



# Python

## Poziom średniozaawansowany

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy



1. Praca z plikami tekstowymi (.csv) – część 2
2. Serializacja - pickle
3. Wyrażenia regularne



# **OPERACJE NA PLIKACH**



- CSV to skrót od *comma separated values* (wartości rozdzielone przecinkiem).
- Są to pliki tekstowe o specjalnej strukturze przeznaczone do przechowywania danych tabularycznych.
- Wartości oddzielone są przecinkiem (lub innym separatorem np. tabulatorem, średnikiem).
- Jest to format przechowywania danych w plikach typu text/csv (to znaczy, że jest to jakiś znany i uznawany format).

```
1 "Index", "Year", "Age", "Name", "Movie"
2 1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
3 2, 1929, 41, "Warner Baxter", "In Old Arizona"
4 3, 1930, 62, "George Arliss", "Disraeli"
5 4, 1931, 53, "Lionel Barrymore", "A Free Soul"
6 5, 1932, 47, "Wallace Beery", "The Champ"
```



## UWAGA:

- Spacje i inne białe znaki (w szczególności te przyległe do separatorów) należą do pól.
- Pierwsza linia może stanowić nagłówek zawierający nazwy pól rekordów, jednak pierwszy wiersz pliku CSV wg standardu ma takie samo znaczenie jak pozostałe.
- Zazwyczaj pierwszy wiersz określa nazwy kolumn.

```
1 "Index", "Year", "Age", "Name", "Movie"
2 1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
3 2, 1929, 41, "Warner Baxter", "In Old Arizona"
4 3, 1930, 62, "George Arliss", "Disraeli"
5 4, 1931, 53, "Lionel Barrymore", "A Free Soul"
6 5, 1932, 47, "Wallace Beery", "The Champ"
```



- Do obsługi plików csv służy moduł csv.
- Plik otwieramy tak jak w przypadku normalnych plików tekstowych (zalecane: menadżer kontekstu with).
- Funkcja do odczytu pliku nazywa się reader i znajduje się w module csv.
- Jako argumenty podajemy jej uprzednio otwarty plik oraz znak delimitera (przecinek, tabulator, średnik).

```
In [1]: import csv
```

```
In [2]: with open("plik.csv", "r") as csvfile:  
...:     csvreader = csv.reader(csvfile, delimiter=",")  
...:
```



- Po wykonaniu funkcji reader zwracającej iterator, możemy po nim przeiterować i uzyskać dostęp do każdego wiersza z pliku po kolei.

```
In [11]: with open("F:\pdf\plik.csv", "r") as csvfile:
...:     csvreader = csv.reader(csvfile, delimiter=",")
...:     for row in csvreader:
...:         print(row)
```

```
['Index', ' "Year"', ' "Age"', ' "Name"', ' "Movie"']
[' 1', ' 1928', ' 44', ' "Emil Jannings"', ' "The Last Command', ' The Way of All Flesh"']
[' 2', ' 1929', ' 41', ' "Warner Baxter"', ' "In Old Arizona"']
[' 3', ' 1930', ' 62', ' "George Arliss"', ' "Disraeli"']
[' 4', ' 1931', ' 53', ' "Lionel Barrymore"', ' "A Free Soul"']
[' 5', ' 1932', ' 47', ' "Wallace Beery"', ' "The Champ"']
```

- Każdy wiersz zostaje zapisany jako lista elementów (na podstawie delimitera (najczęściej przecinka) z pliku; UWAGA – liczby są zapisywane jako stringi!



- Oprócz funkcji reader z modułu csv, do odczytu danych z pliku csv możemy wykorzystać klasę DictReader.

```
: with open("F:\pdf\plik.csv", "r") as csvfile:
:     csvreader = csv.DictReader(csvfile)
:     for row in csvreader:
:         print(row)
```



# Pliki CSV



- Obiekt typu DictReader odczytuje zawartość pliku csv z wykorzystaniem słownika, w którym kluczami są nazwy kolumn z pliku (ustalane na podstawie wartości pierwszego wiersza w pliku).

```
1 "Index", "Year", "Age", "Name", "Movie"
2 1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
3 2, 1929, 41, "Warner Baxter", "In Old Arizona"
4 3, 1930, 62, "George Arliss", "Disraeli"
5 4, 1931, 53, "Lionel Barrymore", "A Free Soul"
6 5, 1932, 47, "Wallace Beery", "The Champ"
```

- Oto przykład odczytania pierwszego wiersza z pliku z wykorzystaniem klasy DictReader.

```
OrderedDict([('Index', ' 1'), (' "Year"', ' 1928'), (' "Age"', ' 44'), (' "Name"', ' "Emil Jannings"'),  
(' "Movie"', ' "The Last Command'), (None, [' The Way of All Flesh'])])
```

- OrderedDict to słownik, który pamięta kolejność par klucz: wartość, które zostały do niego dodane (poza tym zachowuje się jak zwykły dict).



Odczytaj zawartość pliku students.csv z repozytorium i znajdź ucznia oraz uczennicę o najwyższym wzroście.



- Celem zapisania danych do formatu csv, korzystamy z funkcji writer z modułu CSV.

```
with open("F:\pdf\plik.csv", mode="w") as csvfile:  
    writer = csv.writer(csvfile, delimiter=";", quotechar='"')  
    writer.writerow([6, 1950, 44, "An actor", "Any film"])
```

- Argumenty: csvfile – nasz otwarty plik, delimiter – znak oddzielający od siebie kolumny danych, quotechar – jakimi znakami będziemy oznaczać napisy (albo 'napis' albo "napis")
- Funkcja writerow obiektu zwróconego przez funkcję csv.writer zapisuje do pliku pojedynczy wiersz: każdy element to nowa kolumna w wierszu oddzielona od innych znakiem delimitera

# Pliki CSV



- Podobnie jak w przypadku odczytu i przy zapisywaniu możemy skorzystać z słownikowego zarządzania plikami csv – zapis do pliku jest możliwy z wykorzystaniem klasy DictWriter.

```
with open('F:\pdf\plik.csv', mode='w', newline='') as csv_file:
    column_names = ['id', 'name']
    writer = csv.DictWriter(csv_file, fieldnames=column_names)

    writer.writeheader()
    writer.writerow({"id": 1, "name": "Anna"})
```

- Na początku tworzymy listę z nazwami kolumn, jakie będziemy chcieli umieścić w pliku (u nas lista nazywa się column\_names)
- Przekazujemy tę listę jako parametr fieldnames, po czym metodą writeheader obiektu writer zapisujemy pierwszy wiersz z nazwami kolumn do pliku csv.



- `writer.writerow` zapisuje każdy kolejny wiersz do pliku.
- Metoda ta wymaga podania słownika z nazwami kluczy odpowiadającymi nazwom kolumn.

```
with open('F:\pdf\plik.csv', mode='w', newline='') as csv_file:
    column_names = ['id', 'name']
    writer = csv.DictWriter(csv_file, fieldnames=column_names)

    writer.writeheader()
    writer.writerow({"id": 1, "name": "Anna"})
```

- Parametr `newline` w funkcji `open` zapobiega dodatkowemu dodaniu nowej linii po każdym nowym wierszu w pliku csv.



Pobierz zawartość pliku `snake_game.csv` z repozytorium za pomocą biblioteki `csv`. Zaproponuj i stwórz strukturę przechowującą dane na temat przebiegu gry. Dla takich danych napisz:

- a) funkcję sortującą dane, powinny być one poukładane według ilości zdobytych punktów (rosnąco),
- b) funkcję obliczającą średnią ilość zdobytych punktów.
- c) funkcję dodającą nowy wiersz danych do pliku `csv` (funkcja powinna automatycznie inkrementować indeks danych)



**PICKLE**



- Pickle to biblioteka używana do serializacji i deserializacji obiektów pythonowych,
- Do tej pory zapisywaliśmy do pliku dane tekstowe, okazuje się, że to nie jedyne co potrafi Python,
- Każdy **obiekt** może zostać zserializowany i zapisany na dysku.
- Biblioteka pickle potrafi przekonwertować obiekt (list, set, dict itd.) do strumienia znaków.
- Taki strumień posiada wszystko, co jest potrzebne, do odtworzenia takiego obiektu w dogodnym czasie.





- Do zserializowania obiektu, wykorzystujemy funkcję `dump` z biblioteki `pickle`.
- Funkcja ta przyjmuje dwa argumenty: obiekt, który chcemy zapisać do pliku oraz reprezentację tego pliku.
- Przy otwieraniu pliku w trybie do zapisu należy pamiętać, że otwierany jest on w trybie binarnym(!), stąd flaga trybu powinna być ustawiona na `'wb'`

```
In [1]: import pickle

In [2]: testowy_slownik = {
...:     "jeden": 1,
...:     "dwa": 2,
...:     "trzy": 3,
...:     "cztery": 4
...: }

In [3]: with open("F:\pdf\serialize.txt", "wb") as binary_file:
...:     pickle.dump(testowy_slownik, binary_file)
...:
```



- Po wykonaniu funkcji `pickle.dump` do pliku trafia zserializowany obiekt w postaci strumienia znaków.
- Charakterystyczne znaczkiki (krzaczki) świadczą o zawartości binarnej pliku.
- Od tego momentu obiekt wraz z jego stanem i zawartością znajduje się bezpiecznie w pliku. Kiedy nadejdzie potrzeba, będzie go można odtworzyć.

 serialize.txt — Notatnik

Plik Edycja Format Widok Pomoc

```
|€|}q (X|  jedenq|K|X|  dwaq K X|  trzyq|K|X|  czteryq|K|u.
```



- Do odtworzenia zserializowanego wcześniej obiektu, służy funkcja `load` z biblioteki `pickle`.
- Funkcja ta wymaga jednego argumentu: otwartego wcześniej pliku, w którym znajduje się zapisany obiekt. Wydobyty obiekt można przypisać do nowej zmiennej.
- Należy pamiętać o otwarciu pliku w trybie binarnym! Podajemy tryb `'rb'`.

```
In [4]: pusty_slownik = {}
```

```
In [5]: with open("F:\pdf\serialize.txt", "rb") as binary_file:
...:     pusty_slownik = pickle.load(binary_file)
...:
```

```
In [6]: pusty_slownik
```

```
Out[6]: {'jeden': 1, 'dwa': 2, 'trzy': 3, 'cztery': 4}
```



- Dzięki pickle staje się możliwa wymiana obiektów w Pythonie między różnymi maszynami wirtualnymi, środowiskami, a nawet komputerami oddalonymi od siebie o wiele kilometrów.
- Zapisany obiekt zostaje zachowany i jest gotowy do odtworzenia nawet po ponownym włączeniu komputera.
- Zserializowane obiekty można umieścić w bazach danych.
- Należy pamiętać, że korzystanie z pickle i deserializacji obiektów nie jest w pełni bezpieczne! „Wczytując” z pliku obiekt nieznanego pochodzenia narażamy się na kłopoty – obiekt może działać jak mały wirus.



# **REGEX – WYRAŻENIA REGULARNE**



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 wyrażen 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 wystąpienie, 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, życie
- 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'>
```

Autor: Michał Nowotka

# 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** zaniebija wielkość znaków podczas dopasowania
  - **re.A** dokonuje dopasowania wyrażen `\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 wyrażen `\w`, `\W`, `\b`, `\B` według lokalnych ustawień językowych.
  - **re.U** dokonuje dopasowania wyrażen `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, `\S` według standardu Unicode. Uaktywiania 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+', 'wqż'))
<re.Match object; span=(0, 3), match='wqż'>

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

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

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

```
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'
```



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 przechwytyjącą w przód (lookahead assertion): **(?=[A-Z])**.
- Użycie grupy przechwytyjącej 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](#).

# 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']
```

regular expressions

SAVE & SHARE

Save Regex

FLAVOR

- PCRE (PHP)
- ECMAScript (JavaScript)
- Python
- Golang

REGULAR EXPRESSION

3 matches, 26 steps (~0ms)

TEST STRING

AlaMaKota

EXPLANATION

- ▼ `(?=[A-Z])` gm
- ▼ Positive Lookahead `(?=[A-Z])`  
Assert that the Regex below matches
  - ▼ Match a single character present in the list below `[A-Z]`  
`A-Z` a single character in the range between `A` (index 65) and `Z` (index 90) (case sensitive)
- ▼ Global pattern flags
  - `g` modifier: global. All matches (don't return after first match)
  - `m` modifier: multi line. Causes `^` and `$` to match the begin/end of each line (not only begin/end of string)

Do testowania  
wyrażeń regularnych  
bardzo przydaje się  
strona [regex101.com](https://regex101.com)

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

# Wyrażenia regularne - kiedy ich nie używać?



- Kiedy nauczysz posługiwać się młotkiem wszystko wygląda jak gwóźdź - podobnie jest z nauką wyrażeń regularnych.
- Wyrażenia regularne to odrębna notacja, znacznie mniej czytelna niż pythonowy kod. Dlatego należy unikać stosowania bardzo złożonych wyrażeń regularnych ponieważ bardzo trudno się je debuguje.
- Być może kiedyś po kilku miesiącach wrócisz do własnego kodu i zobaczysz w nim wyrażenie regularne ale nie będziesz pamiętać co ono znaczy a jego analizowanie okaże się bardzo trudne.
- Wyrażenia regularne potrafią opisać tylko pewną klasę ciągów znaków, nie można ich stosować do złożonych gramatyk, takich jak XML. Jeśli chcesz zrozumieć dlaczego popatrz [tutaj](#).