REGEX – WYRAŻENIA REGULARNE

Wstęp do wyrażeń regularnych

Mając do dyspozycji napisy typu:

- ‘imie: Jan, nazwisko: Kowalski, wiek: 33’
- ‘imie: Anna, nazwisko: Kowalska, wiek: 28’
- itd…,

napisz funkcję przyjmującą jako parametr jeden string napisany w dokładnie takim samym formacie jak podanym wyżej, a zwracającą nazwisko osoby z takowego stringa.

Wyrażenia regularne

- Wyrażenia regularne to wzorce opisujące łańcuchy symboli. Możemy np. stworzyć wyrażenie, które będzie pasowało do każdego adresu email, każdej daty, numer telefonu, karty kredytowej itd.
- W Pythonie do posługiwania się wyrażeniami regularnymi jest nam potrzebny moduł o nazwie **re**.
- Moduł ten pomoże nam wyszukiwać ciągi pasujące do wzorca w tekście albo sprawdzanie czy dany tekst dokładnie pasuje do danego wzorca.
- Python używa tzw. Perlowej składni wyrażeń regularnych, którą poznamy za chwilę.

Wyrażenia regularne - składnia

- Wyrażenia regularne składają się ze zwykłych znaków oraz znaków specjalnych.
- Najprostsze wyrażenia regularne składają się wyłącznie ze zwykłych znaków.
- Przykładem prostego wyrażenia regularnego jest np “Ala”. To wyrażenie będzie znajdować w tekście wyłącznie wystąpienia wyrazu “Ala”.
- Wszystkie alfanumeryczne znaki (litery alfabetu oraz cyfry) są zwykłymi znakami.
- W wyrażeniach regularnych specjalne znaczenie mają następujące znaki:
  - kropka: **.**
  - nawiasy (okrągłe, kwadratowe, klamrowe): **( ) [ ] { }**
  - plus: **+**
  - minus: **-**
  - gwiazdka: ***
  - znak zapytania: **?**
  - pipe: **|**
  - dolar: **$**
  - daszek (kareta): **^**

Wyrażenia regularne - znaczenie znaków specjalnych

- Kropka w notacji wyrażeń regularnych oznacza dowolny znak z wyjątkiem znaku nowego wiersza. Np. do wyrażenia **.la** pasuje **Ola**, **Ala** i **Ela**.
- Nawiasy kwadratowe oznaczają dopasowanie do dowolnego ze znaków w tych nawiasach np. do wyrażenia **[OA]la** pasuje **Ola** i **Ala** ale nie pasuje **Ela**.
- Znak zapytania oznacza zero lub jedno wystąpnie, np. wyrażenie **Olk?a** pasuje do **Ola** i **Olka**.
- Plus oznacza jedno lub więcej wystąpienie, np. wyrażenie **a+le** pasuje do **ale**, **aaale**, **aaaale**.
- Gwiazdka oznacza zero, jedno lub wiele wystąpień, np. wyrażenie **a*la** pasuje do **la**, **ala**, **aaaala**.
- Nawiasy okrągłe pozwalają grupować znaki w wyrażeniu tak aby móc do nich zbiorczo stosować różne modyfikatory.
- Nawiasy klamrowe mówią o ilości powtórzeń np. **(ala){1,3}** oznacza ciąg ala występujący co najmniej jeden raz i maksymalnie 3 razy np. **ala**, **alaala**, **alaalaala** wszystkie pasują do tego wyrażenia.

Wyrażenia regularne - znaczenie znaków specjalnych

Stwórz wyrażenie regularne, które będzie pasowało do napisów:

- wierszowanka, wierszoowanka, wierszooowanka, wierszoooowanka
- koń, rum, tan, sin, żyć
- ANNA, PANNA, WANNA
- dwór, twór
- dwór, twór, wór
- wydra, wyydra, wydraa, wyyydraaaaaa
- wydra, wyydra, wydraa, wyyydraaaaaa, wdr, wydr, wdraaaa
- kura, kra -> na dwa sposoby
- do byle jakiego stringa, obojętnie jak długiego

Wyrażenia regularne - znaczenie znaków specjalnych

- Jeśli treść podana w kwadratowych nawiasach zaczyna się od daszka to mamy do czynienia z negacją przedziału, to znaczy do wyrażenia pasuje każdy znak spoza listy np. do wyrażenia **[^OA]la** pasuje **Ela** i **Bla** ale nie pasuje **Ola** i **Ala**.
- Jeśli w nawiasie kwadratowym znajduje się znak ‘-’ to oznacza on zakres np. **[a-z]** oznacza wszystkie małe litery alfabetu łacińskiego a **[0-9]** oznacza wszystkie cyfry.
- Pionowa kreska czyli pipe oznacza alternatywę np. wyrażenie **ala|kota** będzie pasowało do słowa **ala** lub do słowa **kota**.
- Daszek oznacza początek wiersza.
- Dolar oznacza koniec wiersza.
- Jeśli chcemy użyć jakiegoś znaku, który jest specjalny, ale tak aby był potraktowany jako zwykły (czyli dosłownie) to powinniśmy go poprzedzić backslashem **\.**, *****, itd.
- **\d** oznacza cyfrę i jest aliasem dla **[0-9]**.
- **\s** oznacza dowolny biały znak
- **\w** oznacza słowo i jest aliasem dla **[a-zA-Z0-9_]**
- **\D**, **\S**, **\W** są negacjami **\d**, **\s**, **\w** pasują do wszystkiego do czego nie pasują ich odpowiedniki.

Wyrażenia regularne - znaczenie znaków specjalnych

Stwórz wyrażenie regularne, które będzie pasowało do napisów/zdań:

- ciąg liczba parzysta nieparzysta parzysta nieparzysta, np. 4567, 2589
- Ala ma kota, Ola ma psa, Ela ma papugę
- dwa słowa pięcioliterowe
- nazwa dowolnej zmiennej bez cyfr
- Python jest super… -> tylko taki string, żaden inny
- adres email, np. konto123@gmail.com, jan.kowalski@poczta.pl
- numer telefonu w formacie „+48 654 321 123”

Wyrażenia regularne - funkcjonalność modułu re

- Funkcja **search** przyjmuje dwa parametry. Pierwszym jest wyrażenie regularne, drugim tekst, w którym szukamy ciągu znaków pasującego do wyrażenia. Jeśli funkcja zwróci **None** to znaczy, że nie znaleziono żadnego pasującego ciągu znaków. Jeśli udało się znaleźć dopasowanie to zwrócony zostanie obiekt **Match**, który zawiera informację o tym jaki ciąg dopasował się do wyrażenia oraz jakie jest jego położenie w tekście.

```
In [16]: import re

In [17]: match = re.search(r".la", "My name is Ala")

In [18]: match
Out[18]: <_sre.SRE_Match object; span=(11, 14), match='Ala'>

In [19]: match.group()
Out[19]: 'Ala'
```

Wyrażenia regularne - funkcjonalność modułu re

- Funkcja **match** przyjmuje dokładnie takie same parametry jak **search**. Różnica polega na tym, że funkcja ta informuje czy początek tekstu pasuje do wyrażenia a nie tylko jego fragment.

```
In [20]: import re

In [21]: match = re.match(r".la", "My name is Ala")

In [22]: match

In [23]: match.group()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-23-bf08e9dfb841> in <module>
----> 1 match.group()

AttributeError: 'NoneType' object has no attribute 'group'
```

```
In [24]: match = re.match(r".la", "Ala ma kota")

In [25]: match
Out[25]: <_sre.SRE_Match object; span=(0, 3), match='Ala'>
```

Wyrażenia regularne - funkcjonalność modułu re

- Funkcja **fullmatch** również przyjmuje dokładnie takie same parametry. Tym razem sprawdzane jest czy cały tekst pasuje do wyrażenia.

```
In [27]: match = re.fullmatch(r".la", "My name is Ala")

In [28]: match

In [29]: match = re.fullmatch(r"\w*\s\w{4} is .la", "My name is Ala")

In [30]: match
Out[30]: <_sre.SRE_Match object; span=(0, 14), match='My name is Ala'>
```

Wyrażenia regularne - funkcjonalność modułu re

- Funkcja **findall** zwraca wszystkie wystąpienia wyrażenia regularnego w tekście. Zwracana jest lista wyników (obiektów typu **Match**)

```
In [31]: match = re.findall(r".la", "Ala ma kota, a Ola ma psa")

In [32]: match
Out[32]: ['Ala', 'Ola']
```

- Funkcja **finditer** działa podobnie do **findall** ale zamiast wrócić na koniec pełną listę wyników zwraca leniwy iterator który zwraca kolejne wyniki w miarę jak po nich przechodzimy.

```
In [46]: match = re.finditer(r".la", "Ala ma kota, a Ola ma psa")

In [47]: for m in match:
    ...:     print(m)
    ...:
<_sre.SRE_Match object; span=(0, 3), match='Ala'>
<_sre.SRE_Match object; span=(15, 18), match='Ola'>
```

Wyrażenia regularne - funkcjonalność modułu re

- Funkcja **split** z modułu **re** działa podobnie do metody **split** dostarczanej przez klasę **str**. Różnica polega na tym, że możemy podać wyrażenie regularne, względem którego dzielimy.

```
In [48]: new_string = re.split(r".la", "Imiona ich to Ola, Ala oraz Ula")

In [49]: new_string
Out[49]: ['Imiona ich to ', ', ', ' oraz ', '']
```

- Funkcja **sub** zamieni wszystkie ciągi opisane wyrażeniem regularnym na podany ciąg znaków a jej wariant **subn** zwróci również informację o tym ile zamian przeprowadzono.

```
In [50]: re.sub(r".la", "Kuba", "Imiona ich to Ola, Ala oraz Ula")
Out[50]: 'Imiona ich to Kuba, Kuba oraz Kuba'

In [51]: re.subn(r".la", "Kuba", "Imiona ich to Ola, Ala oraz Ula")
Out[51]: ('Imiona ich to Kuba, Kuba oraz Kuba', 3)
```

Wyrażenia regularne - tryby dopasowania

- Wszystkie wymienione funkcje przyjmują również opcjonalnie flagi, które decydują w jakim trybie następuje dopasowanie. Poniżej wymienimy najważniejsze z nich:
  - **re.I** zaniedbuje wielkość znaków podczas dopasowania
  - **re.A** dokonuje dopasowania wyrażeń **\w**, **\W**, **\b**, **\B**, **\d**, **\D**, **\s, \S** jedynie według znaków ASCII, w tym trybie słowo **wąż** nie będzie pasować do wyrażenia **\w+**.
  - **re.L** dokonuje dopasowania wyrażeń **\w**, **\W**, **\b**, **\B** według lokalnych ustawień językowych.
  - **re.U** dokonuje dopasowania wyrażeń **\w**, **\W**, **\b**, **\B**, **\d**, **\D**, **\s, \S** według standardu Unicode. Uaktywania znaki spoza ASCII.
  - **re.S** sprawia, że kropka dopasowuje się również do znaku końca linii.
  - **re.M** (multiline), sprawia że daszek pasuje do początku dowolnej linii w tekście a nie tylko początku całego tekstu natomiast dolar pasuje do końca dowolnej linii a nie jedynie końca całego tekstu.

Wyrażenia regularne - przykłady

```
In [1]: import re

In [2]: print(re.search(r'ala', 'ala ola ela'))
<re.Match object; span=(0, 3), match='ala'>

In [3]: print(re.search(r'.la', 'ala ola ela'))
<re.Match object; span=(0, 3), match='ala'>

In [4]: print(re.findall(r'.la', 'ala ola ela'))
['ala', 'ola', 'ela']

In [5]: print(re.findall(r'Ala', 'ala ola ela'))
[]

In [6]: print(re.findall(r'Ala', 'ala ola ela', re.I))
['ala']

In [7]: print(re.match('\w+', 'wąż'))
<re.Match object; span=(0, 3), match='wąż'>

In [8]: print(re.match('\w+', 'wąż', re.A))
<re.Match object; span=(0, 1), match='w'>

In [9]: print(re.fullmatch('\w+', 'wąż', re.A))
None

In [10]: print(re.fullmatch('\w+', 'wąż', re.U))
<re.Match object; span=(0, 3), match='wąż'>
```

```
In [1]: import re

In [2]: re.sub(r'\w{4}', 'psa', 'Ala ma kota')
Out[2]: 'Ala ma psa'

In [3]: re.subn(r'\w{4}', 'psa', 'Ala ma kota')
Out[3]: ('Ala ma psa', 1)

In [4]: it = re.finditer(r'.la', 'ola ala ela')

In [5]: for match in it:
   ...:     print(match)
   ...:
<re.Match object; span=(0, 3), match='ola'>
<re.Match object; span=(4, 7), match='ala'>
<re.Match object; span=(8, 11), match='ela'>
```

Wyrażenia regularne - przykłady

- Grupowanie pozwala na wydobywanie interesujących nas informacji z napisów.
- Grupujemy część stringa we wzorcu opakowując dany fragment w nawiasy.
- W przypadku uzyskania matcha, możemy się do zgrupowanych wyrazów odnieść poprzez match.group(numer).

```
In [63]: match = re.match(r"To sa (.la) i (\w{3})", "To sa Ala i Ola")

In [64]: if match is not None:
    ...:     imie1, imie2 = match.group(1), match.group(2)
    ...:

In [65]: imie1
Out[65]: 'Ala'

In [66]: imie2
Out[66]: 'Ola'
```

Wstęp do wyrażeń regularnych

Powrót do przeszłości…

Mając do dyspozycji napisy typu:

- ‘imie: Jan, nazwisko: Kowalski, wiek: 33’
- ‘imie: Anna, nazwisko: Kowalska, wiek: 28’
- itd…,

napisz funkcję przyjmującą jeden taki string jako parametr, a zwracającą nazwisko osoby z takowego stringa.

Wykorzystaj do tego funkcje search, match, fullmatch oraz findall.

Wyrażenia regularne - przykłady

- Istnieje pewne bardzo prosto brzmiące zadanie, które wyjątkowo trudno wykonać bez znajomości wyrażeń regularnych. Wyobraźmy sobie, że tekst zapisany camel casem (**CzyliDokładnieTak**) musimy przekształcić w listę, której każdy element jest oddzielnym słowem (**[‘Czyli’, ‘Dokładnie’, ‘Tak’]**).
- Pierwszym skojarzeniem jest funkcja **split** z klasy **str** ale tam trzeba podać jeden stały ciąg, który rozdziela elementy. Dlatego to wyjście odpada.
- Wydaje się w takim razie, że użycie funkcji **split** z modułu **re** będzie tutaj idealne. Problem polega na napisaniu odpowiedniego wyrażenia regularnego.
- Naiwnie wydaje się, że tym wyrażeniem będzie: każda wielka litera czyli **[A-Z]** ale to jest błąd - jeśli potraktujemy wielką literę jako coś co rozdziela wyrazy to zostanie ona z nich usunięta a tego nie chcemy.
- W takim razie powinniśmy dodać do wyrażenia grupę przechwytującą w przód (lookahead assertion): **(?=[A-Z])**.
- Użycie grupy przechwytującej sprawi, że również litera, po której rozdzielamy znajdzie się w wyniku, ponieważ w tej chwili rozdzielamy tak naprawdę po pustym stringu, przed którym stoi wielka litera.
- Właśnie dlatego w poprawionej wersji pierwszy element jest zawsze pustym stringiem.
- Łatwo zauważyć, że jeśli wejściowy string składa się z wyłącznie z wielkich liter, to zostaną one rozdzielone, jeśli chodzi nam żeby następujące po sobie wielkie litery były razem połączone to zadanie robi się jeszcze trudniejsze. Jego rozwiązanie można znaleźć tutaj.

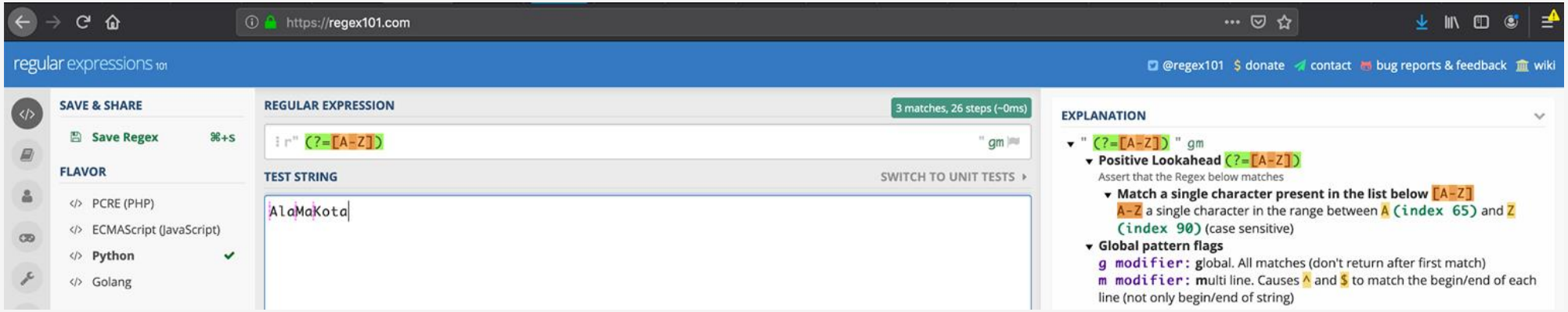
Autor: Michał Nowotka

Wyrażenia regularne - przykłady

```
In [11]: print(re.split(r'[A-Z]', 'AlaMaKota'))
['', 'la', 'a', 'ota']
```

```
In [13]: print(re.split(r'(?=[A-Z])', 'AlaMaKota'))
['', 'Ala', 'Ma', 'Kota']
```

```
In [19]: print(re.split(r'(?=[A-Z])', 'UPPER'))
['', 'U', 'P', 'P', 'E', 'R']
```



Do testowania wyrażeń regularnych bardzo przydaje się strona regex101.com