**Regex.**

Wyrażenia regularne.

**Czym jest ReGex?**

**Wyrażenie regularne (ang. regular expression, regex, regexp)** - to wzorzec opisujący łańcuch symboli, znaków.

Wyrażenia te znalazły bardzo szerokie zastosowanie, pozwalają bowiem w łatwy sposób opisywać wzorce tekstu oraz umożliwiają w efektywny sposób określić, czy podany ciąg znaków pasuje do danego wzorca.

Pamiętasz, z szkoleń dotyczących łańcuchów tekstowych, metodę **find()**? Pozwalała ona znaleźć indeks konkretnego znaku/podciągu w tekście lub określić, czy w ogóle znajduje się on w tekście.

Przyznaj jednak, że standardowe wykorzystanie metody find i poszukiwanie konkretnych wartości w danym łańcuchu tekstu, np.

txt = "Example1"

txt.find('ple1')

jest dość ograniczone, ponieważ musimy uciekać się do konkretnych wartości (w tym przypadku **ple1**).

A co, gdybyśmy chcieli sprawdzić, czy w danym tekście znajdują się, np. 3 dowolne cyfry i następuje po nich pojedyncza litera?

Rozwiązując ten problem moglibyśmy zastosować podejście mocno algorytmiczne, bo np. przechodzilibyśmy pętlą po każdym znaku w tekście, wyszukalibyśmy literę oraz sprawdzali, czy trzy poprzedzające ją znaki są cyframi.

Ale to oczywiście nie wszystko! Musielibyśmy się jeszcze zabezpieczyć przed próbą przeanalizowania łańcucha składającego się z mniej niż 3 liter oraz zapewne przed kilkoma jeszcze innymi edge case'ami, o których łatwo zapomnieć...

Remedium dla Naszego problemu, mogłoby stać się właśnie wykorzystanie wyrażeń regularnych.

Wystarczyłoby bowiem zastosować metodę (o której za chwilę) sprawdzającą, czy dany pattern (zapisany przy wykorzystaniu regex'a; w naszym przypadku byłyby to 3 cyfry zakończone literą) jest obecny w tekście.

**Po co ten cały ReGex?**

Zanim przejdziemy do konkretów, chciałbym zwrócić jeszcze uwagę na to, gdzie najczęściej możemy wykorzystywać wyrażenia regularne:

* Tworząc uniwersalne i efektywne rozwiązania do problemów bazujących na trudnościach w znalezieniu pewnych wzorców w tekście
* Przetwarzając długie teksty, np. e-maile i chcąc wyłuskać z nich typowe frazy świadczące o spamie. Technika ta może okazać się niezwykle przydatna przy trenowaniu własnego modelu wykorzystującego algorytmy uczenia maszynowego.
* Przy web scrapingu podczas procesu rozpoznawczego (w cyberbezpieczeństwie, tzw. rekonesans), szukając luk w serwisach i potencjalnych wycieków danych.

**Bardziej przyziemne przykłady:**

Analizując natomiast przegląd możliwych zastosowań ReGex przy codziennej pracy z kodem to choćby przypadki, gdy chcemy:

* Przeanalizować tekst pod kątem obecności pewnych wzorców
* Porównać cały tekst z dowolnym wzorcem (patternem)
* Przeprowadzić proces strings slicing’u
* Gdy chcemy sformatować tekst i zmodyfikować jego części na inne

**Pierwszy przykład użycia:**

Aby jeszcze lepiej uzmysłowić Ci, czym są omawiane wyrażenia, wyobraź sobie, że ktoś pyta Cię, jakiego formatu jest Twój kod pocztowy.

Wówczas odpowiedziałbyś, że:

składa się w sumie z 5 cyfr z przedziału 0-9, z czego dwie pierwsze są oddzielone myślnikiem od pozostałych

Gdybyś natomiast chciał tę informację przedstawić komputerowi, z wykorzystaniem ReGex, to zapisałbyś coś w podobie do:

**[0-9]{2}-[0-9]{3}** lub **\d{2}-\d{3}**

a swój 9-cio cyfrowy numer telefonu jako:

**\d{9}** lub **[0-9]{9}**

lub jeszcze na wiele innych możliwości!

**Wsparcie przy opanowywaniu wyrażeń regularnych!**

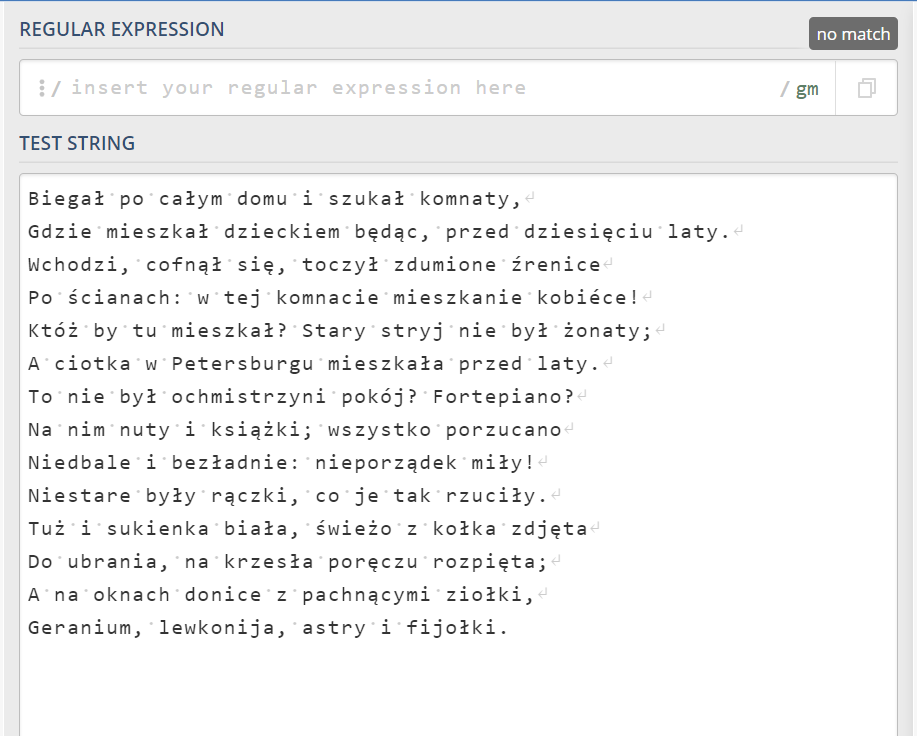
Chcąc skutecznie uczyć się zarówno składni wyrażeń regularnych, jak i również schematów ich tworzenia, niewątpliwie bezcenną pomocą mogą okazać się internetowe narzędzia do testowania pisanych wyrażeń.

Osobiście polecam Ci: [https://regex101.com/](https://regex101.com/%20) i właśnie przy wykorzystaniu tej strony, będziemy tworzyli Nasze pierwsze ReGexy!

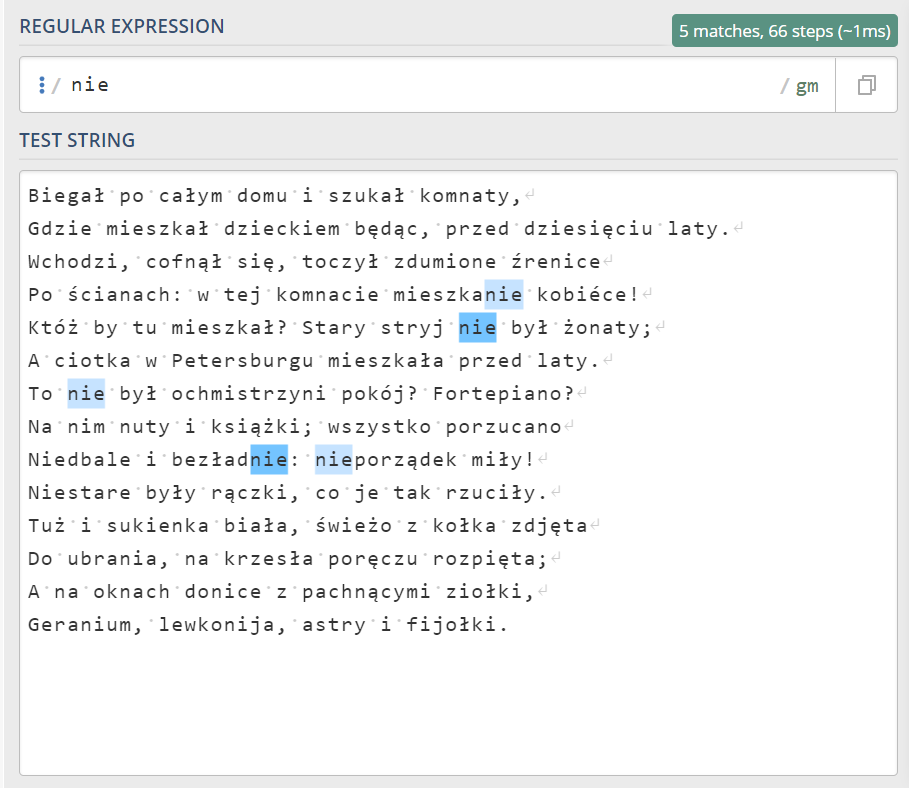
**Analiza narzędzia**

Zanim przejdziemy do teorii i zmierzymy się z koniecznością zapamiętania wszystkich ReGex’owych feature’ów, zmierzmy się z pierwszym przykładem przetwarzania tekstu przy użyciu ReGex.

Mam pewien tekst (swoją drogą, ciekawe, czy zgadniesz, jaki to utwór :) i chcę w nim znaleźć wszystkie wystąpienia liter tworzących ciąg liter **"nie"** (mogą to być zarówno oddzielne słowa jak i części innych wyrazów).



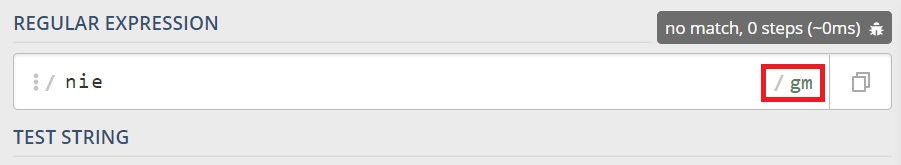
**Umieszczenie tekstu w pole test string**



**Wyszukanie wyrażenia 'nie'**

Jak widzisz w sekcji Test String, z podanego tekstu zostały wyłuskane części, których poszukujemy po zdefiniowaniu wyrażenia regularnego **'/nie/'**.

Sprawa banalna, zwróć jednak uwagę na zaznaczony czerwonym prostokątem obszar:

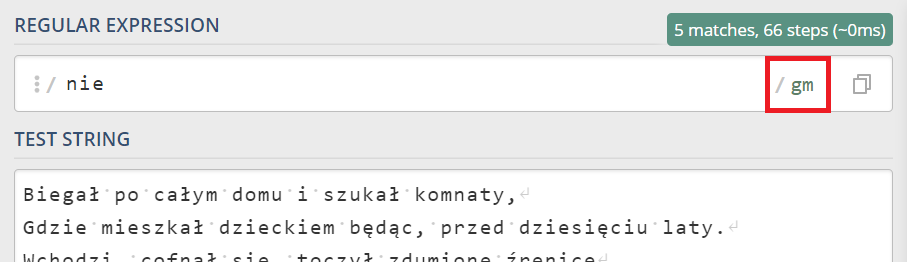


Tajemniczy ciąg **/gm** to nic **ReGex’owe flagi**.

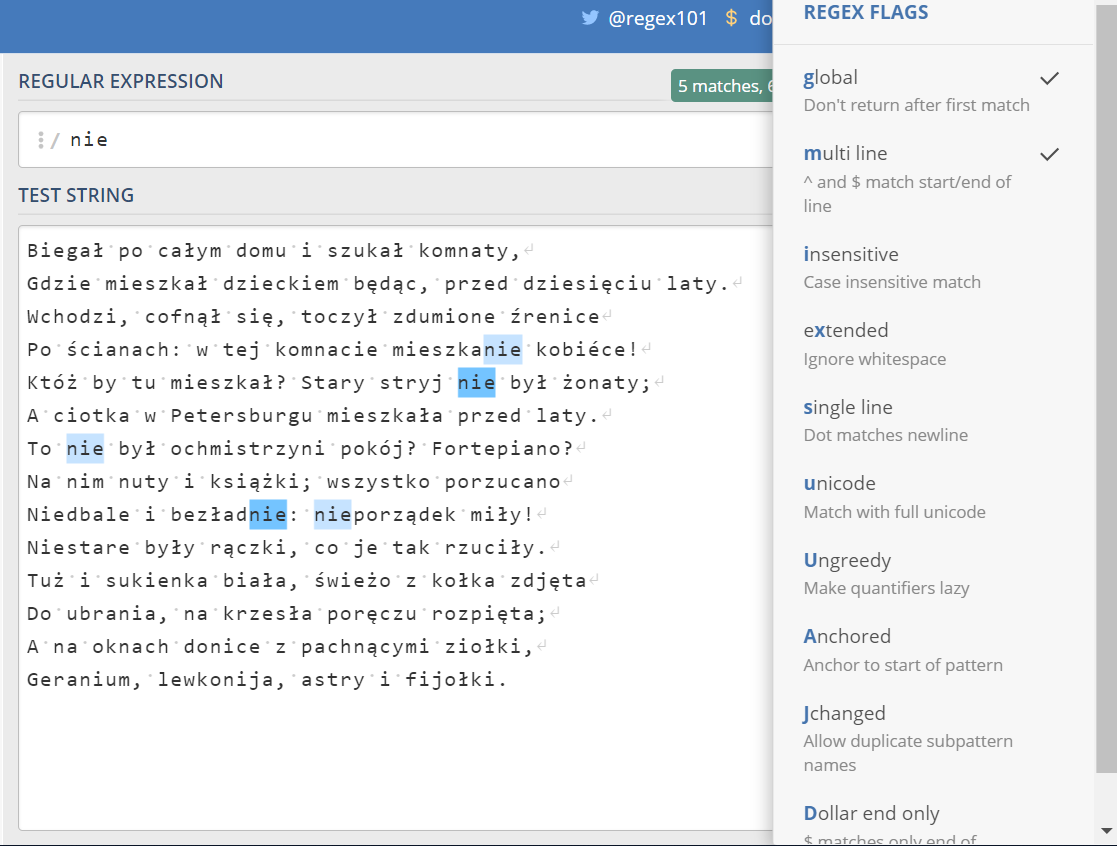
**Flagi** umożliwiają Nam wyszukiwanie danego tekstu, dodając dodatkowe warunki wyszukiwania jak, np. ignorowanie wielkości liter, wyszukiwanie tylko i wyłącznie pierwszego powtórzenia danego pattern'u, pomijanie znaków przejścia do nowej linii **etc.**

Przeanalizujmy, jakie flagi możemy umieścić w utworzonym uprzednio wyrażeniu.

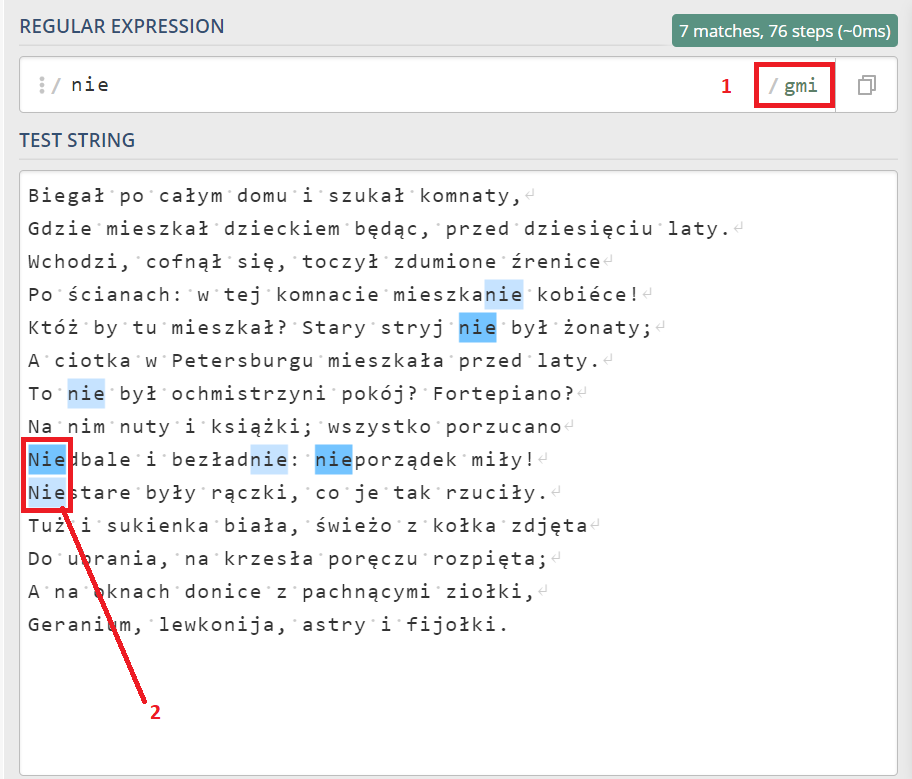
Rozwińmy możliwe do zastosowania opcje, klikając w niżej zaznaczony obszar:



Jak się okazuje, flagi, jakie mamy domyślnie ustawione przez silnik to **global (g)** oraz **multi line (m)**:



Chcąc dodać kolejną, możemy wybrać choćby opcję **i (insensitive)**, która spowoduje, że oprócz ciągu znaków w postaci **'nie'**, zostaną również wyszukane łańcuchy postaci: **'Nie'** (patrz niżej).



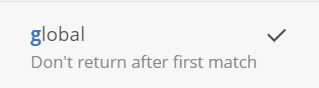
Zwróć szczególną uwagę na **1)** z powyższego screenshot'u.

Flagi dodawane są za znakiem **forward-slash: /** na koniec tworzonego wyrażenia regularnego.

Wiadome jest, że w tym momencie umieszczane są one automatycznie po interaktywnym wyborze flagi z listy. Aczkolwiek bądź świadomy tego, że choćby integrując już tworzony kod z wyrażeniami regularnymi, konieczne będzie ręczne ustawianie flag.

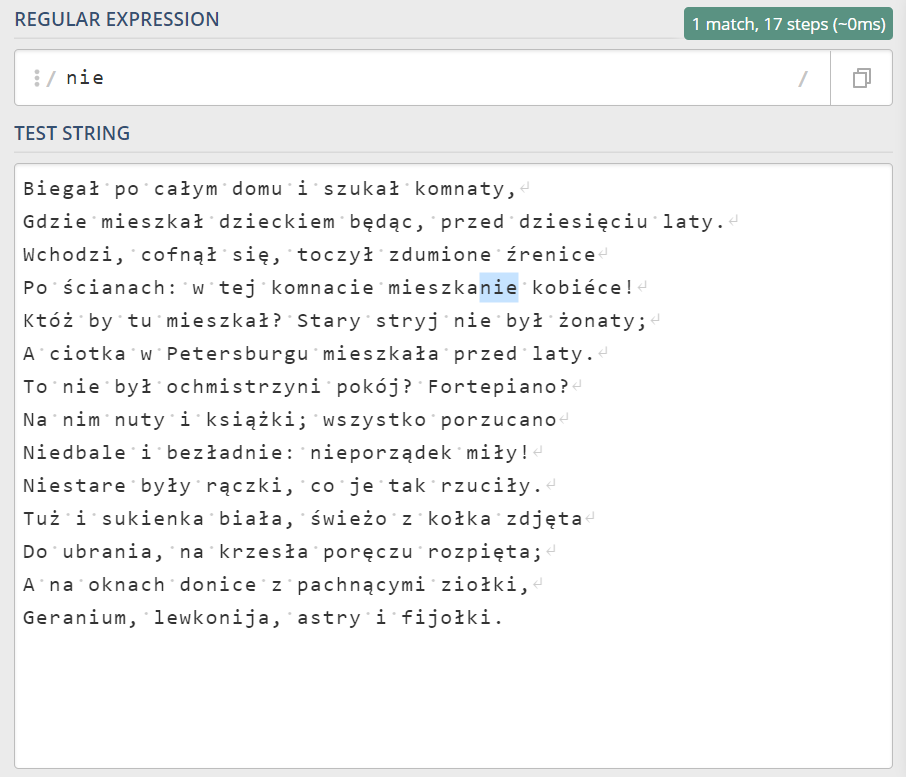
**Więcej o najpopularniejszych rodzajach flag**

1. **Flaga g - global**



Jak sam opis wskazuje, dodanie tej flagi do wyrażenia, umożliwia Nam wyszukanie w tekście **wszystkich** wystąpień danego pattern’u.

Gdybyśmy nie umieścili tej flagi, wówczas wyszukanie łańcucha 'nie', ograniczyłoby się tylko do wyszukania jego **pierwszego wystąpienia** w tekście.



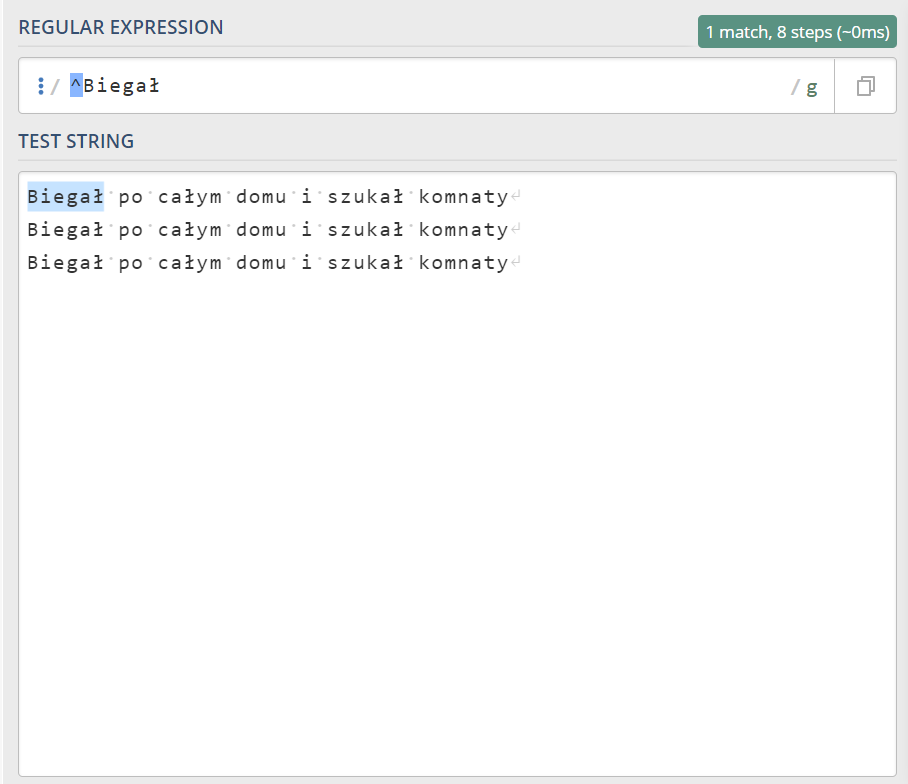
1. **Flaga m - multi line**

Zanim przystąpimy do analizy działania tej flagi, warto poruszyć temat **znaków specjalnych** w ReGex'ie.

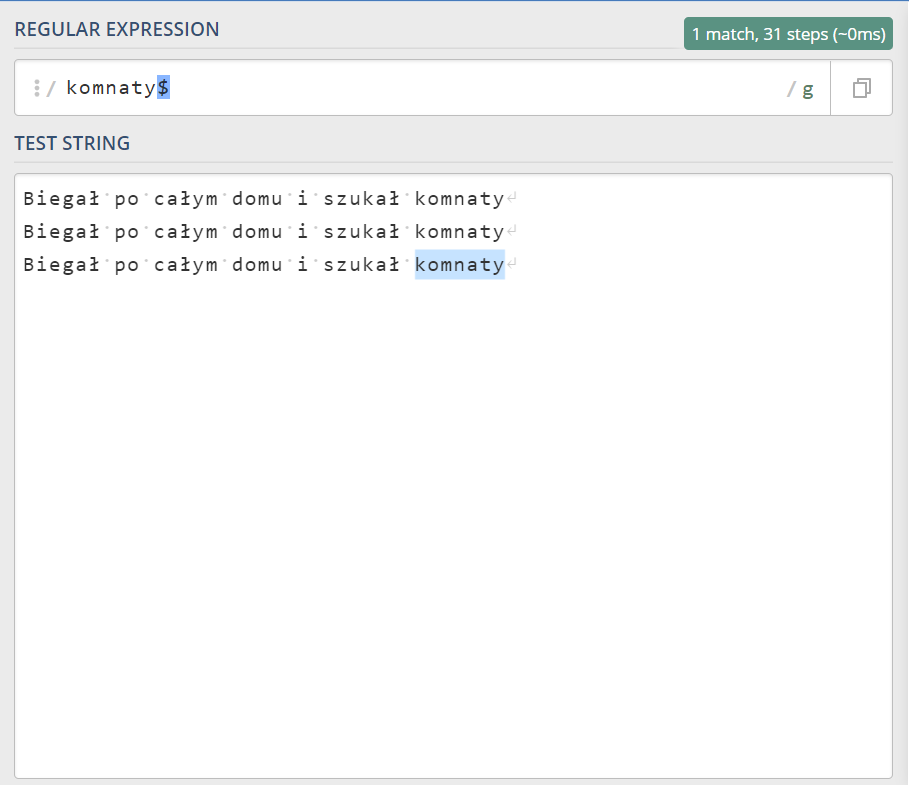
W tej sekcji zajmiemy się znakami **^** oraz **$**.

**Znak kareta (^)** umieszczany przed danym słowem/pattern'em, sprawdza czy dane wyrażenie znajduje się na początku tekstu.

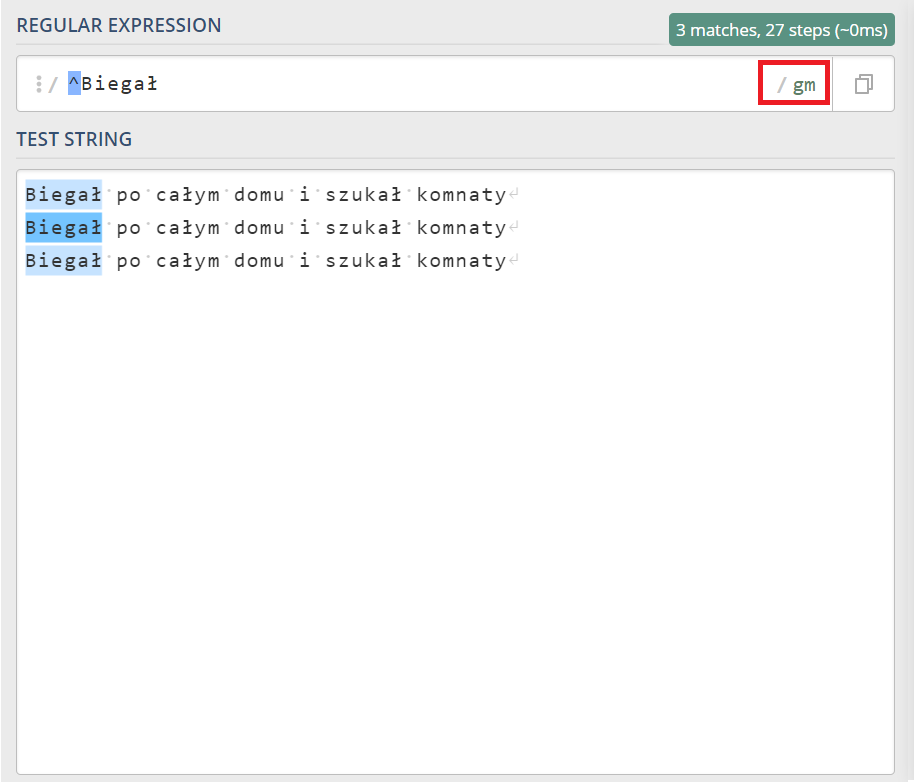
**Znak dolara ($)** umieszcza się go na końcu wyrażenia i umożliwia sprawdzenie, czy wyrażenie znajduje się na końcu tekstu.



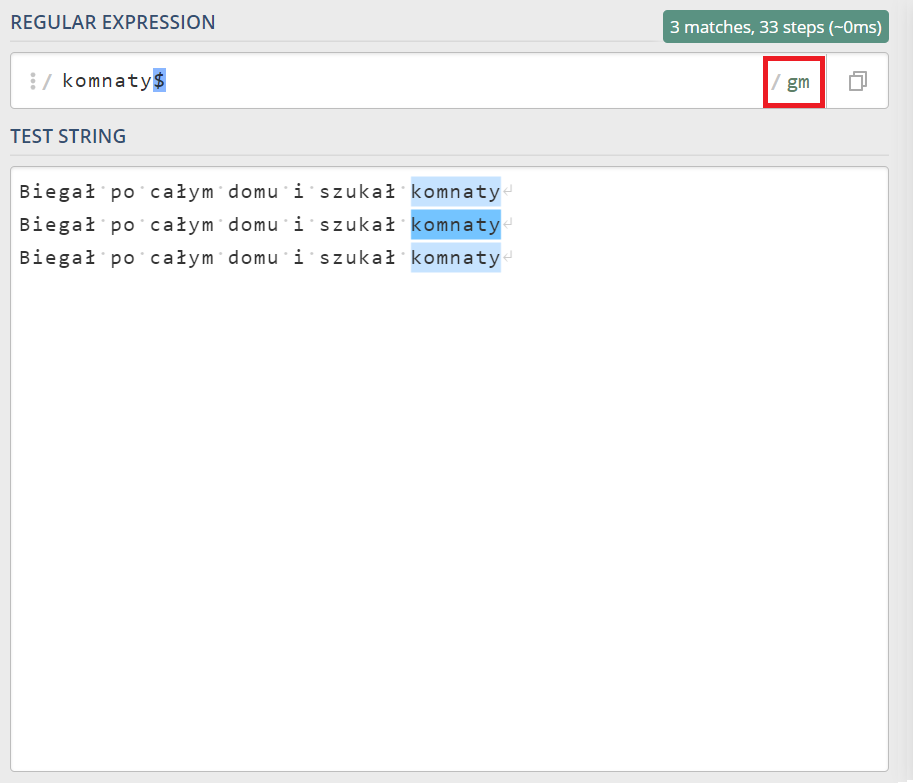
**Znak kareta bez flagi m**



**Znak dolara bez flagi m**



**Znak kareta z flagą m**

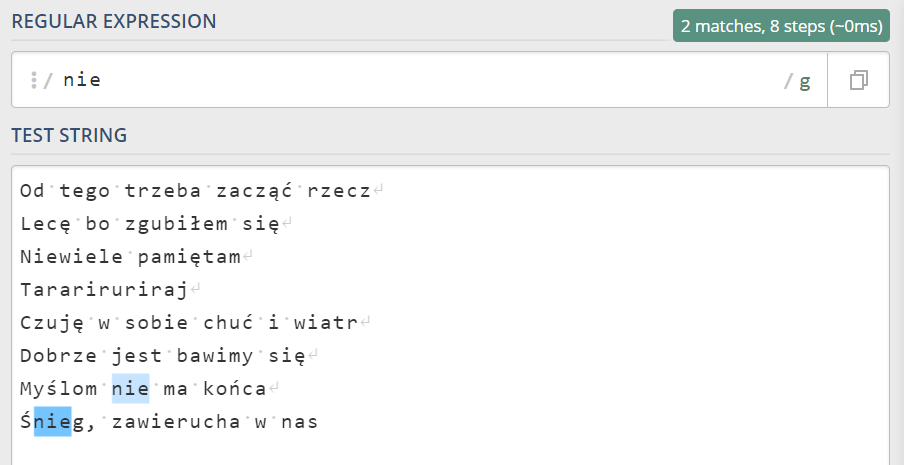


**Znak dolara z flagą m**

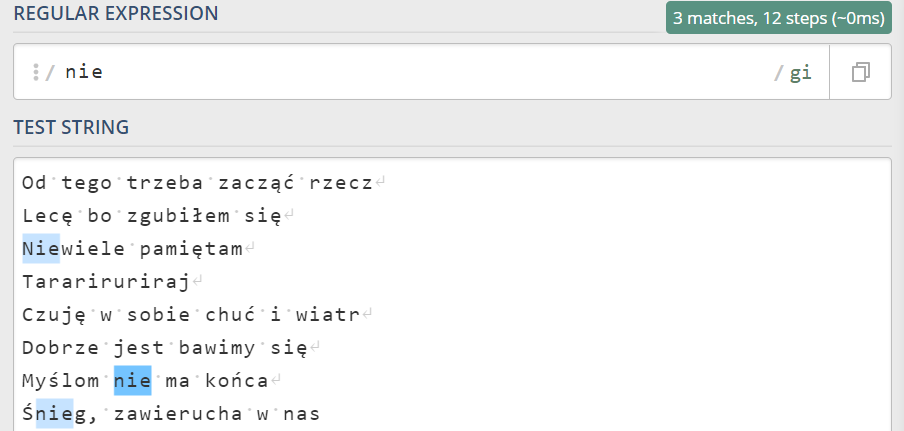
Jak możesz wywnioskować z powyższych przykładów, dodanie flagi m - multi line do dowolnego wyrażenia, umożliwia sprawdzenie, czy każda linia rozpoczyna się od danego wzoru tekstu.

1. **Flaga i - insensitive**

O tej fladze już wspominałem, także tylko przypomnę. Służy ona do ignorowania wielkości liter w danym pattern’ie. W tekście więc będą wyszukiwane wyrazy spełniające wyrażenie niezależnie od tego, czy pisane są wielką czy małą literą.



**Bez flagi i**



**Z flagą i**

1. **Flaga x - extended**

Załóżmy, że masz do zapisania długi pattern postaci **[b-chm-pP]at|ot** (na razie nieistotne, co oznacza).

W celu poprawienia jego czytelności można by przekształcić go do:

[b-chm-pP]

at | ot

**Wniosek:**

Bez ustawienia flagi x, białe znaki traktowane by były jako część patternu, a tak są ignorowane.

1. **Flaga u - unicode**

Pamiętasz użytą w wcześniejszych przykładach **klasę** \d?

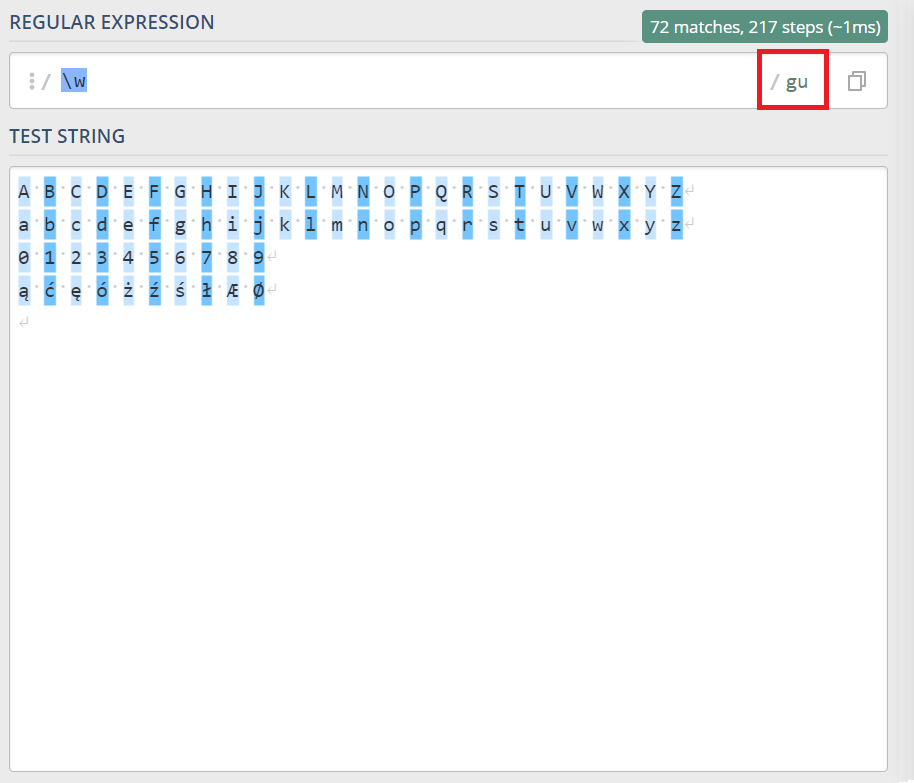
Klasy to na tyle obszerny temat, że skupimy się na nich w następnych sekcjach, jednak na ten moment powinieneś wiedzieć, że możemy wyróżnić **klasę dostępną pod tokenem \w**.

Określa ona zarówno **małe i duże znaki z przedziałów [a-z] oraz [A-Z] a także wszystkie cyfry [0-9].**

Problem jednak jest taki (patrz poniższy listing), że token **\w** nie obejmuje znaków z zakresu Unicode. Tak więc choćby polskie znaki narodowe nie zostaną wyszukane.



Rozwiązanie na to, aby zapewnić obsługę także znaków zakodowanych w Unicode, sprowadza się właśnie do dodania flagi **u**.



**Klasy znaków**

Proszę o werble! Rozpoczynamy kolejną mega praktyczną sekcję związaną z efektywnym tworzeniem wyrażeń regularnym.

**Klasy umożliwiają tworzenie o wiele krótszych wyrażeń regularnych,** Twór ten umożliwia "grupowanie/klasowanie" pewnych powtarzających się zależności - czyli, np**. zamiast wypisywać wszystkie możliwe do pojawienia się cyfry, zastosujemy token \d**.

**Wbudowane klasy znaków**

Powstały z racji częstego wykorzystywania tych samych grup znaków.

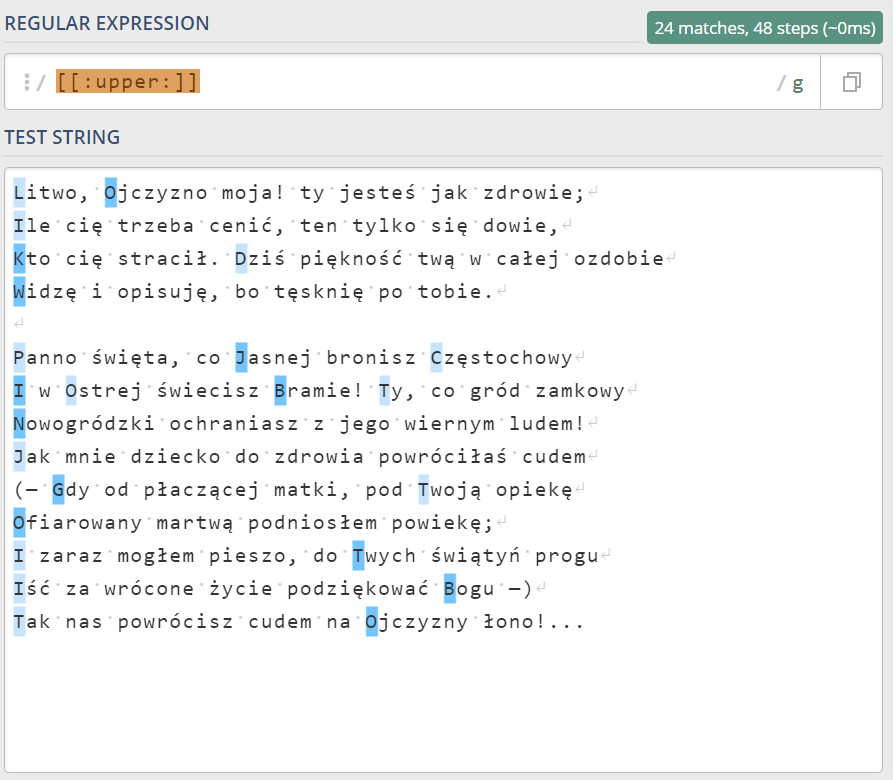
Do dyspozycji mamy:

* **\d** - dowolna cyfra
* **\D** - dowolny znak niebędący cyfrą
* **\s** - dowolny biały znak (np. spacja)
* **\S** - dowolny znak niebędący białym znakiem
* **\w** - dowolny znak należący do słowa (cyfry, litery i znak \_)
* **\W** - dowolny znak nie należący do słowa

Pozostałe klasy:

* **[[:digit:]]** dowolny znak będący cyfrą
* **[[:alpha:]]** dowolny znak będący literą
* **[[:alnum:]]** dowolny znak będący literą lub cyfrą
* **[[:lower:]]** dowolny znak będący małą literą
* **[[:upper:]]** dowolny znak będący dużą literą
* **[[:punct:]]** dowolny znak interpunkcyjny

**Przykładowe zastosowania:**



Klasa [[:upper:]]

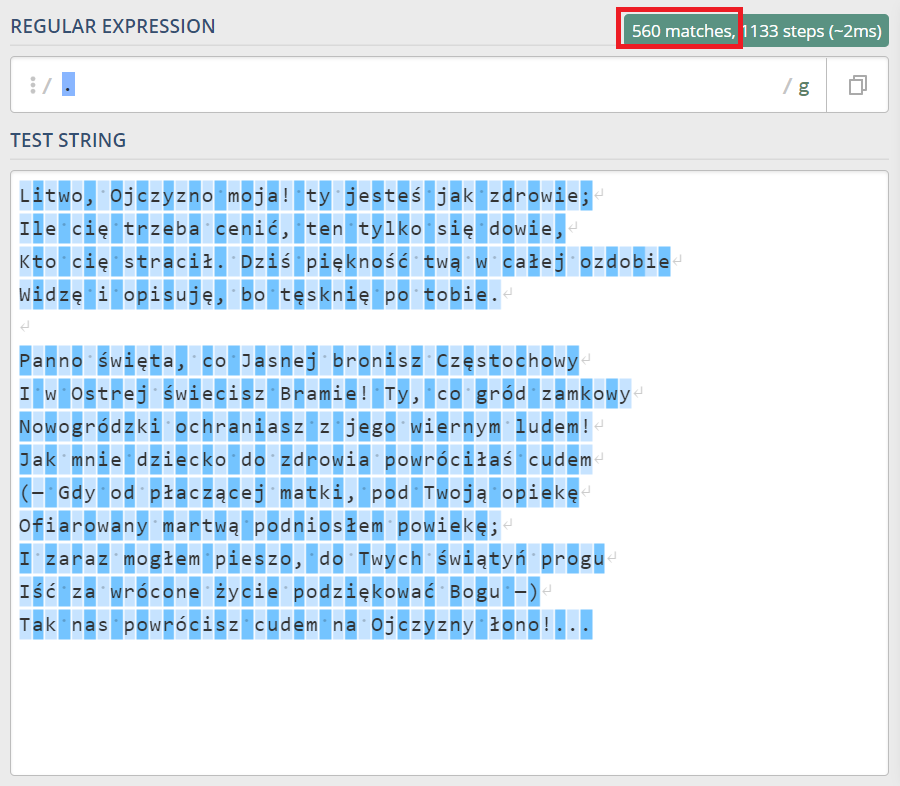


Klasa \W

**Kropka i jeszcze jedna flaga**

Silnik ReGex interpretuje kropkę jako dopasowanie do każdego znaku. Każdego! Czyli litery, cyfry, znaki diakrytyczne, spacje, ale... nie przejścia do nowej linii.

Spójrz:

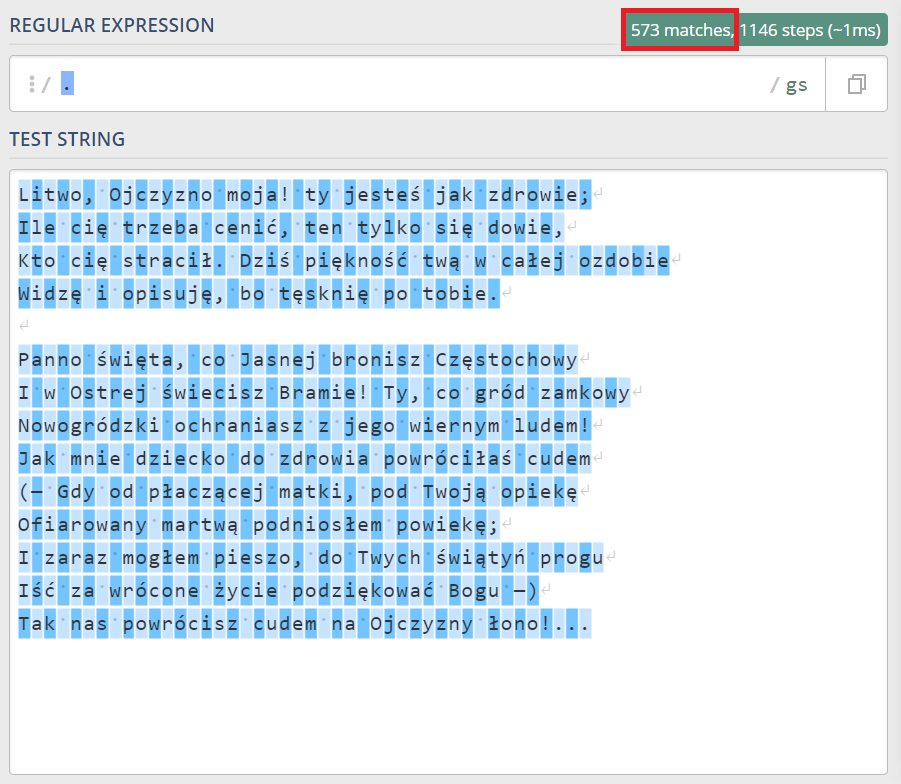


**560 dopasowań**

Mamy 560 dopasowań, czyli tyle, ile jest znaków (łącznie z spacjami) w tekście. Chcąc do grona rozpatrywanych znaków, zaliczyć również znak **'\n'** (przejście do nowej linii), należy zastosować flagę **s - single line**.

Celowo nie wspomniałem o niej w sekcji Flagi, aby móc teraz przywołać ją wraz praktycznym przykładem.

Flaga s umożliwia więc dopasowywanie znaków przejścia do nowej linii do odpowiednio stworzonego pattern’u (cały tekst jest traktowany tak, jakby był jednolinijkowy).



**573 dopasowań z flagą s**

**Zbiory**

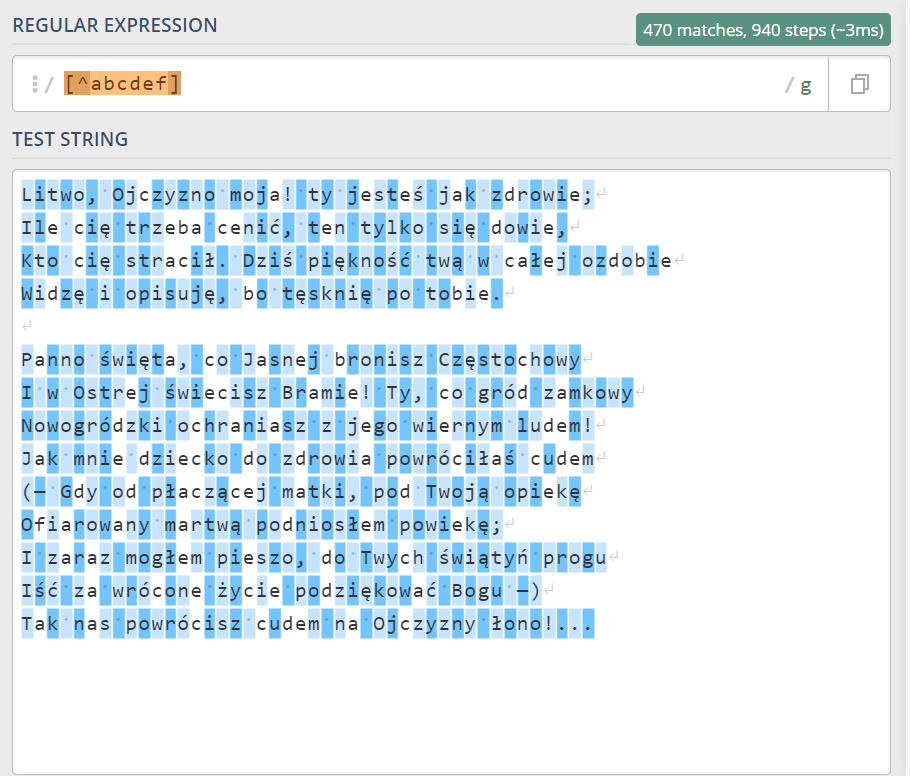
Chcąc przeszukać tekst w celu odnalezienia wystąpień określonych pojedynczych znaków, **niekoniecznie tworzących określony łańcuch**, wykorzystamy zbiór.

Zasada jego tworzenia jest prosta - wykorzystujemy **nawiasy kwadratowe [],** a między nimi umieszczamy te znaki, które chcemy znaleźć w tekście.



Ale to nie wszystko! Z tekstu możemy również wykluczać określone zbiory znaków przy użyciu znaku karety ^ (z którym de facto zapoznaliśmy się już wcześniej, ale używany był w innym kontekście).

**Chcąc znaleźć zatem wszystkie pozostałe znaki, które nie zaliczają się do zbioru [abcdef], zapiszemy [^abcdef].**

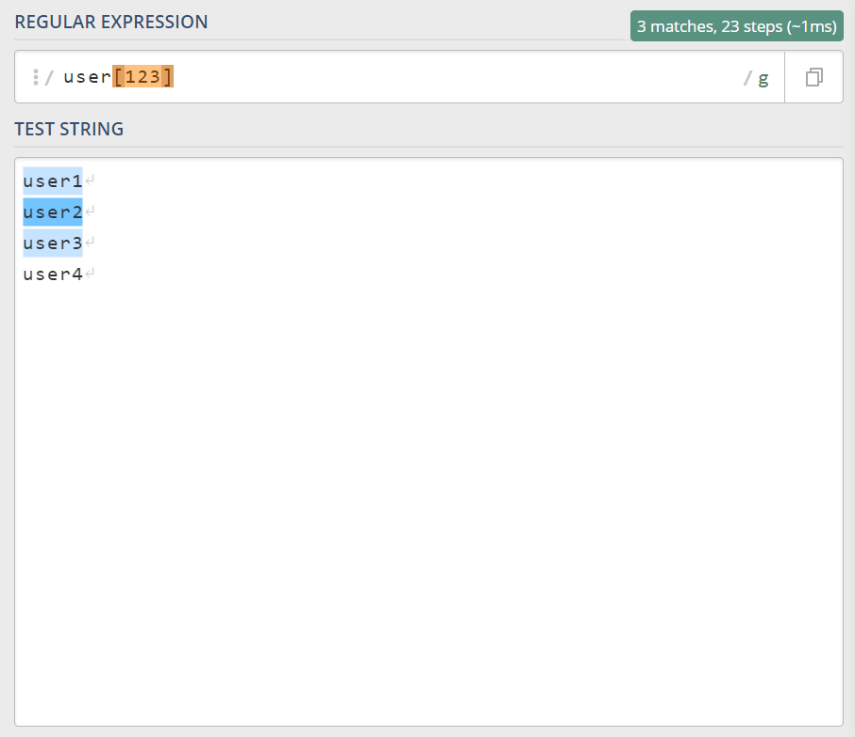


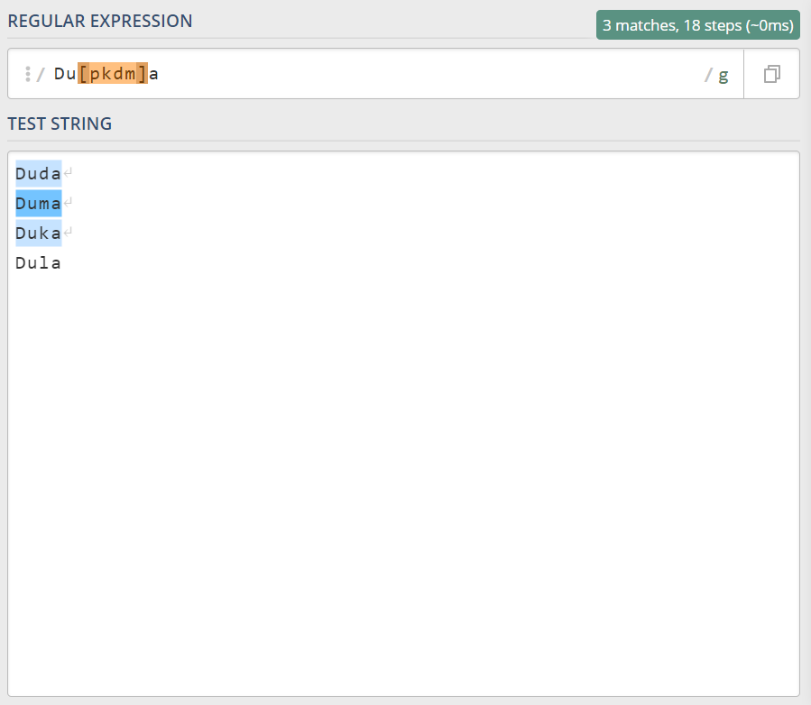
Zbiory umożliwiają efektywną analizę tekstu i rozpatrywanie wielu kombinacji. Warto jednak już teraz sobie uświadomić i zapamiętać, że elementy w zbiorze odnoszą się do pojedynczych liter.

**UWAGA:**

Zapisując zbiór w postaci [ala] wcale nie chcemy w tekście poszukiwać wszystkich wystąpień słowa ‘ala’, ale pojedynczych liter: ‘a’, ‘l’, ‘a’.

Kilka praktycznych przykładów.





**:)**

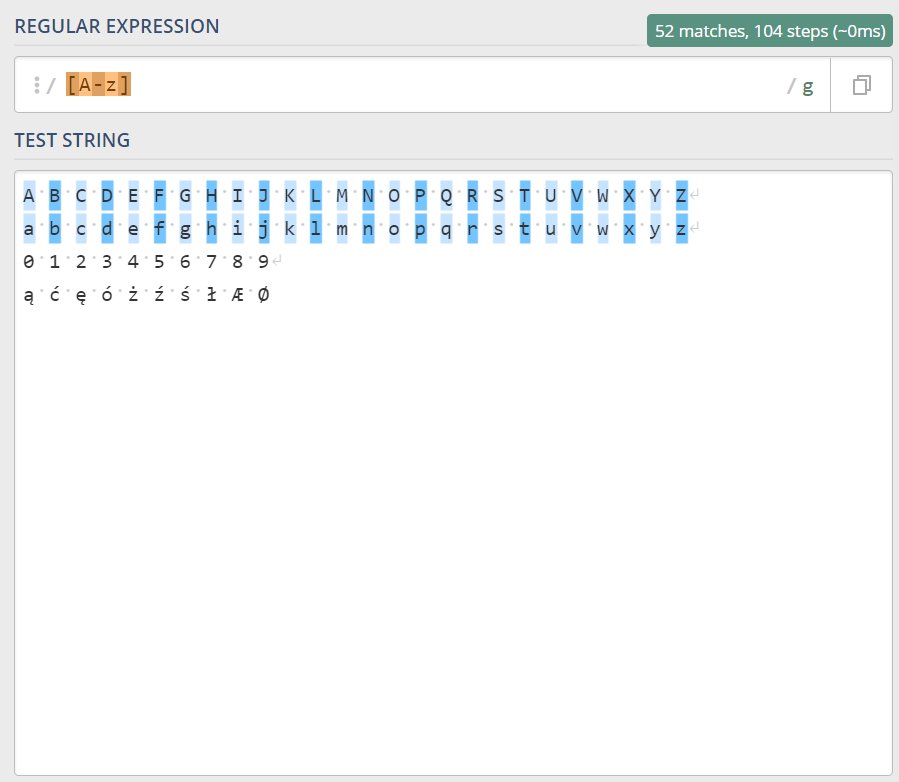
**Zakresy**

Zakresy są mocno związane z omówionymi klasami (\d, \w, \W itd).

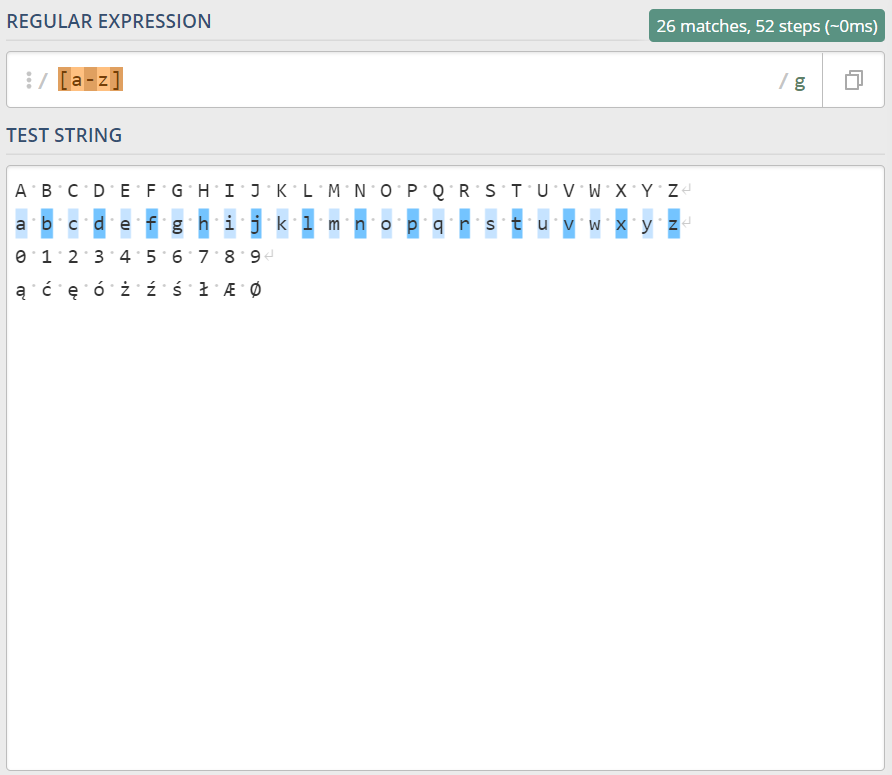
I tak grupę [0123456789] możemy zastąpić zakresem **[0-9],** cały alfabet łaciński   
**[A-z].**

Wynika to z tego, że każdy znak ma przypisany kod ASCII (odpowiednik liczbowy) i tak na przykład litery a, b, c, d, ..., z mają przypisane wartości od 97 do 122 (jeżeli nie pamiętasz, czym są kody ASCII, wróć do Szkolenia 1 + 2).

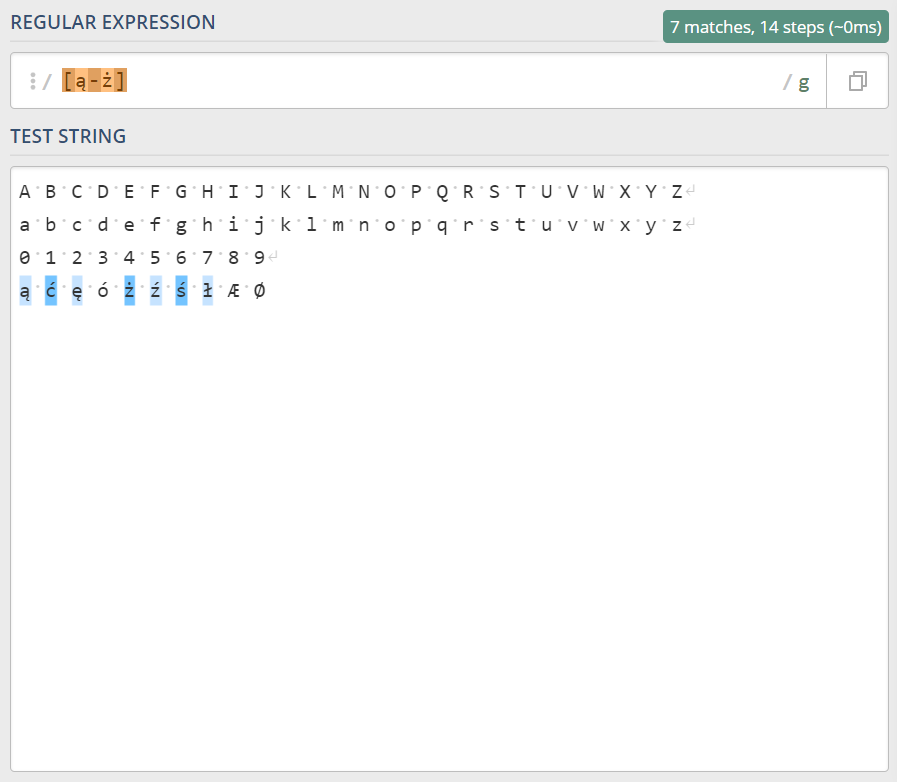
Niestety z racji, że polskie znaki diakrytyczne nie mają "odpowiedników liczbowych" ustawionych w określonej kolejności, to zapis [ą-ż] n**ie pobrałby wszystkich oczekiwanych znaków z przedziału** (w gwoli ścisłości, zostałaby pominięta litera ó).



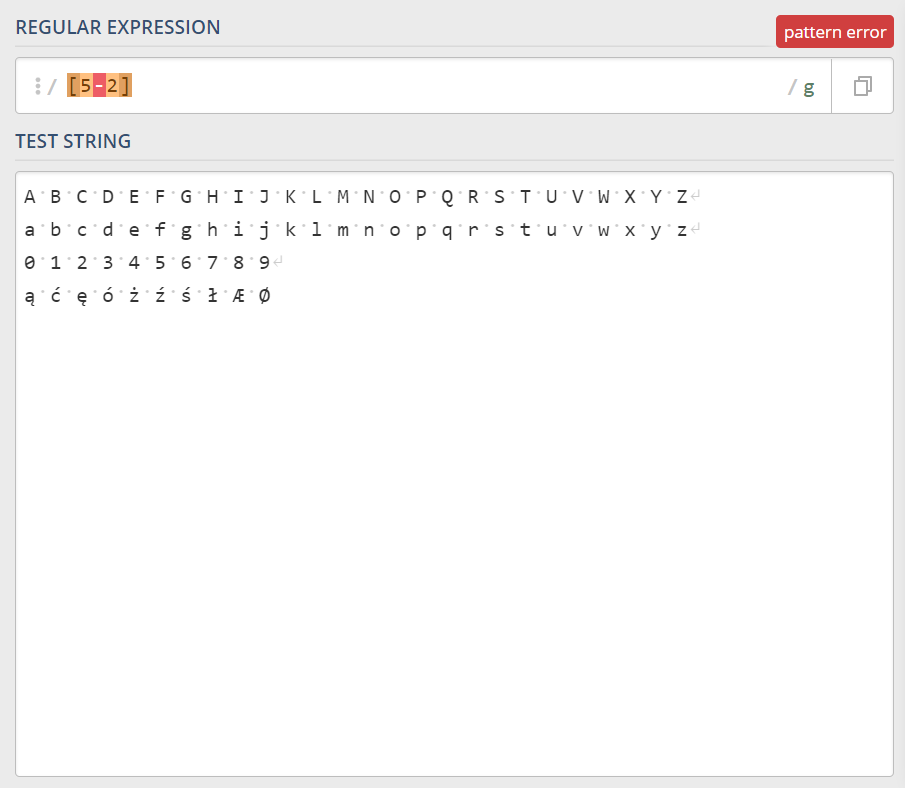
**Alfabet łaciński**



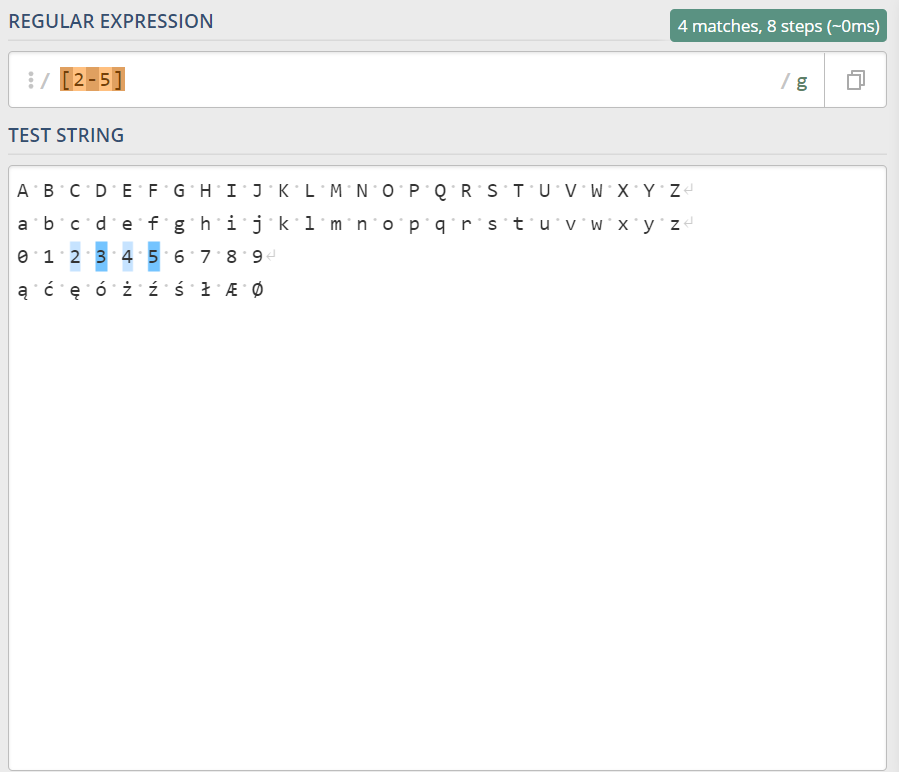
**Małe litery alfabety**



**Nieoczekiwany rezultat!**



**Pamiętaj o definiowaniu poprawnych zakresów!**

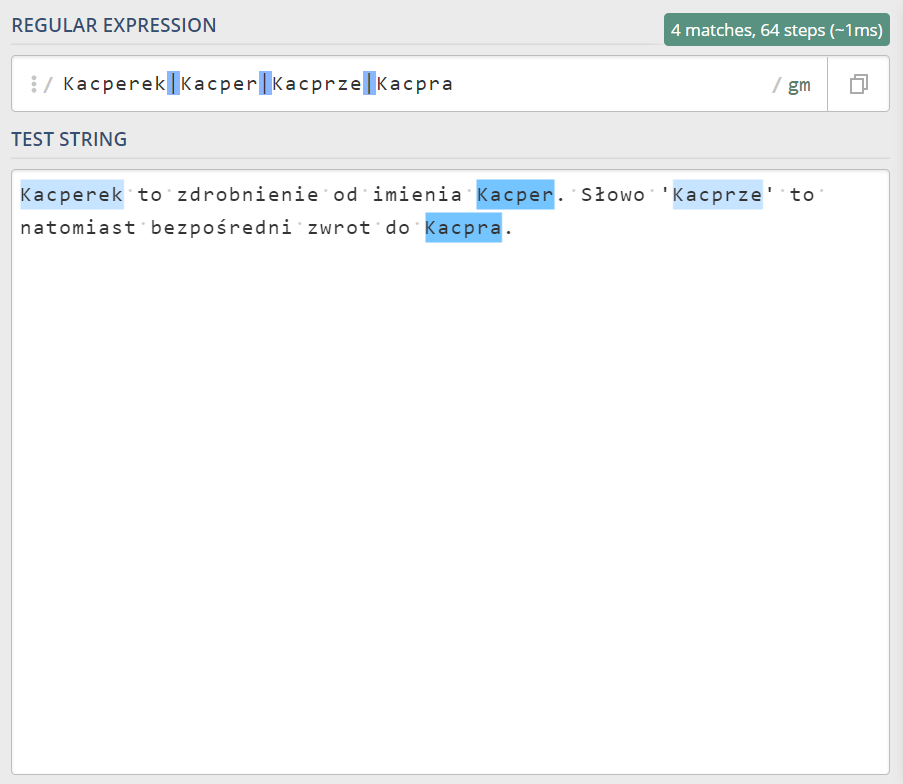


**Cyfry z zbioru [2;5]**

**Sprawdzanie kilku warunków równocześnie (alternatywa)**

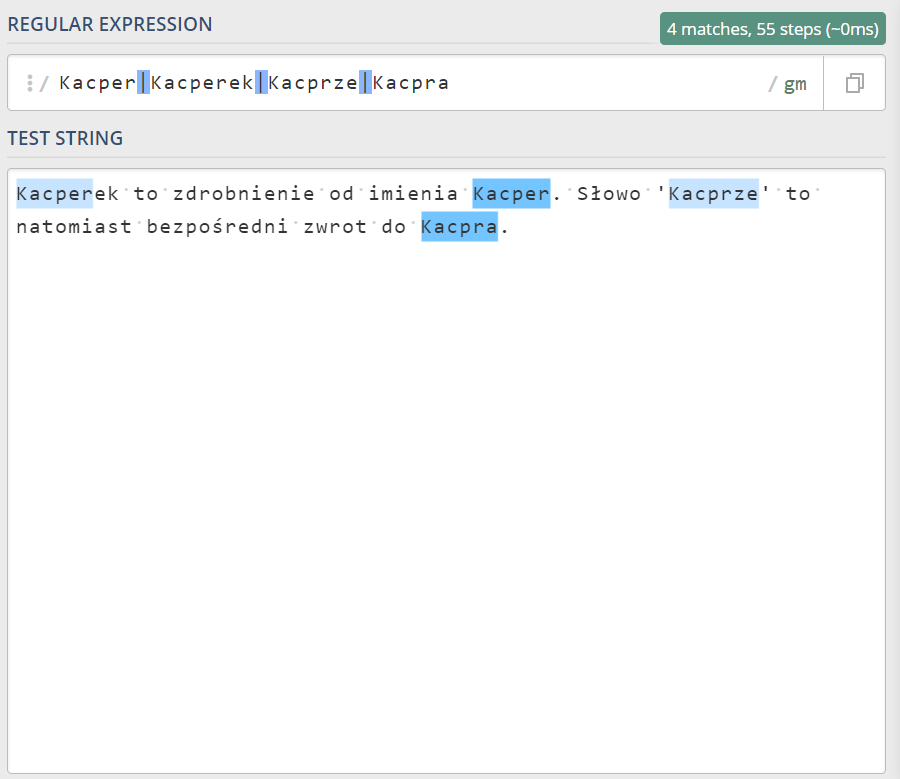
Do tej pory, jeżeli chcieliśmy znaleźć różne wzory w danym tekście, konieczne było użycie różnych niezależnych wyrażeń regularnych.

Od tej pory możemy łączyć dane wzory za pomocą znaku **alternatywy |.**



**UWAGA:**

Zwróć uwagę, że gdybyś zmienił kolejność alternatywy w ReGexie na choćby:   
**Kacper|Kacperek|Kacprze|Kacpra**, to otrzymałbyś następujący efekt:



Różnica dość subtelna, ale zwróć uwagę na dopasowanie słowa Kacperek a Kacperek.

Z racji, że początek alternatywy **Kacper|**Kacperek pasuje do wyrazu **Kacper**ek, to silnik ReGexa rozpatrzy tylko i wyłącznie wyrazy spełniające pierwszą część alternatywy.

**Powtórzenia**

Kolejnym sposobem na ułatwienie sobie życia z ReGex'ami są powtórzenia. Pozwalają one skracać zapis wyrażenia oraz **określać, ile razy ma zostać powielony** dany **znak/grupa/klasa** znaków.

Wróćmy do przykładu z początku tego szkolenia, który odnosił się do zapisu kodu pocztowego w formie wyrażenia regularnego.

Zapis ten można przedstawić jako:

\d\d-\d\d\d lub uprościć na \d{2}-\d{3}



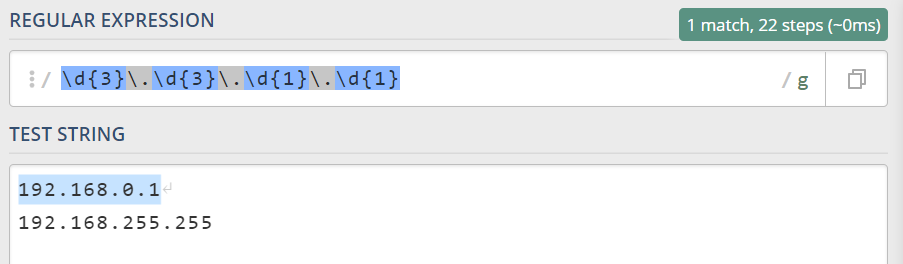
To właśnie wartości {2} oraz {3} określają, ile razy ma się coś powtórzyć w tworzonym wyrażeniu.

**Jak więc możesz zobaczyć, powtórzenia definiujemy umieszczając liczbę N oczekiwanych powtórzeń między klamrami: {N}.**

**Zakres powtórzeń**

Czas na kolejny przykład! Załóżmy, że chcesz zapisać swój adres IP. Najpopularniejszy przykładem z zakresu adresów prywatnych jest chyba postać: 192.168.0.1

Chcąc go wyszukać, zapiszemy:

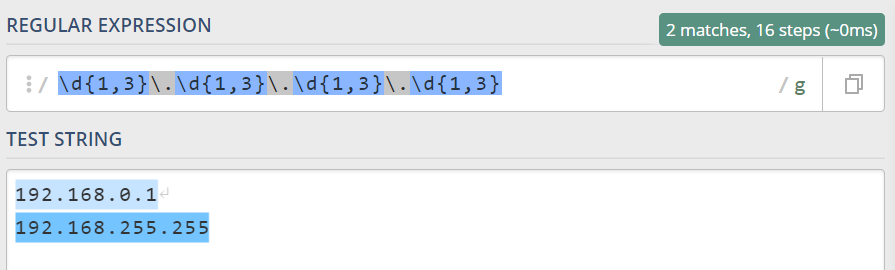


**Pamiętaj o tym, aby przed kropką umieścić backslash**( **\** umieszczany przed znakami specjalnymi umożliwia traktowanie ich jako normalne znaki)

No dobrze, ale powiesz teraz - **"A co w momencie, gdyby adres na ostatnich dwóch oktetach miał wartość nie jednocyfrową, ale na przykład dwu- lub trzy- cyfrową?**”

W takim przypadku skorzystamy z zakresu powtórzeń!

I tak, chcąc poprawić stworzony ReGex, aby sprawdzał zarówno jedno-, dwu- oraz trójznakowe wartości, należy zapisać:



Zapis **\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}** mówi więc, aby poszukiwać cyfr \d, które występują w zakresie od d do 3 powtórzeń włącznie.

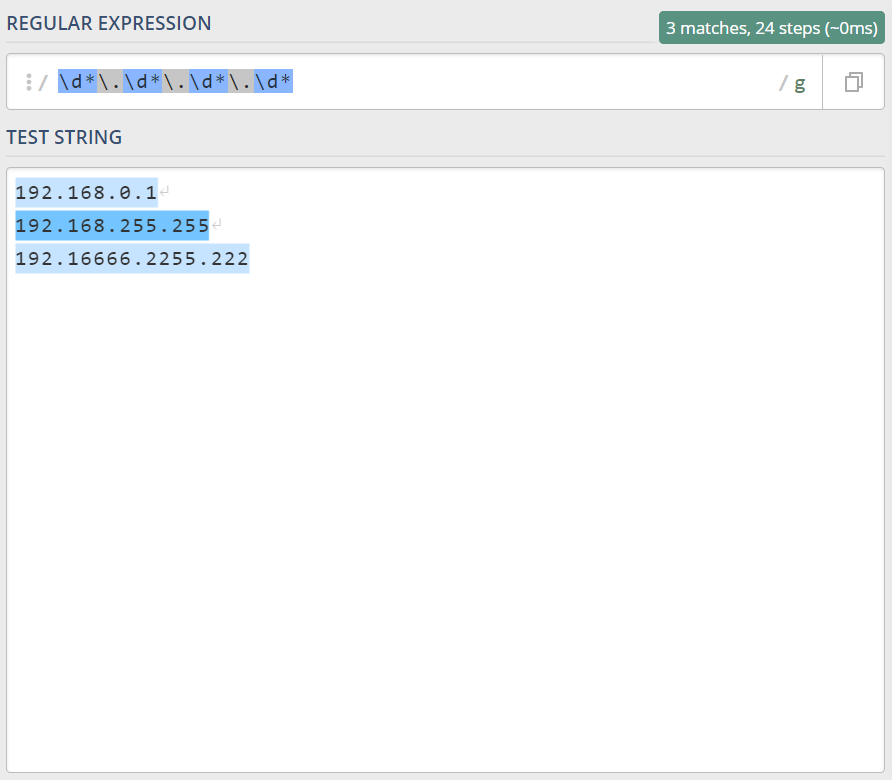
I tak właśnie definiujemy zakresy powtórzeń.

**\d{2, 3}** – szukaj ciągu cyfr, które występują 2 lub 3 razy

**\d{1, 10}** – szukaj ciągu cyfr, które występują 1 lub 2 lub 3 lub … lub 10 razy

**Skrótowce:**

* **{1,}** - 1 lub więcej powtórzeń
* **{0,}** - brak lub dowolna ilość powtórzeń
* **{0,1}** - czyli wystąpienie opcjonalne, możemy zapisać jako **?**
* **{0,}** - zero lub więcej powtórzeń, możemy zastąpić jako **\***
* **{1,}** - jedno lub więcej powtórzeń, możemy zapisać jako **+**



**Przykład z zastosowaniem \d\***

**Zachłanność**

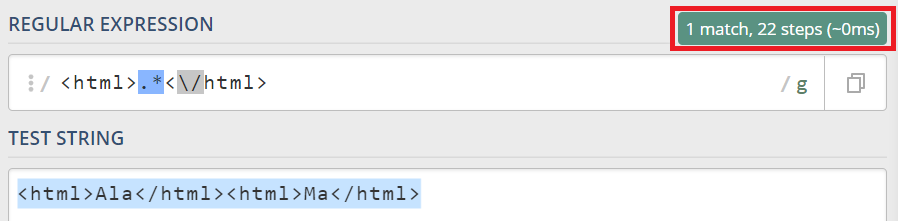
Temat zachłanności dotyczy tak naprawdę specyfiki działania silnika. Warto zaznajomić się z tym pojęciem, aby w pełni rozumieć zasadę działania silnika i w trakcie tworzenia wyrażeń, nie natrafić na nieoczekiwane przez Nas zachowania.

**Problem:**

Jak zachowa się silnik dla pewnego tekstu i jego wyrażenia? (kod HTML)

**ReGex:**

<html>.\*</html> (to samo co <html>.{0,}</html>)



**????**

Zwróć uwagę na **zaznaczony obszar (czerwony prostokąt).**

Jeżeli nie rozumiesz jeszcze dlaczego taki powinien być wynik, zauważ, że ReGex <html>.\*<\/html> mówi Nam**: znajdź wszystkie dopasowania, które zaczynają się od znacznika <html>, mają dowolną ilość liter pomiędzy (.\*) oraz kończą się znacznikiem </html>**

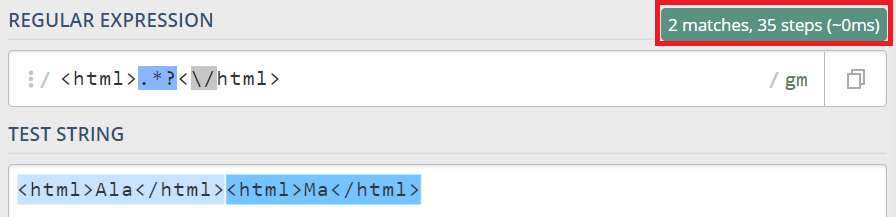
W rezultacie widzimy tylko **1 dopasowanie**! A powinny być przecież 2 (dla **<html>Ala</html>** oraz **<html>Ma</html>**).

Wynika to właśnie z **zachłannego działania silnika ReGex**.

Dopasowuje on bowiem kolejne znaki do .\* **(łącznie z </html> i wszystkimi pozostałymi)**.

A więc zachłanne przeszukiwanie kończymy dopiero w momencie, gdy dany tekst się zakończy (a powinniśmy kończyć na napotkaniu znacznika </html>).

To, co powinniśmy zmienić w wyżej przedstawionym wyrażeniu, aby umożliwić wyszukiwanie zgodnie z oczekiwanym rezultatem, to modyfikacja ReGex **<html>.\*</html**> na **<html>.\*?</html>** (przypominam, że ? odnosi się do powtórzeń i oznacza wystąpienie opcjonalne - 0 razy lub 1).



**Grupy**

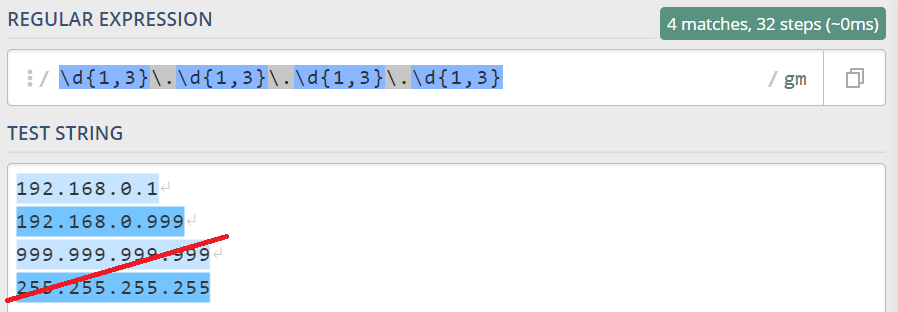
Przyznaję otwarcie, że grupy to chyba najpotężniejszczy i naczęściej wykorzystywany twór przy tworzeniu i analizowaniu wyrażeń ReGex.

Grupy umożliwiają Nam odwoływanie się do określonych części ReGexu. Dzięki temu może choćby walidować przedziały wyszukanych wartości liczbowych.

**Pamiętasz przykład z adresami IP?**

Badaliśmy tam tylko i wyłącznie **ilość wystąpień cyfr w oktetach**. Wspomnieliśmy jednak, że w uproszczeniu, jeden oktet może osiągać wartość **od 0 do 255**.

Mówię o takiej sytuacji:



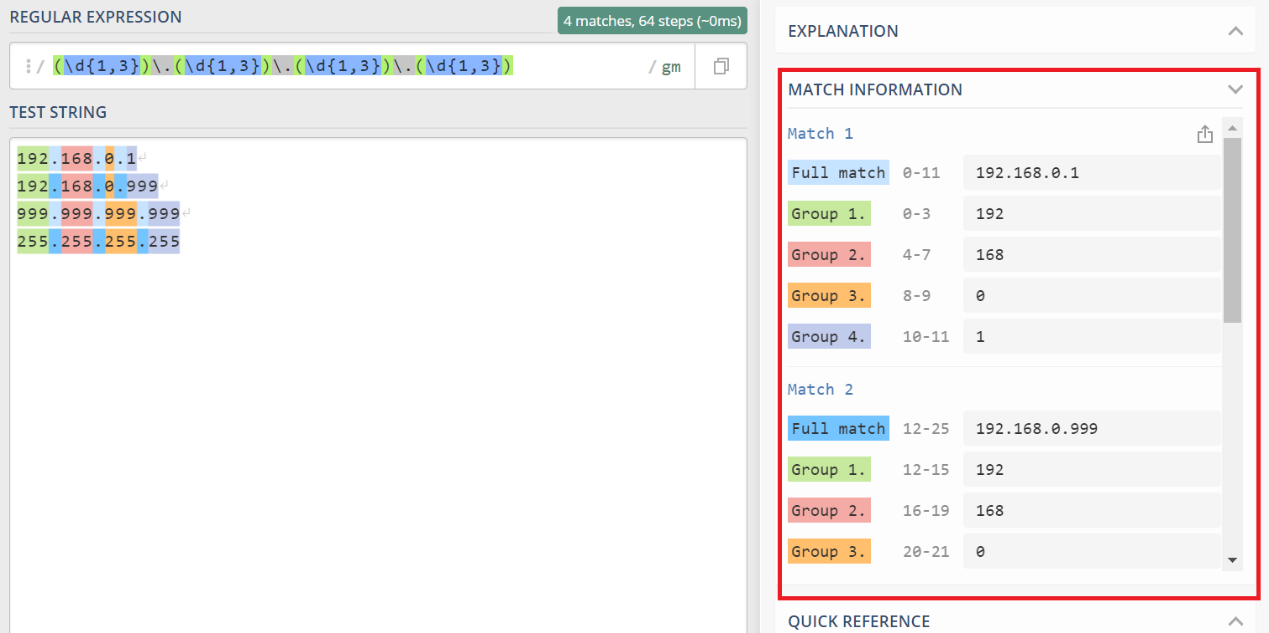
Silnik ReGexa nie jest w stanie walidować zakresu wartości całych liczb, jednak możemy go do tego zmusić, tworząc grupy i odwołując się do nich **z poziomu języka programowania**.

Tak więc moglibyśmy każdy z czterech oktetów umieścić w osobnych grupach. Po wyszukaniu dopasowań, zaprogramowalibyśmy metodę sprawdzającą, czy wyłuskany z tekstu substring, przechowuje liczby z wyznaczonego przedziału.

**Grupy przechwytujące**

Tworzymy je przy użyciu **nawiasów okrągłych ().**

Wcześniej przytoczony przykład możemy więc zmienić na:



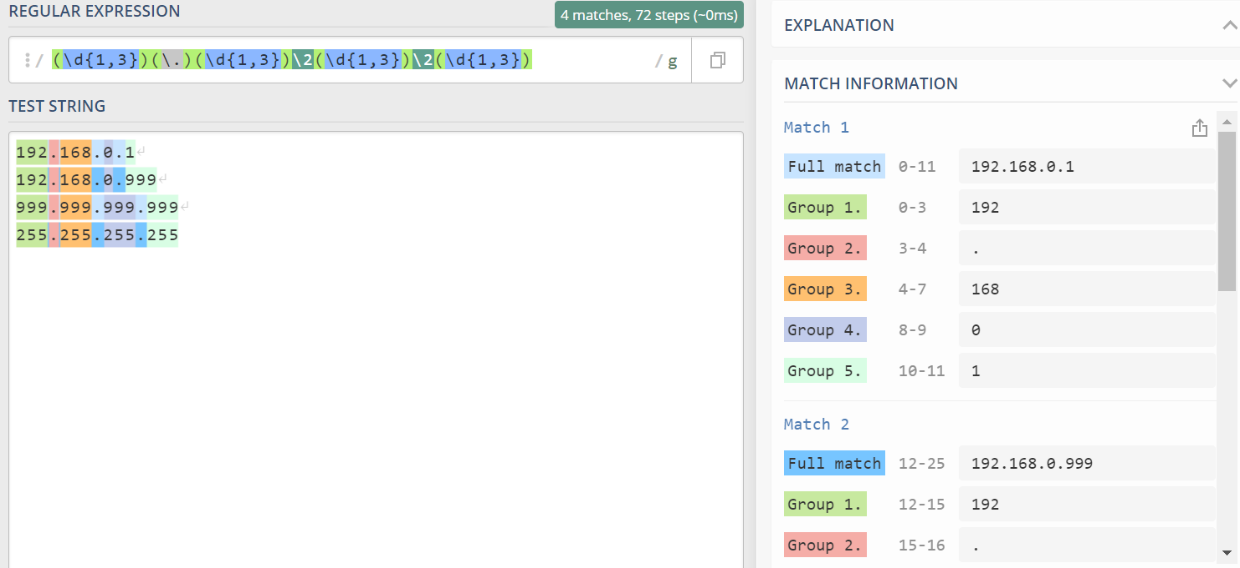
**Dodanie grupy przechwytujących**

Zauważ, że po wprowadzeniu grup, zaczęliśmy wykorzystywać **sekcję Match Information**. Widzimy, jak ładnie znalezione sekcje zostały podzielone na poszczególne grupy (Group 1., Group 2. Group 3., Group 4.).

Do tak wydzielonych grup możemy się wielokrotnie odwoływać za pomocą \N, gdzie N to nr grupy.

I tak chcąc sprawdzić grupę nr 1 z powyższego przykładu, czyli **tekst 192**, zapiszemy **\1**.

Myślę, że stosowanie takiej numerologii w odwoływaniu się do grup dobrze zaprezentuje poniższy przykład:

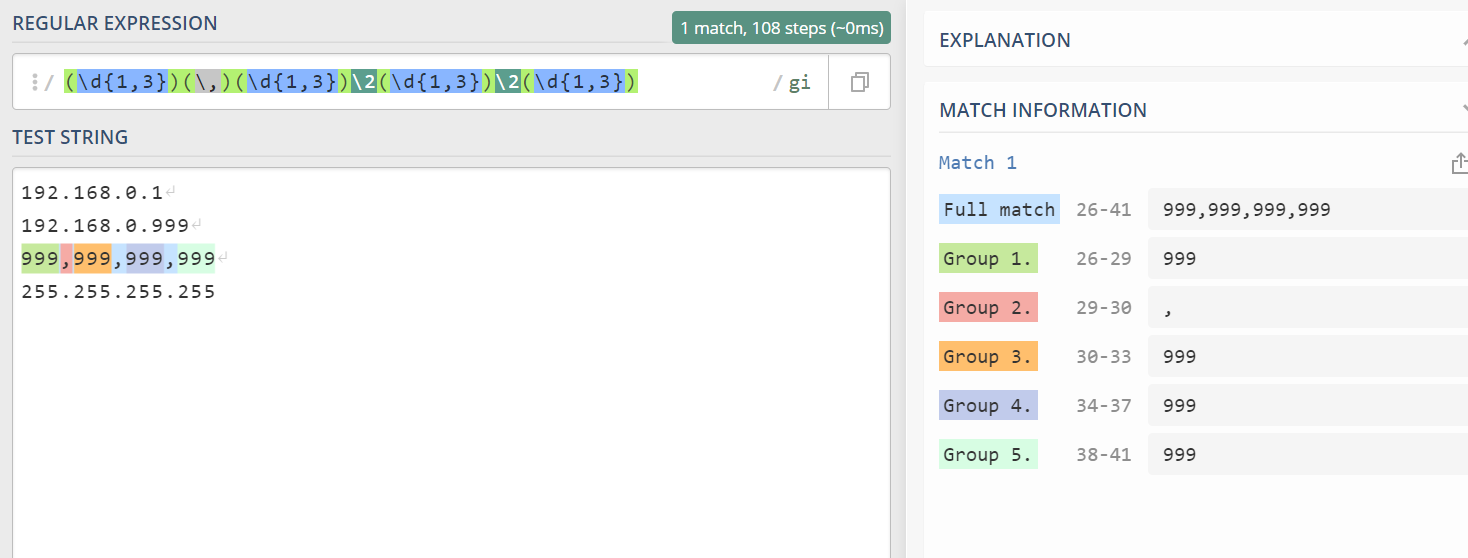


Znak kropki umieściliśmy w oddzielnej grupie **(Group 2)** i odwołujemy się do niej, wykorzystując zapis **\2**.

(\d{1,3})**\2**(\d{1,3})**\2**

**Praktyczne zastosowanie**

Wykorzystanie grup przechwytujących może się sprawadzić, gdy chcemy choćby zmienić separator występujący między liczbami na, na przykład przecinek. Wówczas wystarczy dokonać modyfikacji w jednym miejscu wyrażenia, zamiast wszędzie zmieniać znak kropki na przecinek.



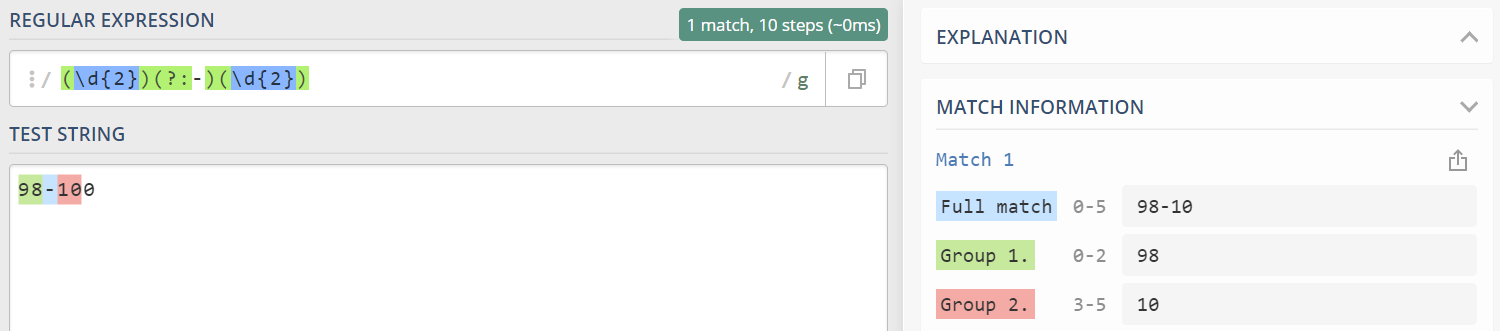
**Grupy nieprzechwytujące**

Różnica między grupami przechwytującymi a nieprzechwytującymi jest taka, że do tych drugich są przypisane żadne numery.

Grupy te bowiem mogą służyć do wielu innych celów niż grupy przechwytujące. Przede wszystkim umożliwiają one wydzielić pewne części w kodzie tylko po to, by łatwiej odczytywać i interpretować wyrażenie.

Na nasze potrzeby, grupy nieprzechwytujące, możemy nazwać potocznie grupami „niedolarowanymi”. Dlaczego? Czytaj dalej!

Aby utworzyć grupę nieprzechwytująca, należy bowiem oprócz nawiasów okrągłych, skorzystać z zapisu **?:**. Między nawiasami umieszczamy natomiast wyrażenie, według którego mamy grupować fragmenty tekstu.



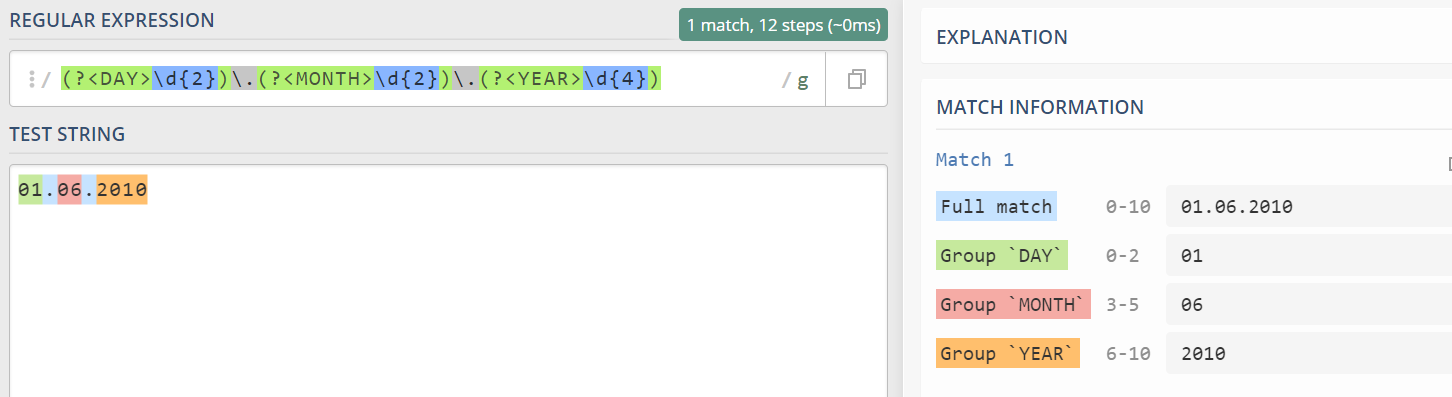
Przykład z dwiema grupami przechwytującymi i jedną nieprzechwytującą – (?:-)

**Grupy nazwane**

I na koniec **grupy nazwane**... Grupy nazwane różnią się od grup przechwytujących tylko tym, że identyfikujemy je nie numerem, ale **nazwą**.

Zapis jest zbliżony do wcześniejszych przypadków: grupę otwierasz przy użyciu (, następnie znak zapytania oraz między znakami mniejszości i większości podajemy nazwę grupy, np. **(<YEAR>\d{4}).**

Użycie grup nazwanych poprawia rozeznania programistów w tym, jakie rodzaje grupowań zostały zastosowane w obrębie wyrażenia.



Porównaj następujące zapisy:

* **(?<DAY>\d{1, 2}) vs \d{2}**
* **(?<YEAR>\d{4}) vs \d{4}**
* **(?<HOUR>\d{2}) vs \d{2}**
* **(?<MONTH>\d{2}) vs (?:\d{2})**

itd.

Przyznaj otwarcie, że grupy nazwane od razu informują Nas, do czego chcemy się odwołać i dzięki temu umożliwiamy innym programistom czytającym Nasz kod, zwiększyć efektywność i szybkość pracy.