



# Python



## Poziom średniozaawansowany

### Kamil Nowak

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy

# Plan na dziś



- Struktury danych:
  - Kopiec
- Refleksje
- Klasy i metody abstrakcyjne
- Wyrażenia regularne



# Struktury danych - kopiec

- Ostatnią strukturą, którą omówimy jest kopiec.
- Kopiec jest drzewem binarnym o specjalnej właściwości - w korzeniu (na szczycie kopca) zawsze znajduje się najmniejsza wartość jeśli kopiec jest minimalny lub największa jeśli kopiec jest maksymalny.
- Python dostarcza implementacji minimalnego kopca w module **heapq**.
- Pomimo, że kopiec formalnie jest drzewem, w praktyce przechowuje się go w postaci listy.
- Kopiec słabo nadaje się do wyszukiwania ale świetnie sprawdza się do tego aby pobrać z jego wierzchołka najmniejszy element. Wtedy na to miejsce wskoczy następny w kolejności najmniejszy element.
- W takim razie z kopca pobieramy elementy uporządkowane rosnąco - tak właśnie działa sortowanie przez kopcowanie - z listy losowych elementów tworzymy kopiec i wyciągamy je z kopca aż będzie on pusty, dostaniemy uporządkowaną listę elementów.
- Kopiec można również traktować jak priorytetową kolejkę - w końcu zawsze wyjmujemy z niego element o ekstremalnym priorytecie.

```
In [1]: paste
from heapq import heappush, heappop

def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]
## -- End pasted text --

In [2]: import random

In [3]: random_list = random.sample(range(100), 10)

In [4]: random_list
Out[4]: [15, 46, 88, 19, 69, 64, 26, 77, 78, 32]

In [5]: sorted_list = heapsort(random_list)

In [6]: sorted_list
Out[6]: [15, 19, 26, 32, 46, 64, 69, 77, 78, 88]
```

<https://www.youtube.com/watch?v=t0Cq6tVNRBA>

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy



# Refleksja - `isinstance`, `issubclass`

- W poprzednim module omawialiśmy programowanie obiektowe. Jednym z jego podstawowych mechanizmów jest dziedziczenie.
- W trakcie wykonania programu możemy łatwo sprawdzić czy dany obiekt jest instancją danej klasy. Służy do tego wbudowana funkcja **`isinstance`**.
- Podobnie możemy sprawdzić czy dana klasa dziedziczy z innej klasy. Służy do tego funkcja **`issubclass`**.

```
In [1]: class A:
...:     pass
...:

In [2]: class B:
...:     pass
...:

In [3]: class C(B, A):
...:     pass
...:

In [4]: a = A()

In [5]: isinstance(a, A)
Out[5]: True

In [6]: isinstance(a, B)
Out[6]: False

In [7]: c = C()

In [8]: isinstance(c, C)
Out[8]: True

In [9]: isinstance(c, A)
Out[9]: True

In [10]: isinstance(c, B)
Out[10]: True
```

```
In [11]: issubclass(C, A)
Out[11]: True

In [12]: issubclass(A, C)
Out[12]: False

In [13]: issubclass(A, B)
Out[13]: False

In [14]: issubclass(B, A)
Out[14]: False
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy



# Refleksja - hasattr, getattr

- Podczas wykonania programu można również dynamicznie sprawdzić, czy dany obiekt posiada jakiś atrybut. Służy do tego wbudowana funkcja **hasattr**.
- **hasattr** jako pierwszy parametr przyjmuje konkretny obiekt, w którym poszukujemy atrybutu. Jako drugi parametr przyjmuje zaś nazwę atrybutu, który szukamy w postaci ciągu znaków. Wynikiem działania funkcji jest boolowska wartość **True/False**.
- Aby pobrać wartość atrybutu (oprócz najbardziej oczywistego sposobu - operatora kropki) można użyć wbudowanej funkcji **getattr**.
- **getattr** jako pierwszy parametr przyjmuje konkretny obiekt, z którego wyciągamy atrybut. Jako drugi parametr przyjmuje nazwę atrybutu, którego wartość chcemy poznać w postaci ciągu znaków. Trzecim, opcjonalnym parametrem jest domyślna wartość, która będzie zwrócona jeśli obiekt nie ma takiego atrybutu.

```
In [2]: user = UserData('Maciej', 'Tarnowski', 'maciej@gmail.com',  
...: 'Mazowieckie', 'Kapitulna 9', 'Warszawa', 'Polska', '01-123')  
  
In [3]: hasattr(user, 'first_name')  
Out[3]: True  
  
In [4]: hasattr(user, 'middle_name')  
Out[4]: False  
  
In [5]: getattr(user, 'first_name')  
Out[5]: 'Maciej'  
  
In [6]: getattr(user, 'middle_name')  
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-6-117649628b05> in <module>  
----> 1 getattr(user, 'middle_name')  
  
AttributeError: 'UserData' object has no attribute 'middle_name'  
  
In [7]: getattr(user, 'middle_name', 'brak drugiego imienia')  
Out[7]: 'brak drugiego imienia'
```



# Klasy i metody abstrakcyjne - ABC

- Czasami klasy bazowe, po których dziedziczymy, informują jedynie jakie metody muszą być zaimplementowane w klasach pochodnych ale nie dostarczają konkretnej implementacji.
- Takie klasy nazywamy abstrakcyjnymi. Dokładnie każda klasa, która posiada co najmniej jedną abstrakcyjną metodę (taką, która posiada jedynie sygnaturę ale nie dostarcza jej implementacji) jest nazywana abstrakcyjną.
- Nie można stworzyć bezpośrednio obiektu klasy, która jest abstrakcyjna, najpierw trzeba dostarczyć konkretną klasę, dziedziczącą po abstrakcyjnej, która implementuje wszystkie abstrakcyjne metody.
- Np. możemy stworzyć abstrakcyjną klasę reprezentującą instrument muzyczny. Wiemy, że każdy instrument potrafi jakoś grać i to jest nasza abstrakcyjna metoda.
- Każdy konkretny instrument będzie dostarczał implementacji abstrakcyjnej metody **play**.

```
1  import abc
2
3
4  class MusicalInstrument(abc.ABC):
5
6      @abc.abstractmethod
7      def play(self):
8          pass
```





# Klasy i metody abstrakcyjne - ABC

- Do stworzenia abstrakcyjnej klasy jest nam potrzebny standardowy moduł **abc** (**A**bstr**a**ct **B**ase **C**lasses).
- Klasa abstrakcyjna powinna dziedziczyć po **abc.ABC** aby zaznaczyć fakt, że jest abstrakcyjna.
- Ponadto powinna posiadać co najmniej jedną metodę udekorowaną dekoratorem **@abc.abstractmethod**. W naszym przypadku jest to metoda **play**. Jak widać, nie dostarczamy żadnej implementacji tej metody, od razu piszemy **pass**.
- Chcemy stworzyć konkretną klasę **Guitar** dlatego w deklaracji, klasy zaznaczamy, że dziedziczymy z **MusicalInstrument**. Jednak nasza definicja nie jest poprawna, co sygnalizuje nam PyCharm podkreślając nazwę naszej klasy.
- Problem polega na tym, że nie dostarczyliśmy w klasie **Guitar** konkretnej implementacji metody **play**.

```
1  import abc
2
3
4  class MusicalInstrument(abc.ABC):
5
6      @abc.abstractmethod
7      def play(self):
8          pass
```

```
11 class Guitar(MusicalInstrument):
12     pass
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy



# Klasy i metody abstrakcyjne - ABC

- Teraz nasza konkretna klasa jest zdefiniowana poprawnie - gitara mówi nam konkretnie jakie dźwięki z siebie wydaje.
- Ponadto PyCharm na marginesie prezentuje nam przyciski, dzięki którym możemy się wygodnie przełączać pomiędzy abstrakcyjną deklaracją metody a jej konkretnymi implementacjami w klasach pochodnych - nawet jeśli klasy są w oddzielnych plikach.

```
1 import abc
2
3
4 class MusicalInstrument(abc.ABC):
5
6     @abc.abstractmethod
7     def play(self):
8         pass
9
10
11 class Guitar(MusicalInstrument):
12     def play(self):
13         return "Brzdek, brzdek"
```

```
In [1]: paste
import abc

class MusicalInstrument(abc.ABC):

    @abc.abstractmethod
    def play(self):
        pass

class Guitar(MusicalInstrument):
    def play(self):
        return "Brzdek, brzdek"
## -- End pasted text --

In [2]: some_instrument = MusicalInstrument()

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a5b6411af71c> in <module>
----> 1 some_instrument = MusicalInstrument()

TypeError: Can't instantiate abstract class MusicalInstrument with abstract methods play

In [3]: my_gibson = Guitar()

In [4]: my_gibson.play()
Out[4]: 'Brzdek, brzdek'
```

- Na przykładzie jasno widać, że nie da się stworzyć obiektu klasy abstrakcyjnej.

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy





# Klasy i metody abstrakcyjne - ABC

- Możemy stworzyć więcej klas instrumentów i zestawić je w orkiestrę!
- Możemy ponadto stworzyć funkcję, która dyryguje instrumentami.
- Najlepsze jest to, że funkcja wcale nie musi wiedzieć, jakimi konkretnie instrumentami dyryguje. Wszystko co jest wystarczy to wiedza, że każdy instrument, aby mógł nazywać się instrumentem, musi dostarczać metodę **play**, więc można ją wykonać i wtedy instrument zagra w sposób specyficzny dla swojego rodzaju.
- Ta możliwość jednolitego traktowania obiektów różnych klas o wspólnej bazie jest nazywana **polimorfizmem**, który oprócz dziedziczenia i abstrakcji jest kolejną ważną koncepcją programowania obiektowego.

```
12 class Guitar(MusicalInstrument):
13     def play(self):
14         return "Brzdęk, brzdęk"
15
16
17 class Flute(MusicalInstrument):
18     def play(self):
19         return "Fiu, fiu!"
20
21
22 class Violin(MusicalInstrument):
23     def play(self):
24         return "Skrzyp, skrzyp!!"
25
26
27 def conductor(instruments:
28               typing.Sequence[MusicalInstrument]
29               ) → None:
30     for instrument in instruments:
31         print(instrument.play())
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

# Klasy i metody abstrakcyjne - ABC



- Możemy stworzyć więcej klas instrumentów i zestawić je w orkiestrę!
- Możemy ponadto stworzyć funkcję, która dyryguje instrumentami.
- Najlepsze jest to, że funkcja wcale nie musi wiedzieć, jakimi konkretnie instrumentami dyryguje. Wszystko co jest wystarczy to wiedza, że każdy instrument, aby mógł nazywać się instrumentem, musi dostarczać metodę **play**, więc można ją wykonać i wtedy instrument zagra w sposób specyficzny dla swojego rodzaju.
- Ta możliwość jednolitego traktowania obiektów różnych klas o wspólnej bazie jest nazywana polimorfizmem, który oprócz dziedziczenia i abstrakcji jest kolejną ważną koncepcją programowania obiektowego.

```
In [2]: orchestra = [Guitar(), Violin(), Flute()]

In [3]: conductor(orchestra)
Brzdęk, brzdęk
Skrzyp, skrzyp!!
Fiu, fiu!
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy

# Klasy i metody abstrakcyjne - ABC



- Możemy stworzyć więcej klas instrumentów i zestawić je w orkiestrę!
- Możemy ponadto stworzyć funkcję, która dyryguje instrumentami.
- Najlepsze jest to, że funkcja wcale nie musi wiedzieć, jakimi konkretnie instrumentami dyryguje. Wszystko co jest wystarczy to wiedza, że każdy instrument, aby mógł nazywać się instrumentem, musi dostarczać metodę **play**, więc można ją wykonać i wtedy instrument zagra w sposób specyficzny dla swojego rodzaju.
- Ta możliwość jednolitego traktowania obiektów różnych klas o wspólnej bazie jest nazywana polimorfizmem, który oprócz dziedziczenia i abstrakcji jest kolejną ważną koncepcją programowania obiektowego.

```
In [2]: orchestra = [Guitar(), Violin(), Flute()]

In [3]: conductor(orchestra)
Brzdęk, brzdęk
Skrzyp, skrzyp!!
Fiu, fiu!
```

# 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.



Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

# 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): ^

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy

# 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 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



- 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.
- **\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 - 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.
- 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.
- Funkcja **fullmatch** również przyjmuje dokładnie takie same parametry. Tym razem sprawdzane jest czy cały tekst pasuje do wyrażenia.
- Funkcja **findall** zwraca wszystkie wystąpienia wyrażenia regularnego w tekście. Zwracana jest lista wyników (obiektów typu **Match**)
- 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.
- 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.
- Funkcja **sub** zamieni wszystkie ciągi opisane wyrażeniem regularnym na podany ciąg znaków a jej wariant **subn** zwróci również informację o tym ile zamian przeprowadzono.

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy

# 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 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.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.

Autor: Michał Nowotka



# 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', 'Ała ma kota')
Out[2]: 'Ała ma psa'

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

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development  
Academy

# 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 wprzód (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](#).

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development

Academy



# 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 101

SAVE & SHARE  
Save Regexp %s

FLAVOR  
PCRE (PHP)  
ECMAScript (JavaScript)  
Python ✓  
Golang

REGULAR EXPRESSION  
r"(?<=[A-Z])" gm

TEST STRING  
AlaMaKota

EXPLANATION  
▼ "(?<=[A-Z])" gm  
▼ Positive Lookahead (?=[A-Z])  
Assert that the Regexp 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](#).

# Podsumowanie



- Do pobierania atrybutu warto użyć `getattr`
- Do sprawdzenia czy atrybut istnieje używamy `hasattr`
- Moduł `abc` wykorzystujemy do tworzenia abstrakcji

# Plan na kolejne zajęcia



- Wyrażenia lambda (funkcje anonimowe)
- Wyjątki
- Funkcje wewnętrzne