+1} = L_{j,i} + 1$. W przeciwnym razie $L_{j+1,i+1} = 0$, czyli napis nie jest wspólnym podnapisem. Jako warunek początkowy ustalamy $L_{0,i} = L_{j,0} = 0$ i próbujemy wypełniać macierz wierszami, dzięki czemu nigdy nie zabraknie nam wartości. Spójrzmy na macierz dla napisów `kotek` i `bojkot`, zawierającą na początek wartości początkowe.

$$\begin{array}{c} \\ . \\ k \\ o \\ t \\ e \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} . & b & o & j & k & o & t \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \end{pmatrix} \end{array}$$

Wypełnijmy teraz pierwszy wiersz. Jedynie tam, gdzie literki są równe, zwiększamy wartość na skos o jeden ($L_{j+1,i+1} = L_{j,i} + 1$).

$$\begin{array}{c} \\ . \\ k \\ o \\ t \\ e \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} . & b & o & j & k & o & t \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \end{pmatrix} \end{array}$$

W kolejnym wierszu o jeden zwiększamy te skosy, w których występuje literka „o”.

$$\begin{array}{c} \\ . \\ k \\ o \\ t \\ e \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} . & b & o & j & k & o & t \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & & & & & & \\ 0 & & & & & & \\ 0 & & & & & & \end{pmatrix} \end{array}$$

Dalej postępujemy analogicznie.

$$\begin{array}{c} \\ . \\ k \\ o \\ t \\ e \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} . & b & o & j & k & o & t \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Największa liczba w macierzy to 3, zatem najdłuższy wspólny podnapis jest długości 3 i kończy się na ostatnim znaku napisu „bojkot” lub analogicznie na 3 znaku napisu „kotek”. Oznacza to, że najdłuższy znaleziony wspólny podnapis, znajdujący się w obu napisach, to „kot”.

Formalnie, możemy wyliczyć maksimum wierszami. Uzyskamy wtedy wartości, które wystarczą nam do odczytania wspólnego podnapisu.

$$\begin{array}{c} \\ . \\ k \\ o \\ t \\ e \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} . & b & o & j & k & o & t \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 0 \\ 1 \end{array}$$

lub krócej

$$\begin{array}{cc} . & 0 \\ k & 1 \\ o & 2 \\ t & 3 \\ e & 0 \\ k & 1 \end{array}$$

Teraz wystarczy, począwszy od maksimum liczb, cofnąć się wskazaną liczbę znaków, w tym przypadku 3.

$$\begin{matrix} . & 0 \\ \left(\begin{matrix} k \\ o \\ t \end{matrix}\right) & \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \\ e & 0 \\ k & 1 \end{matrix}$$

Spróbujmy to zaimplementować. Macierz będziemy reprezentować jako listę list. Spróbuj wypisać ją zaraz po wypełnieniu wartościami komendą print i porównaj z wyznaczoną w przykładzie powyżej. Funkcję nazwiemy `LCS` od *Longest Common Substring*.

```
def LCS(napis1, napis2):
    m, n = len(napis1), len(napis2)
    L = [[0]*(n+1) for _ in range(m+1)]
    for i in range(n):
        for j in range(m):
            if napis1[j] == napis2[i]:
                L[j+1][i+1] = L[j][i]+1
    M = [max(wiersz) for wiersz in L]
    najdłuższy = max(M)
    if not najdłuższy:
        return []
    wyniki = []
    for j in range(1, m+1):
        if M[j] == najdłuższy:
            wyniki.append(napis1[j-najdłuższy:j])
    return list(set(wyniki))
```

Spróbuj uruchomić funkcję dla różnych wartości, prześledź jej działanie i przeanalizuj dla różnych przykładów. Jak w poprzednich przypadkach, PythonTutor może pomóc.

Zastanów się, w jaki sposób za pomocą najdłuższego wspólnego podciągu, znajdziesz najdłuższy podpalindrom długości większej niż 1 w napisie? Przez podpalindrom, mamy na myśli podnapis, który jest jednocześnie palindromem, czyli od początku i końca czyta się tak samo.

Czy w funkcji `LCS` jest coś, co działa tylko dla napisów? Czy można jej użyć dla list albo krotek? A czy można jej użyć dla zbiorów lub słowników? Postaraj się uzasadnić odpowiedź. Jeśli nie wiesz, patrząc tylko na kod, przepisz kod do Jupytera i sprawdź działanie funkcji `LCS` w praktyce.