Zaawansowane programowanie w Pythonie

Wykład 6-7-8

Agnieszka Zbrzezny

KMMI WMII UWM

18/21/25 listopada 2024

References

- Regular expressions
  https://pl.wikipedia.org/wiki/Wyrażenie_regularne
- Standard library documentation
  https://docs.python.org/3/library/re.html
- Regular Expression HOWTO
  https://docs.python.org/3/howto/regex.html
- Python 3 Module of the Week
  https://pymotw.com/3/re/index.html
- Tutorial on `www.tutorialspoint.com`
  https://www.tutorialspoint.com/python3/python_reg_expressions.htm
- A web-based tools for testing regular expressions
  https://pythex.org
  https://regex101.com/

Wyrażenia regularne

Wprowadzenie

- Wyrażenia regularne to sposób opisywania zbioru ciągów znaków na podstawie cech wspólnych dla każdego ciągu w tym zbiorze.
- Można ich używać do wyszukiwania, edycji lub manipulacji tekstem i danymi.
- Aby tworzyć wyrażenia regularne, należy poznać specyficzną składnię – taką, która wykracza poza normalną składnię języka programowania Python.
- Wyrażenia regularne różnią się stopniem złożoności, ale gdy zrozumie się podstawy ich konstrukcji, będzie można rozszyfrować (lub utworzyć) dowolne wyrażenie regularne.

Wyrażenia regularne

Wprowadzenie

- Załóżmy na przykład, że chcemy odnaleźć odnośniki w pliku HTML. Musimy szukać ciągów znaków pasujących do wzorca `<a href="...">`.
- Jednak to nie wszystko – mogą pojawić się dodatkowe odstępy lub URL może być zamknięty w pojedynczych cudzysłowach.
- Wyrażenia regularne udostępniają precyzyjną składnię pozwalającą określać, jakie ciągi liter są poprawnym dopasowaniem.
- W dalszej części zobaczymy składnię wyrażeń regularnych wykorzystywaną w module `re` i dowiemy się, w jaki sposób korzystać z wyrażeń regularnych.

Wyrażenia regularne

Składnia wyrażeń regularnych

- W wyrażeniach regularnych wszystkie znaki oprócz wymienionych poniżej 14 znaków zastrzeżonych oznaczają takie same znaki:
  `. * + ? { } | ( ) [ ] \ ^ $`
- Na przykład wyrażenie regularne `Python` pasuje jedynie do ciągu znaków `Python`.
- Znak `.` jest dopasowywany do dowolnego znaku. Przykładowo, `.a.a` zostanie dopasowane zarówno do `masa`, jak i `data`.
- Znak `*` oznacza, że poprzedzające konstrukcje mogą być powtórzone 0 lub więcej razy; dla znaku `+` jest to 1 lub więcej razy.
- Przyrostek `?` oznacza, że konstrukcja jest opcjonalna (0 lub 1 raz). Przykładowo, `be+s?` dopasowuje się do `be` , `bee` oraz `bees`.
- Można określić inne liczby powtórzeń za pomocą `{ }` – szczegółowe informacje znajdują się na dalszych slajdach.

Wyrażenia regularne

Składnia wyrażeń regularnych

- Klasa znaków (ang. character class) jest zestawem alternatywnych znaków zamkniętych w nawiasach, takim jak `[Jj]`, `[0-9]`, `[A-Za-z]` czy `[^0-9]`.
- Wewnątrz klasy znaków znak `-` służy do opisu zakresu (wszystkich znaków, których wartości Unicode leżą pomiędzy dwoma krańcowymi wartościami).
- Jednak znak `-` będący pierwszym lub ostatnim znakiem klasy znaków reprezentuje sam siebie.
- Znak `^` jako pierwszy znak w klasie znaków oznacza dopełnienie (wszystkie znaki oprócz określonych w klasie).
- Istnieje wiele predefiniowanych klas znaków, takich jak `\d` (cyfry), `\w` (znaki alfanumeryczne), czy `\s` (białe znaki).

Wyrażenia regularne

Składnia wyrażeń regularnych

- Znaki `^` oraz `$` są dopasowywane do początku i końca danych wejściowych.
- Aby wykorzystać metaznak
  `. * + ? { } | ( ) [ ] \ ^ $`
  należy poprzedzić go znakiem `\`.
- Metaznaki (z wyjątkiem `\`) nie są aktywne wewnątrz klas.
- Przykładowo, `[akm$]` dopasuje dowolny ze znaków `a`, `k`, `m` lub `$`.
- `$` jest zwykle metaznakiem, ale wewnątrz klasy znaków jest pozbawiony swojej specjalnej natury.
- Wewnątrz klasy znaków należy poprzedzać znakiem `\` jedynie znaki `[` oraz `\`, oczywiście uwzględniając położenie znaków `\ - ^`.
- Przykładowo, klasa `[]^-]` zawiera wszystkie trzy wymienione znaki.

Wyrażenia regularne

Składnia wyrażeń regularnych

- Surowe ciągi znaków (ang. raw strings) ułatwiają pisanie wyrażeń regularnych w Pythonie, zmniejszając potrzebę użycia znaków odwrotnego ukośnika (backslash'a).
- Aby uzyskać surowy łańcuch znaków poprzedzamy łańcuch znaków literą `r`. Python będzie traktował wszystko w takim łańcuchu jako znaki kodowane w trybie Raw-Unicode-Escape.
- Wewnątrz surowego łańcucha znaków znaki specjalne, zawierajace odwrotny ukośnik nie są specjalnie interpretowane.
- Przykłady:

```
"ab*"                   r"ab*"
"\\ten"                 r"\ten"
"\\w+\\s+"              r"\w+\s+"
```

Wyrażenia regularne

Klasy znaków

Klasa znaków	Dopasuj
`.`	dowolny znak z wyjątkiem znaku nowej linii
`\d`	dowolną cyfrę 0–9
`\D`	dowolny znak niebędący cyfrą
`\s`	dowolny biały znak, w tym `\t`, `\n` i `\r` oraz znak spacji
`\S`	dowolny znak niebędący białym znakiem
`\w`	dowolny znak alfanumeryczny (litera lub cyfra)
`\W`	dowolny znak niebędący alfanumerykiem
`\t`	znak tabulatora
`\n`	znak nowej linii
`\r`	znak powrotu karetki
`[...]`	jeden znak spośród zbioru znaków
`[^...]`	jeden znak spoza zbioru znaków

Wyrażenia regularne

Kwantyfikatory zachłanne

Kwantyfikator	Dopasuj poprzedzający element
`*`	`0` lub więcej razy
`+`	`1` lub więcej razy
`?`	`0` lub `1` raz
`{m}`	dokładnie `m` razy
`{m,}`	co najmniej `m` razy
`{,n}`	co najwyżej `n` razy
`{m,n}`	od `m` do `n` razy

Granice słowa

Asercja	Dopasuj
`\b`	granicę słowa w łańcuchu znaków

Wyrażenia regularne

Funkcja `re.findall`

- Składnia:
  `re.findall(pattern, string, flags=0)`
- Znajduje wszystkie dopasowania wyrażenia regularnego `pattern` w łańcuchu `string` i zwraca je w formie listy łańcuchów.

Przykład

```
>>> import re
>>> string = "black, blue and brown"
>>> regex = r"bl\w+"
>>> matches = re.findall(regex, string)
>>> print(matches)
['black', 'blue']
```

Wyrażenia regularne

Funkcja `re.compile`

- Jeżeli chcemy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go.
- Obiekty wzorców, powstałe w wyniku kompilacji wyrażeń regularnych, posiadają metody analogiczne do funkcji modułu `re`.
- Metody te nie przyjmują pierwszego parametru `pattern`, gdyż są wywoływane dla skompilowanego obiektu wzorca.

Przykład

```
>>> import re
>>> string = "black, blue and brown"
>>> pattern = re.compile(r"bl\w+")
>>> matches = pattern.findall(string)
>>> print(matches)
['black', 'blue']
```

Wyrażenia regularne

Przykład

```
>>> import re
string = "Ewa ma kota i żółwia"
>>>
>>> re.findall(r"\w\w", string)
['Ew', 'ma', 'ko', 'ta', 'żó', 'łw', 'ia']
>>>
>>> re.findall(r"\w+", string)
['Ewa', 'ma', 'kota', 'i', 'żółwia']
>>>
>>> re.findall(r"\w{3}", string)
['Ewa', 'kot', 'żół', 'wia']
>>>
>>> re.findall(r"\b\w{3}\b", string)
['Ewa']
>>>
```

Wyrażenia regularne

Przykład

```
>>> re.findall(r"\w{1,4}", string)
['Ewa', 'ma', 'kota', 'i', 'żółw', 'ia']
>>>
>>> re.findall(r"\b\w{1,4}\b", string)
['Ewa', 'ma', 'kota', 'i']
>>>
>>> re.findall(r"\w{3,4}", string)
['Ewa', 'kota', 'żółw']
>>>
>>> re.findall(r"\b\w{3,4}\b", string)
['Ewa', 'kota']
>>>
>>> re.findall(r"\w{3,}", string)
['Ewa', 'kota', 'żółwia']
```

Wyrażenia regularne

Funkcja `re.search`

- `re.search(pattern, string [, pos [, endpos]])`
  Przeszukuje łańcuch w poszukiwaniu pierwszego miejsca, w którym wyrażenie regularne `pattern` daje dopasowanie i zwraca odpowiadający mu obiekt dopasowania (obiekt klasy `re.Match`).
- Zwraca `None`, jeżeli żadna pozycja w łańcuchu nie pasuje do wzorca; zauważmy, że różni się to od znalezienia dopasowania o zerowej długości w jakimś miejscu łańcucha.
- Opcjonalny drugi parametr `pos` podaje indeks w łańcuchu, od którego ma się rozpocząć szukanie; domyślnie wynosi on `0`.
- Opcjonalny parametr `endpos` określa, jak daleko łańcuch będzie przeszukiwany: w poszukiwaniu dopasowania będą szukane tylko znaki od `pos` do `endpos - 1`.
- Jeżeli `endpos < pos`, to nie zostanie znalezione dopasowanie.

Wyrażenia regularne

Odnajdywanie jednego dopasowania

- Jeżeli `endpos >= pos`, to
  `re.search(pattern, string, pos, endpos)`
  jest równoważne
  `re.search(pattern, string[:endpos], pos)`.

```
import re
regex = r"[+-]?\d+"
text = "Ewa ma 12 kotów i 3 psy"

match = re.search(regex, text)
if match:
    print("Znaleziono:", match.span())
else:
    print("Nie znaleziono")

# Znaleziono: (7, 9)
```

Wyrażenia regularne

Odnajdywanie jednego dopasowania

- Jeżeli chcemy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go:

```
import re

pattern = re.compile(r"[+-]?\d+")
text = "Ewa ma 12 kotów i 3 psy";

match = pattern.search(text)
if match:
    print("Znaleziono:", match.span())
else:
    print("Nie znaleziono")


# Znaleziono: (7, 9)
```

Wyrażenia regularne

Funkcja `re.match`

- `re.match(pattern, string [, pos [, endpos]])`
  Szuka dopasowania wyrażenia regularnego `pattern`, ale tylko na początku łańcucha `string`, czym różni się od funkcji `search`.
- Jeżeli zero lub więcej znaków na początku łańcucha pasuje do tego wyrażenia regularnego, to zwraca odpowiedni obiekt dopasowania (obiekt klasy `re.Match`).
- Zwraca `None`, jeżeli łańcuch nie pasuje do wzorca; zauważmy, że różni się to od dopasowania o zerowej długości.
- Opcjonalne argumenty `pos` i `endpos` mają takie samo znaczenie jak w funkcji `search`.

Wyrażenia regularne

Odnajdywanie jednego dopasowania

- Jeżeli chcemy ustalić, czy ciąg znaków pasuje do początku wyrażenia, należy skorzystać z metody `match`:

```
import re
regex = r"[+-]?\d+"
text = "Ewa ma 12 kotów i 3 psy";

match = re.match(regex, text)
if match:
    print("Znaleziono:", match.span())
else:
    print("Nie znaleziono")


# Nie znaleziono
```

Wyrażenia regularne

Odnajdywanie jednego dopasowania

- Jeżeli chcemy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go:

```
import re

pattern = re.compile(r"[+-]?\d+")
text = "Ewa ma 12 kotów i 3 psy";

match = pattern.match(text)
if match:
    print("Znaleziono:", match.span())
else:
    print("Nie znaleziono")


# Nie znaleziono
```

Wyrażenia regularne

Grupy przechwytujące

- Często trzeba uzyskać więcej informacji niż tylko to, czy wyrażenie regularne (w skrócie RE) pasuje, czy nie.
- RE są często używane do rozcinania łańcuchów na kilka podgrup, które pasują do różnych interesujących nas elementów.
- Przykładowo, wiersz nagłówka RFC-822 jest podzielony na nazwę nagłówka i wartość, oddzielone znakiem ":", jak poniżej:

```
Message-Id: <20001114144603.00abb310@oreilly.com>
Date: Tue, 14 Nov 2000 14:55:07 -0500
To: "Fredrik Lundh" <fredrik@effbot.org>
From: Frank
Subject: Re: python library book!

Where is it?
```

- Można napisać wyrażenie regularne, które pasuje do całego wiersza nagłówka i ma jedną grupę, która pasuje do nazwy nagłówka, oraz

Wyrażenia regularne

Grupy przechwytujące

- Grupy są oznaczane metaznakami `(` oraz `)`.
- Mają one takie samo znaczenie jak w wyrażeniach matematycznych: grupują wyrażenia zawarte wewnątrz nich.
- Zawartość grupy można powtórzyć za pomocą kwantyfikatora powtórzenia, takiego jak `*`, `+`, `?` lub `{m,n}`.
- Przykładowo, `(ab)*` będzie odpowiadać zeru lub większej liczbie powtórzeń `ab`.

```
>>> p = re.compile("(a)b")
>>> m = p.match("ab")
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Wyrażenia regularne

Grupy przechwytujące

- Grupy oznaczone metaznakami `(` oraz `)` przechwytują również indeks początkowy i końcowy tekstu, który dopasowują.
- Można go pobrać, przekazując jako argument do metod obiektu klasy `re.Match`: `group()`, `start()`, `end()` i `span()`.
- Grupy są numerowane począwszy od `0`. Grupa `0` jest zawsze obecna; jest to całe RE, więc metody obiektu dopasowania mają grupę `0` jako domyślny argument.

```
>>> p = re.compile("(a)b")
>>> m = p.match("ab")
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Wyrażenia regularne

Grupy przechwytujące

- Podgrupy są numerowane od lewej do prawej strony, od 1 w górę.
- Grupy mogą być zagnieżdżone; aby określić ich liczbę, wystarczy policzyć znaki nawiasów otwierających, idąc od lewej do prawej.
- Gdy metodzie `group()` przekażemy wiele numerów grup naraz, to zwróci ona krotkę zawierającą wartości dla tych grup.

```
>>> p = re.compile("(a(b)c)d")
>>> m = p.match("abcd")
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
>>> m.group(2, 1, 2)
('b', 'abc', 'b')
```

Wyrażenia regularne

Grupy przechwytujące

- Metoda `groups()` zwraca krotkę zawierającą łańcuchy wszystkich podgrup, począwszy od grupy nr 1.

```
>>> p = re.compile("(a(b)c)d")
>>> m = p.match("abcd")
>>> m.groups()
('abc', 'b')
```

- Odwołania wsteczne we wzorcu pozwalają określić, że zawartość wcześniejszej grupy przechwytującej musi zostać znaleziona również w bieżącym miejscu łańcucha.
- Przykładowo, `\1` powiedzie się tylko wtedy, gdy na bieżącej pozycji zostanie znaleziona dokładna zawartość grupy 1.
- Aby umożliwić wstawianie dowolnych znaków do łańcucha, należy używać surowego łańcucha podczas włączania odwołań wstecznych do RE.

Wyrażenia regularne

Funkcja `re.findall`

- `re.findall(pattern, string, flags=0)`
  Zwraca wszystkie nienakładające się na siebie dopasowania wzorca w łańcuchu, jako listę łańcuchów lub krotek.
- Łańcuch jest skanowany od lewej do prawej, a dopasowania są zwracane w kolejności znalezienia. Puste dopasowania są uwzględniane w wyniku.
- Wynik zależy od liczby grup przechwytujących we wzorcu. Jeżeli nie ma żadnych grup, to zwracana jest lista łańcuchów pasujących do całego wzorca.
- Jeżeli istnieje dokładnie jedna grupa, to zwracana jest lista łańcuchów pasujących do tej grupy.
- Jeżeli występuje wiele grup, to zwracana jest lista krotek łańcuchów pasujących do grup.

Wyrażenia regularne

Funkcja `re.findall`

- Grupy, które nie zostały przechwycone, nie mają wpływu na postać wyniku.
- Począwszy od wersji 3.7 dopasowania niepuste mogą rozpoczynać się tuż po poprzednim dopasowaniu pustym.

Przykład (Funkcja `re.findall`)

```
>>> string = "which foot or hand fell fastest"
>>> re.findall(r"\bf[a-z]*", string)
['foot', 'fell', 'fastest']
>>> string = "set width=20 and height=10"
>>> re.findall(r"(\w+)=(\d+)", string)
[('width', '20'), ('height', '10')]
```

Odnajdywanie wszystkich dopasowań

- Aby znaleźć wszystkie dopasowania można wykorzystać funkcję `re.findall`, która zwraca listę wszystkich dopasowań.

```
import re

regexp = r".a"
text = "Ewa ma kota i psa"

matches = re.findall(regexp, text)
print(matches)


# ['wa', 'ma', 'ta', 'sa']
```

Wyrażenia regularne

Odnajdywanie wszystkich dopasowań

- Jeżeli musimy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go:

```
import re

regexp = r".a"
text = "Ewa ma kota i psa"

pattern = re.compile(regexp)
matches = pattern.findall(text)
print(matches)


# ['la', 'ma', 'ta', 'sa']
```

Wyrażenia regularne

Odnajdywanie wszystkich dopasowań będących słowami

- Aby znaleźć wszystkie dopasowania będące słowami można wykorzystać funkcję `re.findall` oraz odpowiednio zmodyfikować wyrażenie regularne:

```
import re

regexp = r"\b.a\b"
text = "Ewa ma kota i psa"

matches = re.findall(regexp, text)
print(matches)


# ['ma']
```

Wyrażenia regularne

Funkcja `re.finditer`

- `re.finditer(pattern, string, flags=0)`
  Zwraca iterator dostarczający obiekty dopasowania dla wszystkich nie pokrywających się dopasowań dla wzorca `pattern` w łańcuchu `string`.
- Łańcuch jest skanowany od lewej do prawej, a dopasowania są zwracane w znalezionej kolejności.
- Puste dopasowania są uwzględniane w wyniku.
- Począwszy od wersji 3.7 dopasowania niepuste mogą rozpoczynać się tuż po poprzednim dopasowaniu pustym.

Wyrażenia regularne

Odnajdywanie wszystkich dopasowań

- Aby znaleźć wszystkie dopasowania można wykorzystać funkcję `re.finditer`, która zwraca iterator do wszystkich dopasowań.

```
import re

regexp = r".a"
text = "Ewa ma kota i psa"

iterator = re.finditer(regexp, text)
for match in iterator:
    s = match.string[match.start():match.end()]
    print(s, end=" ")
print()


# wa ma ta sa
```

Wyrażenia regularne

Odnajdywanie wszystkich dopasowań

- Jeżeli musimy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go:

```
import re

regexp = r".a"
text = "Ewa ma kota i psa"

pattern = re.compile(r".a")
for match in pattern.finditer(text):
    s = match.string[match.start():match.end()]
    print(s, end=" ")
print()


# wa ma ta sa
```

Wyrażenia regularne

Testowanie wyrażeń regularnych

- Zdefiniujemy teraz funkcję `reth` służącą do badania konstrukcji wyrażeń regularnych obsługiwanych przez moduł `re`.
- Poleceniem uruchamiającym tę funkcję jest:

```
>>> from reth import reth
>>> reth(r"...") # ... to dowolne wyrażenie regularne

Enter input string (press <CTRL>+D to finish):
```

- Funkcja zapętla się, prosząc użytkownika o podanie łańcucha wejściowego.
- Użycie tego narzędzia testowego jest opcjonalne, ale może okazać się wygodne podczas poznawania przypadków testowych omawianych na poprzednim i na dzisiejszym wykładzie.

Wyrażenia regularne

Testowanie wyrażeń regularnych

```
import re

def reth(regex):
    print("RE:", regex)
    try:
        pattern = re.compile(regex)
    except Exception as ex:
        print(ex)
        return
    while True:
        try:
            string = input("\nEnter input string"
                + " (press <CTRL>+C to finish):\n")
        except BaseException:
            print()
            break
        ...
```

Wyrażenia regularne

Testowanie wyrażeń regularnych

```
import re

def reth(regex):
    ...
    while True:
        ...
        found = False
        for matched in pattern.finditer(string):
            start, end = matched.start(), matched.end()
            s = matched.string[start:end]
            print(f"I found the text \"{s}\""
                + f" starting at index {start}"
                + f" and ending at index {end}.")
            found = True
        if not found:
            print("No match found.\n")
```

Wyrażenia regularne

Funkcja `re.split`

- `re.split(pattern, string, maxsplit=0, flags=0)`
  Dzieli łańcuch źródłowy według wystąpień wzorca, zwracając listę zawierającą wynikowe podłańcuchy.
- Jeżeli we wzorcu użyto nawiasów przechwytujących, to jako część wynikowej listy zwracane są również teksty wszystkich grup we wzorcu.
- Jeżeli `maxsplit` jest niezerowe, to nastąpi co najwyżej `maxsplit` podziałów, a pozostała część łańcucha jest zwracana jako ostatni element listy.
- Jeżeli w separatorze znajdują się grupy przechwytujące, a separator jest dopasowany na początku łańcucha, wynik będzie zaczynał się od pustego łańcucha. To samo dotyczy końca łańcucha.

Wyrażenia regularne

Przykład (Funkcja re.split)

```
>>> re.split(r"\W+", "Words, words, words.")
['Words', 'words', 'words', '']
>>>
>>> re.split(r"(\W+)", "Words, words, words.")
['Words', ', ', 'words', ', ', 'words', '.', '']
>>>
>>> re.split(r"\W+", "Words, words, words.", 1)
['Words', 'words, words.']
>>>
>>> re.split("[a-f]+", "0a3B9", flags=re.IGNORECASE)
['0', '3', '9']
>>>
>> re.split(r"(\W+)", "...words, words...")
['', '...', 'words', ', ', 'words', '...', '']
```

Wyrażenia regularne

Kwantyfikatory niezachłanne

- Kwantyfikatory niezachłanne `*?`, `+?`, `??` oraz `{m,n}?` dopasowują się do jak najkrótszych fragmentów tekstu.

Kwantyfikator	Dopasuj
`*?`	0 lub więcej wystąpień
`+?`	1 lub więcej wystąpień
`??`	0 lub 1 wystąpienie
`{m}?`	dokładnie m wystąpień
`{m,}?`	co najmniej m wystąpień
`{,n}?`	co najwyżej n wystąpień
`{m,n}?`	od m do n wystąpień

Wyrażenia regularne

Przykład (Kwantyfikator zachłanny)

```
import re

text = '<button type="submit" class="btn">Send</button>'

pattern = '".+"'
matches = re.findall(pattern, text)
print("text:", text)
print("pattern", pattern)
print("matches:", matches)


matches: ['"submit" class="btn"']
```

Wyrażenia regularne

Przykład (Kwantyfikator niezachłanny)

```
import re

text = '<button type="submit" class="btn">Send</button>'

pattern = '".+?"'
matches = re.findall(pattern, text)
print("text:", text)
print("pattern", pattern)
print("matches:", matches)


matches: ['"submit"', '"btn"']
```

Wyrażenia regularne

Funkcja `re.sub`

- `re.sub(pattern, repl, string, count=0, flags=0)` Zwraca łańcuch znaków uzyskany przez zastąpienie najbardziej po lewej stronie nienakładających się wystąpień wzorca `pattern` w łańcuchu `string` przez łańcuch `repl`. Jeżeli wzorzec nie został znaleziony, łańcuch jest zwracany bez zmian.
- Argument `repl` może być łańcuchem znaków lub funkcją. Jeżeli jest to łańcuch, to przetwarzane są wszystkie występujące w nim odwrotne ukośniki. To znaczy:
  - `\n` jest konwertowane na znak nowej linii, `\r` na powrót karetki itd.
  - Nieznane ucieczki od liter ASCII są zachowane do przyszłego użytku i traktowane jako błędy.
  - Inne nieznane ucieczki, takie jak `\&`, są pozostawiane bez zmian.
  - Odsyłacze wsteczne, takie jak `\1`, są zastępowane podłańcuchem pasującym do grupy nr `1` we wzorcu. Puste dopasowania wzorca są zastępowane, jeśli sąsiadują z niepustym dopasowaniem.

Wyrażenia regularne

Przykład (Funkcja re.sub)

```
>>> regex = r"\*{2}(.*?)\*{2}"
>>> repl = r"<b>\1</b>"
>>> text = "Make this **bold**.  This **too**."
>>> print("Bold:", re.sub(regex, repl, text))
Bold: Make this <b>bold</b>.  This <b>too</b>.
```

Przykład (Metoda Pattern.sub)

```
>>> bold = re.compile(r"\*{2}(.*?)\*{2}")
>>> repl = r"<b>\1</b>"
>>> text = "Make this **bold**.  This **too**."
>>> print("Bold:", bold.sub(repl, text))
Bold: Make this <b>bold</b>.  This <b>too</b>.
```

Wyrażenia regularne

Funkcja `re.sub`

- Jeżeli `repl` jest funkcją, to jest ona wywoływana dla każdego nienakładającego się wystąpienia wzorca. Funkcja ta przyjmuje jako argument obiektu klasy `re.Match` i zwraca łańcuch zastępujący, który zostanie użyty do zamian.

Przykład (Funkcja `re.sub`)

```
import re

def square(match):
    num = int(match.group())
    return str(num * num)

print(re.sub(r"\d+", square, "A1 A2 A3 A4"))

# A1 A4 A9 A16
```

Wyrażenia regularne

Funkcja `re.subn`

- `re.subn(pattern, replacement, string)`
  Działa tak samo jak funkcja `sub`, ale zwraca krotkę:
  (zmodyfikowany łańcuch, liczba zamian).

Przykład (Funkcja `re.subn`)

```
>>> regex = r"\*{2}(.*?)\*{2}"
>>> repl = r"<b>\1</b>"
>>> text = "Make this **bold**.  This **too**."
>>> print("Bold:", re.subn(regex, repl, text))
Bold: ('Make this <b>bold</b>.  This <b>too</b>.', 2)
```

Wyrażenia regularne

Operator alternatywy

- Wyrażenie `A|B`, gdzie `A` i `B` mogą być dowolnymi RE, tworzy wyrażenie regularne, które będzie pasowało do `A` lub `B`. W ten sposób można oddzielić dowolną liczbę RE znakiem `|`.
- Operatora alternatywy można tego również używać wewnątrz grup.
- Podczas skanowania docelowego łańcucha znaków, wyrażenia regularne oddzielone znakiem `|` są sprawdzane od lewej do prawej strony.
- Gdy jeden z wzorców jest całkowicie zgodny, to ta gałąź jest akceptowana.
- Oznacza to, że gdy A zostanie dopasowane, B nie będzie dalej testowane, nawet jeśli dałoby to dłuższe dopasowanie.
- Innymi słowy, operator `|` nigdy nie jest zachłanny.

Wyrażenia regularne

Operator alternatywy

- Aby dopasować dosłowne `|`, należy użyć `\|` lub zawrzeć je wewnątrz klasy znaków, jak w `[|]`.
- Alternatywa jest przydatna, gdy musimy dopasować jedną z kilku różnych alternatyw.
- Przykładowo, RE `airways|airplane|bomber` dopasuje każdy tekst, który zawiera `airways`, `airplane` lub `bomber`.
- To samo można osiągnąć, używając RE `air(ways|plane)|bomber`.
- Gdybyśmy użyli RE `(airways|airplane|bomber)`, dopasowałoby ono dowolne z tych trzech wyrażeń.

Wyrażenia regularne

Operator alternatywy

- Rozważmy RE `(air(ways|plane)|bomber)`, który ma dwa przechwycenia, jeżeli pasuje pierwsze wyrażenie (`airways` lub `airplane` jako pierwsze przechwycenie i `ways` lub `plane` jako drugie przechwycenie), oraz jedno przechwycenie, jeżeli pasuje drugie wyrażenie (`bomber`).
- Możemy wyłączyć efekt przechwytywania, umieszczając po nawiasie otwierającym znaki `?:`, jak poniżej:
  `(air(?:ways|plane)|bomber)`
- To wyrażenie będzie miało tylko jedno przechwycenie, jeżeli będzie pasowało (`airways` lub `airplane` lub `bomber`).

Wyrażenia regularne

Przykład (Operator alternatywy)

```
>>> import re
>>> s = "airways aircraft airplane bomber"
>>> result1 = re.findall(r"(airways|airplane|bomber)", s)
>>> print(result1)
['airways', 'airplane', 'bomber']
>>>
>>> result2 = re.findall(r"(air(ways|plane)|bomber)", s)
>>> print(result2)
[('airways', 'ways'), ('airplane', 'plane'), ('bomber', '')]
>>>
>>> result3 = re.findall(r"(air(?:ways|plane)|bomber)", s)
>>> print(result3)
['airways', 'airplane', 'bomber']
>>>
```

Wyrażenia regularne

Przykład (Zliczanie liczby unikalnych słów w pliku)

```
# count_words_func.py

import re

def count_words(file):
    content = file.read().lower()
    words = re.findall(r"[^\W\d_]+", content)
    words = sorted(set(words))
    return len(words)
```

Wyrażenia regularne

Przykład (Zliczanie liczby unikalnych słów w pliku)

```
import sys
from count_words_func import count_words

def main():
    if len(sys.argv) != 2:
        print(f"Usage: python3 {sys.argv[0]} file")
        sys.exit(1)
    try:
        file = open(sys.argv[1])
    except Exception as ex:
        print(ex)
        sys.exit(2)
    how_many = count_words(file)
    file.close()
    print(f"There are {how_many} unique words in", sys.argv[1])

main()
```

Wyrażenia regularne – asercje

Asercja	Dopasuj
`^ $`	odpowiednio początek i koniec tekstu, także początek nowej linii (koniec linii) w przypadku włączonej opcji `re.MULTILINE`
`\A \Z`	odpowiednio początek tekstu i koniec tekstu
`\b`	pusty łańcuch znaków na początku lub końcu słowa (dopasowuje granicę słowa albo początek lub koniec tekstu)
`\B`	pusty łańcuch znaków, lecz nie na początku lub końcu słowa (dopasowanie wewnątrz słowa)
`(?=e)`	łańcuch znaków, jeżeli bezpośrednio po nim następuje wyrażenie pasujące do `e` (ang. positive lookahead)
`(?!e)`	łańcuch znaków, jeżeli bezpośrednio po nim nie następuje wyrażenie pasujące do `e` (ang. negative lookahead)
`(?<=e)`	łańcuch znaków, jeżeli bezpośrednio przed nim następuje wyrażenie pasujące do `e` (ang. positive lookbehind)
`(?<!e)`	łańcuch znaków, jeżeli bezpośrednio przed nim nie następuje wyrażenie pasujące do `e` (ang. negative lookbehind)

Wyrażenia regularne

Przykład (Lookahead i lookbehind)

```
>>> import re
# Lookahead
>>> re.findall("(Bobby)(?= Fischer)", "Bobby Fischer")
['Bobby']
>>>
>>> re.findall("(Bobby)(?! Fischer)", "Bobby Charlton")
['Bobby']
>>>
# Lookbehind
>>> re.findall("(?<=Bobby )Fischer", "Bobby Fischer")
['Fischer']
>>>
>>> re.findall("(?<!Bobby )Fischer", "Robert Fischer")
['Fischer']
```

Wyrażenia regularne

Flagi

- `re.ASCII` lub `re.A` dla wybranych klas znaków takich jak `\w`, `\b`, `\s` oraz `\d` oraz ich dopełnień dopasowuje tylko znaki ASCII.
- `re.DEBUG`: wyświetla informacje dotyczące debugowania dotyczące skompilowanego wyrażenia.
- `re.DOTALL` lub `re.S`: sprawia, że symbol jest dopasowywany do wszystkich znaków, włącznie ze znakami końca linii.
- `re.IGNORECASE` lub `re.I`: dopasowuje znaki niezależnie od wielkości liter.
- `re.MULTILINE` lub `re.M`: sprawia, że `^` i `$` są dopasowywane do początku i końca wiersza, a nie do całości danych wejściowych.
- `re.VERBOSE` lub `re.X`: pozwala na pisanie wyrażeń regularnych, które są bardziej czytelne, poprzez umożliwienie wizualnego oddzielanie sekcji logicznych wzorca i dodawanie komentarzy.

Wyrażenia regularne

Przykład (Flaga re.ASCII)

```
>>> re.findall(r"\w+", 'fox:ąćęłńóśźż')
['fox', 'ąćęłńóśźż']
>>>
>>> re.findall(r"\w+", 'fox:ąćęłńóśźż', flags=re.A)
['fox']
>>>
re.findall(r"[a-zA-Z0-9_]+", "fox:ąćęłńóśźż")
[]
```

Wyrażenia regularne

Przykład (Flaga re.IGNORECASE)

```
>>> bool(re.search(r"cat", "Cat"))
False
>>> bool(re.search(r"cat", "Cat", re.I))
True
>>> string = "Cat cot CATER ScUtTLe"
>>> re.findall(r'c.t', string)
['cot', 'cUt']
>>>
>>> re.findall(r"c.t", string, flags=re.I)
['Cat', 'cot', 'CAT', 'cUt']
>>>
>>> re.findall(r"[a-z]+", "New York")
['ew', 'ork']
>>>
>>> re.findall(r"[a-z]+", "New York", re.I)
['New', 'York']
```

Uwagi dodatkowe

- Zwróćmy uwagę, że gdy wzorce Unicode `[a-z]` lub `[A-Z]` są używane w połączeniu z flagą `re.IGNORECASE`, pasują one do 52 liter ASCII i 4 dodatkowych liter spoza ASCII:
  - `İ` (U+0130, łacińska wielka litera `I` z kropką nad nią),
  - `ı` (U+0131, łacińska mała litera `i` bez kropki),
  - `ſ` (U+017F, łacińska mała litera długie `s`) oraz
  - `K` (U+212A, znak Kelvina).
- Jeżeli użyta jest flaga `re.ASCII`, dopasowywane są tylko litery od `a` do `z` oraz od `A` do `Z`.

Przykład (Flagi `re.ASCII` i `re.IGNOREACSE`)

- Program `a_band_of_four.py`

Wyrażenia regularne

Przykład (Flaga `re.DOTALL`)

```
regex = r"the.*ice"
>>> text = "Hi there\nHave a Nice Day"
>>>
>>> re.sub(regex, "X", text)
'Hi there\nHave a Nice Day'
>>>
>>> re.sub(regex, "X", text, flags=re.S)
'Hi X Day'
```

Wyrażenia regularne

Przykład (Flaga re.MULTILINE)

```
>>> re.findall(r"^top", "hi hello\ntop spot")
[]
>>>
>>> re.findall(r"^top", "hi hello\ntop spot", flags=re.M
['top']
>>>
>>> re.findall(r"ar$", "spare\npar\ndare")
[]
>>>
>>> re.findall(r"ar$", "spare\npar\ndare", flags=re.M)
['ar']
```

Wyrażenia regularne

Flagi

```
import re

pattern = re.compile(r"[a-z]+")
text = "Ewa ma 12 kotów i 3 psy";

match = pattern.search(text)
if match:
    s = match.string[match.start():match.end()]
    print("Znaleziono:", s)
else:
    print("Nie znaleziono")


# Znaleziono: wa
```

Wyrażenia regularne

Flagi

```
import re

pattern = re.compile(r"[a-z]+", re.IGNORECASE)
pattern = re.compile(r"(?i)[a-z]+")
text = "Ewa ma 12 kotów i 3 psy";

match = pattern.search(text)
if match:
    s = match.string[match.start():match.end()]
    print("Znaleziono:", s)
else:
    print("Nie znaleziono")


# Znaleziono: Ewa
```

Wyrażenia regularne

Flagi

```
import re

charref = re.compile("&#(0o[0-7]+"
                     "|[0-9]+"
                     "|0x[0-9a-fA-F]+);")

print(charref.findall("&#0o377;&#255;&#0xFF;"))
print(charref.findall("&#0o377;"))
print(charref.findall("&#255;"))
print(charref.findall("&#0xFF;"))
```

Wyrażenia regularne

Flagi

```
import re

charref = re.compile(r"""
 &[#]                    # Start of a numeric entity
 (                       # reference
     0o[0-7]+            # Octal form
   | [0-9]+              # Decimal form
   | 0x[0-9a-fA-F]+      # Hexadecimal form
 )
 ;                       # Trailing semicolon
""", re.VERBOSE)

print(charref.findall("&#0o377;&#255;&#0xFF;"))
```

Wyrażenia regularne

Przykład (Flaga re.MULTILINE)

```
import re

def main():
    text = "This is some text -- with punctuation.\
        \nA second line."
    pattern = r"(^\w+)|(\w+\S*$)"
    single_line = re.compile(pattern)
    multiline = re.compile(pattern, re.MULTILINE)

    print("Text:\n  {!r}".format(text))
    print("Pattern:\n  {}".format(pattern))
    print("Single Line :")
    for match in single_line.findall(text):
        print("  {!r}".format(match))
    print("Multline    :")
    for match in multiline.findall(text):
        print("  {!r}".format(match))
```

Wyrażenia regularne

Przykład (Flaga re.DOTALL)

```
import re

def main():
    text = "This is some text -- with punctuation.\
        \nA second line."
    pattern = r"(^\w+)|(\w+\S*$)"
    single_line = re.compile(pattern)
    multiline = re.compile(pattern, re.MULTILINE)

    print("Text:\n  {!r}".format(text))
    print("Pattern:\n  {}".format(pattern))
    print("Single Line :")
    for match in single_line.findall(text):
        print("  {!r}".format(match))
    print("Multline    :")
    for match in multiline.findall(text):
        print("  {!r}".format(match))
```

Flagi wbudowane

- Aby zastosować flagi do określonych fragmentów RE, należy określić je w specjalnej składni grupującej.
- Spowoduje to zastąpienie flag zastosowanych do całych definicji RE, jeżeli takie istnieją.

Flaga	Alias	Flaga wbudowana
`re.ASCII`	`re.A`	`(?a)`
`re.DEBUG`	`N/A`	`N/A`
`re.IGNORECASE`	`re.I`	`(?i)`
`re.LOCALE`	`re.L`	`(?L)`
`re.MUTILINE`	`re.M`	`(?m)`
`re.DOTALL`	`re.S`	`(?s)`
`re.VERBOSE`	`re.X`	`(?x)`

Flagi wbudowane

- Dostępne są następujące warianty składni:
  - `(?flags:patttern)` zastosuje flagi tylko dla wyrażenia `pattern`
  - `(?-flags:pattern)` zaneguje flagi tylko dla wyrażenia `pattern`
  - `(?flags-flags:patttern)` zastosuje i zaneguje poszczególne flagi tylko dla wyrażenia `pattern`
  - `(?flags)` zastosuje flagi dla całej definicji RE, może być podany tylko na początku definicji RE
- Jeżeli potrzebne są kotwice, należy je podać po `(?flags)`.
- W ten sposób flagi mogą być określane dokładnie tylko tam, gdzie są potrzebne.
- Flagi należy podawać jako małe, jednoliterowe wersje stałych o krótkiej formie. Przykładowo, `i` dla `re.I`, `s` dla `re.S` i tak dalej, z wyjątkiem `L` dla `re.L` lub `re.LOCALE`.

Wyrażenia regularne

Flagi wbudowane

- Jak pokazują poniższe przykłady, flagi wbudowane nie działają jako grupy przechwytujące.

Przykład (Flagi wbudowane)

```
>>> string = "Cat SCatTeR CATER cAts"
>>> re.findall(r"Cat[a-z]*\b", string)
['Cat']
>>> re.findall(r"Cat(?i:[a-z]*)\b", string)
['Cat', 'CatTeR']
>>> re.findall(r"Cat[a-z]*\b", string, flags=re.I)
['Cat', 'CatTeR', 'CATER', 'cAts']
>>> re.findall(r"(?i)Cat[a-z]*\b", string)
['Cat', 'CatTeR', 'CATER', 'cAts']
>>> re.findall(r"(?-i:Cat)[a-z]*\b", string, flags=re.I)
['Cat', 'CatTeR']
```

Wyrażenia regularne – grupowanie

Nawiasy oprócz zwykłej funkcji, wpływania na kolejność obliczeń, pełnią drugą ważną rolę: tworzą z wyrażenia w nawiasach tzw. grupę.

Wyrażenie	Znaczenie
`(...)`	dopasowanie wyrażenia w nawiasie jako grupy; po dopasowaniu można odwoływać się we wzorcu do zawartości grupy poprzez odwołania wsteczne `\1`, `\2`, `\3`, itd.
`(?:...)`	nawiasy nieprzechwytujące; od zwykłych nawiasów różnią się tym, że po dopasowaniu nie można odwoływać się do zawartości grupy poprzez odwołania wsteczne.
`(?P<name>...)`	tworzy grupę nazwaną `name`.
`(?P=name)`	dopasowuje tekst, który został dopasowany wcześniej przez grupę nazwaną `name`.
`(?(1)w1|w2)`	wyrażenie warunkowe. Jeżeli pierwsza grupa przechwytująca dopasowała porcję tekstu, dopasuj wyrażenie `w1`. a w przeciwnym przypadku dopasuj wyrażenie `w2`.

Wyrażenia regularne

Przykłady

- Program `regexdemo.py`
- Program `funcs.py`

Wyrażenia regularne

Przykład (Flag wbudowane)

```
>>> type(re.IGNORECASE)
<enum 'RegexFlag'>
>>>
>> for j in range(9):
...       print(f"{2**j:3}:", re.RegexFlag(2**j))
...
  1: re.TEMPLATE
  2: re.IGNORECASE
  4: re.LOCALE
  8: re.MULTILINE
 16: re.DOTALL
 32: re.UNICODE
 64: re.VERBOSE
128: re.DEBUG
256: re.ASCII
```

Obsługa wyliczeń

Przykład (Iteracje po elementach wyliczenia)

```
# Iteracja po elementach wyliczenia
# nie dostarcza aliasów:
>>> from enum import Enum
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> for element in Shape:
...     print(repr(element))
...
<Shape.DIAMOND: 1>
<Shape.SQUARE: 2>
<Shape.CIRCLE: 3>
```

Obsługa wyliczeń

Przykład

```
# Atrybut specjalny __members__ jest uporządkowanym,
# przeznaczonym tylko do odczytu odwzorowaniem nazw na
# elementy. Zawiera on wszystkie nazwy zdefiniowane
# w wyliczeniu, w tym aliasy:
>>>
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

Obsługa wyliczeń

Przykład

```
# Atrybut __members__ może być użyty do szczegółowego,
# programowego dostępu do elementów wyliczenia.
# Na przykład do znalezienia wszystkich aliasów:
>>>
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...     ALIAS_FOR_CIRCLE = 3
...
>>> [name for name, member in Shape.__members__.items()\
        if member.name != name]
['ALIAS_FOR_SQUARE', 'ALIAS_FOR_CIRCLE']
```

Obsługa wyliczeń

Przykład

```
# Elementy wyliczenia są porównywane według tożsamości:
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
# Porównania porządkowe między wartościami wyliczenia
# nie są obsługiwane, ponieważ elementy wyliczenia
# nie są liczbami całkowitymi.
>>>
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported\
between instances of 'Color' and 'Color'
```

Obsługa wyliczeń

Przykład

```
# Porównania równości są jednak zdefiniowane:
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
>>>
# Porównania z wartościami nie-wyliczeniowymi
# zawsze będą zwracać wartość False:
>>> Color.BLUE == 2
False
```

Obsługa wyliczeń

Przykład (Funkcyjne API)

```
# Klasa Enum jest wywoływalna i udostępnia
# następujący interfejs API:
>>> Animal = Enum("Animal", "ANT BEE CAT DOG")
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> for animal in Animal: print(repr(animal))
...
<Animal.ANT: 1>
<Animal.BEE: 2>
<Animal.CAT: 3>
<Animal.DOG: 4>
```

Obsługa wyliczeń

Funkcyjne API dla klasy `Enum`

- Semantyka tego API (od ang. Application Programming Interface, czyli Interfejs programowania aplikacji) przypomina semantykę klasy `namedtuple` z modułu `collections`.
- Pierwszym argumentem wywołania funkcji `Enum` jest nazwa wyliczenia; drugim argumentem jest źródło nazw elementów wyliczenia.
- Może to być łańcuch nazw oddzielonych białymi znakami, sekwencja nazw, sekwencja 2-krotek z parami klucz-wartość lub odwzorowanie (np. słownik) nazw na wartości.
- Dwie ostatnie opcje umożliwiają przypisanie wyliczeniom dowolnych wartości.
- Pozostałe automatycznie przypisują rosnące liczby całkowite, począwszy od `1` (aby określić inną wartość początkową, należy użyć argumentu `start`).

Obsługa wyliczeń

Funkcyjne API dla klasy `Enum`

- Zwracana jest nowa klasa wywodząca się z `Enum`.
- Innymi słowy, uprzednie przypisanie do zmiennej `Animal` jest równoważne z poniższym:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

- Powodem domyślnego ustawiania `1` jako liczby początkowej, a nie `0`, jest to, że `0` ma wartość `False`, natomiast wszystkie elementy wyliczenia przyjmują wartość `True`.

Obsługa wyliczeń

Klasa `IntEnum`

- Pierwsza udostępniona odmiana `Enum` jest również podklasą `int`.
- Elementy klasy `IntEnum` mogą być porównywane z liczbami całkowitymi. Także elementy różnych podklas klasy `IntEnum` mogą być porównywane między sobą:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1; SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1; GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

Obsługa wyliczeń

Klasa IntEnum

- Jednak nadal nie można ich porównać do standardowych wyliczeń Enum:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

Obsługa wyliczeń

Klasa IntEnum

- Wartości `IntEnum` zachowują się jak liczby całkowite w sposób, jakiego można się spodziewać:

```
>>> int(Shape.CIRCLE)
1
>>> Shape.CIRCLE + 5
6
>>> ["a", "b", "c"][Shape.CIRCLE]
'b'
>>> [j for j in range(Shape.SQUARE)]
[0, 1]
```

Obsługa wyliczeń

Przykład (Klasa Shape)

```
# Obiekty klasy Shape są jednocześnie
# obiektami klasy int oraz klasy IntEnum:
>>> from enum import IntEnum
>>>
>>> class Shape(IntEnum):
...     CIRCLE = 1; SQUARE = 2
...
>>> issubclass(Shape, int)
True
>>> issubclass(Shape, IntEnum)
True
>>> isinstance(Shape.CIRCLE, int)
True
>>> isinstance(Shape.CIRCLE, IntEnum)
True
```

Operatory bitowe (ang. bitwise operators)

Operatory bitowe (w kolejności malejących priorytetów)

- `~` – negacja bitowa
- `<<` – przesunięcie w lewo
- `>>` – przesunięcie w prawo
- `&` – koniunkcja bitowa
- `^` – rozłączna alternatywa bitowa
- `|` – alternatywa bitowa

Uwagi

- Operatory bitowe są zdefiniowane tylko dla obiektów typu całkowitego.
- Operacje bitowe na obiektach typu całkowitego są wykonywane bit po bicie.

Operatory bitowe (ang. bitwise operators)

Bitowe negacja i koniunkcja

- Negacja bitowa ~ jest operacją jednoargumentową, która wykonuje negację na każdym bicie.
  Bity, które mają wartość 0, stają się 1, a te, które mają wartość 1, stają się 0.
- Koniunkcja bitowa & jest operacją dwuargumentową, która wykonuje koniunkcję na każdej parze odpowiednich bitów.
  Jeżeli oba bity w odpowiadającej sobie pozycji mają wartość 1, to bit wynikowy będzie mieć wartość 1; w przeciwnym razie bit wynikowy to 0.

Operatory bitowe (ang. bitwise operators)

Bitowe alternatywy

- Rozłączna alternatywa bitowa ^ jest operacją dwuargumentową, która wykonuje alternatywę rozłączną na każdej parze odpowiednich bitów. Jeżeli dokładnie jeden bit z dwóch na odpowiadającej sobie pozycji ma wartość 1, to bit wynikowy będzie mieć wartość 1; w przeciwnym razie bit wynikowy to 0.
- Alternatywa bitowa | jest operacją dwuargumentową, która wykonuje alternatywę na każdej parze odpowiednich bitów.
  Jeżeli co najmniej jeden bit z dwóch na odpowiadającej sobie pozycji ma wartość 1, to bit wynikowy będzie mieć wartość 1; w przeciwnym razie bit wynikowy to 0.

Operatory bitowe (ang. bitwise operators)

Tabela operatorów bitowych

X	Y	~X	~Y	X & Y	X ^ Y	X | Y
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1

Operatory bitowe

Przykład (Bitowa reprezentacja liczb całkowitych: to_bin.py)

```
def main():
    print("23: ", to_bin(23, 16))
    print("-1: ", to_bin(-1, 16))

def to_bin(num, length):
    binary = ""
    mask = 2**(length - 1)
    while mask > 0:
        binary += "1" if (num & mask) else "0"
        mask //= 2
    return binary

if __name__ == "__main__":
    main()
```

Operatory bitowe (ang. bitwise operators)

Arytmetyczne przesunięcia bitowe

- Przesunięcia bitowe traktują wartość jako serię bitów, a nie jako wielkość liczbową. W tych operacjach bity są przesuwane w lewo lub w prawo.
- Rejestry w procesorze komputerowym mają stałą szerokość, więc niektóre bity zostaną „przesunięte” z rejestru na jednym końcu, podczas gdy ta sama liczba bitów zostanie „przesunięta” z drugiego końca.
- W przesunięciu arytmetycznym bity, które są przesunięte z dowolnego końca, są odrzucane.
- W przesunięciu arytmetycznym w lewo w miejsce przesuwanych bitów pojawiają się zera; w przesunięciu arytmetycznym w prawo, bit znaku (MSB w uzupełnieniu do dwójki) jest powielany, zachowując w ten sposób znak operandu.

Operatory bitowe

Przykład (Przesunięcie w lewo liczby dodatniej)

```
from to_bin import to_bin

>>> 26624, to_bin(26624, 16)
(26624, '0110100000000000')
>>> 26624 << 1, to_bin(26624 << 1, 16)
(53248, '1101000000000000')
>>> 26624 << 2, to_bin(26624 << 2, 16)
(106496, '1010000000000000')
>>> 26624 << 3, to_bin(26624 << 3, 16)
(212992, '0100000000000000')
>>> 26624 << 4, to_bin(26624 << 4, 16)
(425984, '1000000000000000')
>>> 26624 << 5, to_bin(26624 << 5, 16)
(851968, '0000000000000000')
>>> 26624 << 5, to_bin(26624 << 5, 32)
(851968, '00000000000011010000000000000000')
```

Operatory bitowe

Przykład (Przesunięcie w lewo liczby ujemnej)

```
from to_bin import to_bin

>>> -2048, to_bin(-2048, 16)
(-2048, '1111100000000000')
>>> -2048 << 1, to_bin(-2048 << 1, 16)
(-4096, '1111000000000000')
>>> -2048 << 2, to_bin(-2048 << 2, 16)
(-8192, '1110000000000000')
>>> -2048 << 3, to_bin(-2048 << 3, 16)
(-16384, '1100000000000000')
>>> -2048 << 4, to_bin(-2048 << 4, 16)
(-32768, '1000000000000000')
>>> -2048 << 5, to_bin(-2048 << 5, 16)
(-65536, '0000000000000000')
```

Operatory bitowe

Przykład (Przesunięcie w prawo liczby dodatniej)

```
from to_bin import to_bin

>>> 25, to_bin(25, 16)
(25, '0000000000011001')
>>> 25 >> 1, to_bin(25 >> 1, 16)
(12, '0000000000001100')
>>> 25 >> 2, to_bin(25 >> 2, 16)
(6, '0000000000000110')
>>> 25 >> 3, to_bin(25 >> 3, 16)
(3, '0000000000000011')
>>> 25 >> 4, to_bin(25 >> 4, 16)
(1, '0000000000000001')
>>> 25 >> 5, to_bin(25 >> 5, 16)
(0, '0000000000000000')
```

Operatory bitowe

Przykład (Przesunięcie w prawo liczby ujemnej)

```
from to_bin import to_bin

>>> -38, to_bin(-38, 16)
(-38, '1111111111011010')
>>> -38 >> 1, to_bin(-38 >> 1, 16)
(-19, '1111111111101101')
>>> -38 >> 2, to_bin(-38 >> 2, 16)
(-10, '1111111111110110')
>>> -38 >> 3, to_bin(-38 >> 3, 16)
(-5, '1111111111111011')
>>> -38 >> 4, to_bin(-38 >> 4, 16)
(-3, '1111111111111101')
>>> -38 >> 5, to_bin(-38 >> 5, 16)
(-2, '1111111111111110')
>>> -38 >> 6, to_bin(-38 >> 6, 16)
(-1, '1111111111111111')
```

Obsługa wyliczeń

Klasa `IntFlag`

- Kolejna podlasa klasy `Enum`, a mianowicie klasa `IntFlag`, jest także podklasą klasy `int`.
- Powoduje to, że elementy `IntFlag` mogą być łączone za pomocą operatorów bitowych (`&`, `|`, `^`, `~`), a wynikiem jest nadal element `IntFlag`.
- Jednakże, jak sama nazwa wskazuje, elementy `IntFlag` są również instancjami klasy `int` i mogą być używane wszędzie tam, gdzie używane są elementy `int`.
- Wynik każdej operacji na elementach klasy `IntFlag`, poza operacjami bitowymi, powoduje utratę przynależności tego wyniku do klasy `IntFlag`.

Obsługa wyliczeń

Przykład (Klasa dziedzicząca po IntFlag)

```
>>> from enum import IntFlag
>>>
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
>>> Perm.X in RW
False
```

Obsługa wyliczeń

Przykład

```
# Możliwe jest także nadawanie nazw kombinacjom:
>>>
>>> class Perm(IntFlag):
...         R = 4
...         W = 2
...         X = 1
...         RWX = 7
...
>>> Perm.RWX
<Perm.RWX: 7>
>>>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Obsługa wyliczeń

Przykład

```
# Inną ważną różnicą między IntFlag a Enum jest to,
# że jeżeli nie ustawiono żadnych flag (wartość 0),
# to wartość logiczna wynosi False:
>>>
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
>>>
# Ponieważ instancje klasy IntFlag są również instancjam
# klasy int, to można łączyć jedne z drugimi:
>>> Perm.X | 8
<Perm.8|X: 9>
```

Obsługa wyliczeń

Klasa `IntFlag`

- Ostatnią podklasą klasy `Enum` jest klasa `Flag` klasy `int`.
- Podobnie jak w przypadku `IntFlag`, elementy `Flag` można łączyć przy użyciu operatorów bitowych (`&`, `|`, `^`, `~`),
- W przeciwieństwie do `IntFlag`, nie można ich łączyć ani porównywać z żadnym innym wyliczeniem `Flag` ani `int`.
- Chociaż możliwe jest bezpośrednie określenie wartości, zaleca się użycie wartości `auto()` i pozwolenie klasie `Flag` na wybranie odpowiedniej wartości.

Obsługa wyliczeń

Przykład

```
# Podobnie jak dla IntFlag, jeżeli kombinacja elementów
# nie powoduje ustawienia żadnej flagi, to wartością
# logiczną wyniku jest False:
>>>
>>> from enum import Flag, auto
>>>
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Obsługa wyliczeń

Przykład

```
# Poszczególne flagi powinny mieć wartości będące
# potęgami dwójki (1, 2, 4, 8, ...), podczas gdy
# kombinacje flag nimi nie będą:
>>>
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Obsługa wyliczeń

Przykład

```
# Nadanie nazwy warunku ,,brak ustawionych flag''
# nie zmienia jego wartości logicznej:
>>>
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>>
>>> bool(Color.BLACK)
False
```

Obsługa wyliczeń

Uwagi końcowe

- W przypadku większości nowego kodu, `Enum` i `Flag` są zdecydowanie zalecane, ponieważ `IntEnum` i `IntFlag` łamią pewne semantyczne obietnice dla wyliczeń (poprzez porównywalność z liczbami całkowitymi, a tym samym przez przechodniość do innych niepowiązanych wyliczeń).
- `IntEnum` i `IntFlag` powinny być używane tylko w przypadkach, gdy `Enum` i `Flag` nie działają; na przykład, gdy stałe całkowite są zastępowane wyliczeniami lub w celu współdziałania z innymi systemami.