

Regular Expression pada Python

Dewasa ini, *Regular Expression* telah digunakan secara meluas dan memegang peranan yang cukup penting. Bagaimana pula penerapannya dengan bahasa Python?

Banyak programmer yang melakukan *pattern matching* secara manual. Sebagai contoh, banyak programmer yang memeriksa validitas alamat e-mail secara manual. Memeriksa satu per satu karakter misalnya. Semua hal tersebut sebenarnya bisa dilakukan dengan bantuan Regular Expression.

Apa itu Regular Expression?

Regular Expression adalah suatu cara menggambarkan susunan pola dalam suatu kalimat. Regular Expression ini banyak digunakan dalam *text processing*.

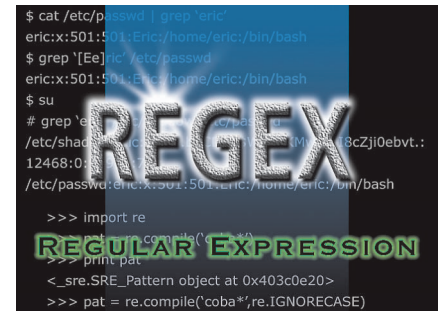
Anda mungkin bertanya-tanya apa sih kegunaan Regular Expression yang sering disingkat regex ini. Pembahasan regex di sini ditujukan untuk pemula sehingga pembahasannya tidak akan terlalu mendalam. Sebagai ilustrasi dari pemakaian regex, kita akan mengambil contoh berikut. Misalnya Anda memiliki sebuah daftar nama dan nomor telepon, Anda mungkin ingin mencari pemilik nomor yang digit ke-3 nya adalah 8 dan digit terakhirnya adalah 9. Atau pun apabila Anda ingin mengekstrak semua alamat e-mail ataupun website dalam suatu dokumen. Anda akan membutuhkan Regular Expression.

Contoh lainnya apabila Anda ingin mengganti semua kata '*large*' dengan kata '*besar*' dalam sebuah dokumen. Dalam hal ini, Regular Expression mendukung fitur searching and replacing sehingga dapat menghemat waktu Anda. Fungsi *search* dan *replace* yang acap kali dijumpai pula pada teks editor juga dibuat berdasarkan Regular Expression. Selain itu, Regular Expression juga sering digunakan untuk memvalidasi input-an dari user misalnya pada validasi inputan pada form suatu halaman web.

Banyak utilitas Linux yang menyisipkan kemampuan regex dalam programnya untuk manipulasi teks. Misalnya beberapa *command-line tools* di lingkungan Linux seperti *grep*, *sed*, *awk* menggunakan regex. Bahkan kini fasilitas Find Files di KDE juga telah dilengkapi dengan Regular Expression editor. Untuk memanggil fasilitas *Find Files* ini, Anda bisa menekan Alt+F2 dan mengetikkan *kfind*. Anda dapat menemukan regex editor pada bagian tab kedua Contents.

```
$ cat /etc/passwd | grep 'eric'
eric:x:501:501:Eric:/home/eric:/bin/bash
$ grep '[Ee]ric' /etc/passwd
eric:x:501:501:Eric:/home/eric:/bin/bash
$ su
# grep 'eric' /etc/shadow /etc/passwd
/etc/shadow:eric:$1$w.
bnCDZo$WpXoKMwvd/I8cZji0ebvt.:
12468:0:99999:7:::
/etc/passwd:eric:x:501:501:Eric:/home/
eric:/bin/bash
```

Di atas adalah salah satu contoh penggunaan perintah *grep* yang digunakan untuk menyeleksi teks pada suatu file di mana juga mendukung ekspresi reguler. Untuk memahami lebih lanjut, kita akan membahas tentang pola pencocokan yang digunakan dalam ekspresi reguler. Pada umumnya, suatu rangkaian karakter atau kata akan cocok dengan karakter atau kata itu sendiri. Misalnya ekspresi '*coba*' akan cocok dengan kata '*coba*' sendiri. Kita bisa mengaktifkan fitur *case-insensitive* di mana ekspresi '*coba*' akan cocok dengan '*COBA*' atau '*Coba*' dst. Ada pengecualian untuk aturan ini di mana ada karakter-karakter tertentu yang tidak cocok dengan karakter itu sendiri. Karakter-karakter tersebut dinamakan metakarakter.



Metakarakter

Metakarakter mempunyai makna khusus dalam penggunaan regex. Beberapa metakarakter yang sering digunakan untuk membuat regex adalah sebagai berikut:

```
$ ^ [ ] . * + ? / { } | ( )
```

Pertama adalah metakarakter '^' untuk menandakan pencocokan dimulai dari awal baris dan metakarakter '\$' untuk menandakan pencocokan dimulai dari akhir baris. Misalnya ekspresi reguler '^coba' akan cocok dengan 'coba' yang didapat dari awal baris dan ekspresi 'coba\$' akan cocok dengan 'coba' didapat dari akhir baris. Selanjutnya kita membahas penggunaan '[' dan ']'. Kedua tanda tersebut digunakan untuk menentukan kelas karakter yang ingin dicocokkan.

Kita bisa menulis karakter-karakter yang ingin dicocokkan ataupun range karakter yang ingin dicocokkan dengan menggunakan tanda '-'. Misalnya ekspresi '[abc]' akan cocok dengan karakter 'a', 'b', dan 'c'. Ekspresi reguler '[abc]' akan sama hasilnya dengan '[a-c]'. Bila kita ingin mencocokkan semua karakter kapital, Anda cukup membuat ekspresi reguler '[A-Z]'. Perlu diingat metakarakter tidak aktif di dalam kelas.

Di dalam kelas, metakarakter kehilangan makna khususnya. Misalnya ekspresi reguler '[abc\$]' akan cocok dengan 'a', 'b', 'c' dan '\$'; di mana '\$' kehilangan makna khususnya. Anda dapat mengkomplemenkan set karakter dalam kelas dengan menggunakan metakarakter '^'. Dengan demikian ekspresi '[^a-c]' akan cocok dengan karakter selain 'a', 'b', dan 'c'.

Berikutnya adalah metakarakter '\' yang digunakan untuk menghilangkan makna khusus dari suatu metakarakter. Misalnya apabila Anda ingin mencocokkan karakter '[' ataupun '\$', Anda cukup memberikan

tanda '\' sebelum karakter tersebut untuk menghilangkan makna khususnya: \l atau \\$. Dalam Python metakarakter '\' biasa diikuti berbagai karakter untuk mewakili berbagai set khusus.

\d ekuivalen dengan kelas [0-9]

\D ekuivalen dengan kelas [^0-9]

\s ekuivalen dengan kelas [\t\n\f\v\r]

\S ekuivalen dengan kelas [^t\n\f\v\r]

\w ekuivalen dengan kelas [a-zA-Z0-9_]

\W ekuivalen dengan kelas [^a-zA-Z0-9_]

Untuk selengkapnya, Anda bisa lihat di dokumentasi modul re. Metakarakter '.' akan cocok dengan semua karakter kecuali karakter *newline*. Selanjutnya metakarakter '|' yang memiliki makna OR (atau). Misalnya ekspresi reguler A|B akan cocok dengan 'A' atau 'B'.

Sekarang kita akan membahas metakarakter yang digunakan untuk hal-hal yang berulang. Pertama adalah metakarakter '*' yang digunakan untuk pengulangan karakter sebelum metakarakter '*' sebanyak 0 kali atau lebih. Misalnya ekspresi reguler cob*a akan cocok dengan 'coa'('b' 0 kali), 'coba'('b' 1 kali), 'cobba'('b' 2 kali) dst. Berikutnya adalah metakarakter '+' di mana fungsinya hampir sama dengan metakarakter '*'.

Perbedaannya adalah di mana metakarakter '+' hanya memperbolehkan pengulangan minimal 1 kali, sedangkan metakarakter '*' memperbolehkan pengulangan 0 kali. Misalnya ekspresi

reguler di atas ditulis ulang menjadi cob+a, maka ekspresi tersebut hanya akan cocok dengan 'coba'('b' 1 kali), 'cobba'('b' 2 kali) dst.

Berikutnya adalah metakarakter '?' yang memperbolehkan pengulangan minimal 0 kali dan maksimal 1 sekali. Misalnya ekspresi reguler cob?a akan cocok dengan 'coa' dan 'coba'. Terakhir adalah metakarakter '{' dan '}'. Penggunaan kedua metakarakter di atas sering ditulis {m,n} di mana m menyatakan banyaknya pengulangan minimal dan n menyatakan banyaknya pengulangan maksimal.

Sebagai contoh ekspresi cob{1,3}a akan cocok dengan 'coba'('b' 1 kali), 'cobba'('b' 2 kali), 'cobbaa'('b' 3 kali). Jika nilai m tidak disertakan, maka defaultnya adalah 0. Sedangkan jika nilai n tidak disertakan maka default-nya adalah tidak terhingga (tergantung kapasitas memory).

Dengan demikian, maka ekspresi {0,} identik dengan metakarakter '*', ekspresi {1,} identik dengan metakarakter '+' dan ekspresi {0,1} identik dengan metakarakter '?'. Dan yang terakhir adalah metakarakter '(' dan ')'. Kedua metakarakter ini digunakan untuk menyatakan grup karakter. Misalnya ekspresi reguler (coba){1,3} akan cocok dengan 'coba', 'cobacoba' dan 'cobacobacoba'.

Setelah mengenal dasar-dasar dari Regular Expression, kita akan membahas tentang pemakaian regex ini pada bahasa Python melalui pemanggilan modul re.

Modul ini disertakan dalam Python sejak versi 1.5 dan telah menggunakan Perl-style Regular Expression. Bila Anda sebelumnya pernah menggunakan regex pada Perl, Anda pasti akan merasa familiar dengan regex di Python. Modul re aslinya merupakan modul yang ditulis dalam bahasa C yang kemudian di-import ke dalam Python. Selanjutnya kita akan melihat bagaimana penerapan regex dengan bahasa Python.

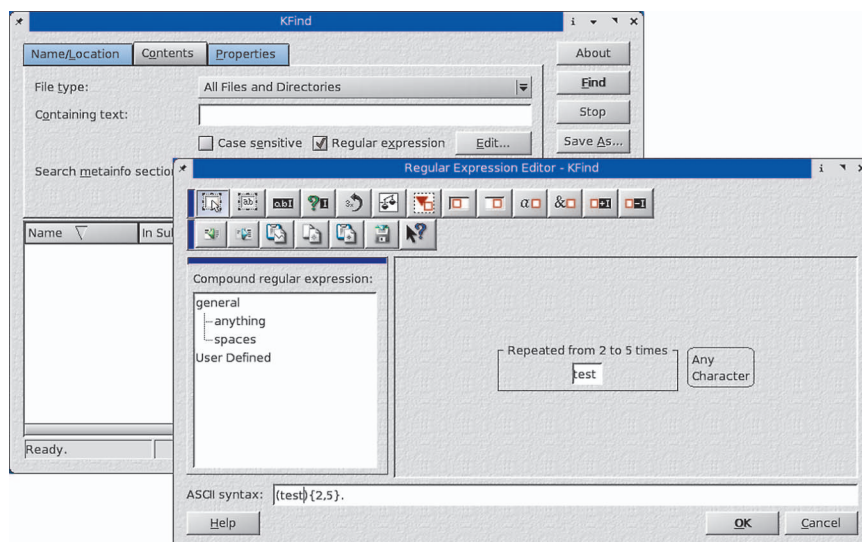
Regular Expression pada Python dikompilasi menjadi suatu instansi objek yang menyediakan metode-metode seperti *matching*, *searching* dan *replacing* serta fungsi-fungsi manipulasi teks lainnya. Untuk memakai regex ini, kita harus mengompilasinya dengan metode compile() dan bila ingin digunakan lagi, maka kita tidak perlu mengompilasi ulang regex tersebut, kita tinggal memakainya. Berikut contohnya:

```
>>> import re
>>> pat = re.compile('coba*')
>>> print pat
<_sre.SRE_Pattern object at
0x403c0e20>
>>> pat = re.compile('coba*',
re.IGNORECASE)
```

Ekspresi reguler dilewatkan sebagai paramater string pada perintah re.compile(). Ada satu hal yang perlu diperhatikan ketika menggunakan karakter '\'. Misalkan kita ingin melakukan pencocokan dengan string 'abc' maka ekspresi reguler yang tepat adalah \\abc. Namun sewaktu kita mengompilasi ekspresi reguler tersebut, perintah yang benar adalah re.compile("\\\\abc') mengingat tanda '\' dalam string Python juga memiliki makna tersendiri sehingga perlu ditambahkan dua buah karakter '\' lagi.

Untuk mengatasi masalah ini kita dapat menggunakan raw strings dengan menambahkan awalan r pada ekspresi reguler. Jadi perintah di atas dapat ditulis menjadi re.compile(r'\\abc'). Kita akan menggunakannya dalam contoh-contoh selanjutnya.

Python menawarkan dua operasi regex yang mendasar yakni match() dan search(). Perbedaan di antara keduanya, yakni match() hanya melakukan pencocokan pada awal string (atau tidak sama sekali), sedangkan search() mencari dari awal



▲ Gambar 1. Fasilitas Find Files di KDE.

sampai akhir string dan berhenti jika telah menemukan string pertama yang cocok dari sebelah kiri.

Untuk fungsi metode `match()` dan `search()` ketika menemukan string yang cocok akan mengembalikan objek `Match`. Jika tidak menemukan string yang cocok, maka akan mengembalikan objek `None`. Untuk jelasnya, mari kita lihat contoh berikut.

```
>>> import re
>>> pat = re.compile(r'Python')
>>> m = pat.match("Python")
>>> print m
<_sre.SRE_Match object at 0x40405090>
>>> m = pat.match("Java dan Python")
>>> print m
None
>>> m = pat.search("Python")
>>> print m
<_sre.SRE_Match object at 0x40405090>
>>> m = pat.search("Jython")
>>> print m
None
>>> m = pat.search("Java dan Python")
>>> print m
<_sre.SRE_Match object at 0x40405090>
```

Di bawah ini, kita akan membuat contoh sederhana penggunaannya regex dalam validasi.

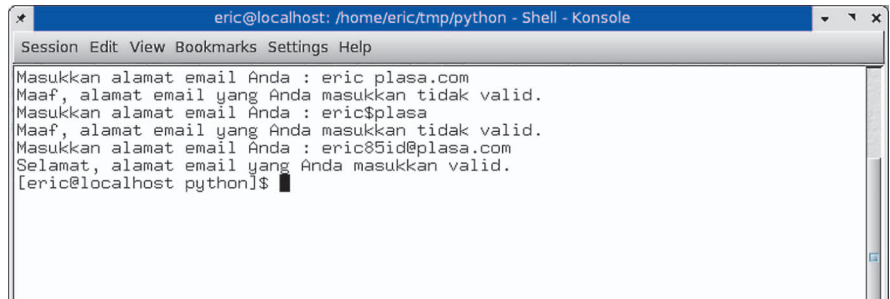
```
#test.py

#!/usr/bin/env python
import re

pat = re.compile(r'[\w.-]+@[.\w.-]+' )

while 1:
    email = raw_input('Masukkan alamat email Anda : ')
    if pat.match(email):
        print "Selamat, alamat email yang Anda masukkan valid."
        break
    else:
        print "Maaf, alamat email yang Anda masukkan tidak valid."
```

Kode di atas bila dijalankan akan menghasilkan keluaran sebagai berikut:



▲ Gambar 2. Contoh program `test.py`.

```
$ python test.py
Masukkan alamat email Anda : eric
plasa.com
Maaf, alamat email yang Anda masukkan
tidak valid.
Masukkan alamat email Anda : eric85id@plasa.com
Maaf, alamat email yang Anda masukkan
tidak valid.
Masukkan alamat email Anda :
eric85id@plasa.com
Selamat, alamat email yang Anda
masukkan valid.
```

Objek `Match` yang dihasilkan dari penerapan perintah `match()` dan `search()` mempunyai atribut dan metode sendiri juga. Anda dapat melihatnya dengan perintah `dir()`.

```
>>> import re
>>> pat = re.compile(r'[A-Z]yth?on')
>>> s = "Saya sedang belajar Python
dan Jython"
>>> m = pat.match(s)
>>> print m
None
>>> m = pat.search(s)
>>> print m
<_sre.SRE_Match object at 0x40404090>
>>> dir(m)
['_copy_', '_deepcopy_', 'end',
'expand', 'group', 'groupdict', 'groups',
'span', 'start']
```

Beberapa metode yang sering digunakan adalah `group()`, `start()`, `end()` dan `span()`. Metode `group()` mengembalikan string yang cocok dengan pola yang kita inginkan. Metode `start()` mengembalikan nilai indeks awal string yang cocok, sedangkan metode `end()` mengembalikan nilai indeks akhir dari string yang cocok. Metode `span()` mengembalikan *tuple* yang berisi nilai

indeks awal dan akhir dari string yang cocok.

```
>>> m.group()
'Python'
>>> m.start(), m.end()
(20,26)
>>> m.span()
(20,26)
```

Metode `search()` akan berhenti jika telah menemukan satu string yang cocok, bila kita ingin menemukan semua string yang cocok maka kita dapat menggunakan metode `findall()`.

Anda juga dapat menggunakan metode `finditer()` untuk mendapatkan nilai indeks awal dan akhir dari setiap string yang cocok.

```
>>> m = pat.findall(s)
>>> print m
['Python', 'Jython']
>>> iterator = pat.finditer(s)
>>> print iterator
<callable_iterator object at 0x404008ec>
>>> for m in iterator:
...     print m.span()
...
(20,26)
(31,37)
```

Modifikasi string

Modul `re` juga mendukung fungsi-fungsi modifikasi string, seperti metode `split()`, `sub()`, dan `subn()`.

Metode `split()` pada modul `re` sebenarnya hampir sama dengan metode `split()` pada objek string. Metode `split()` pada modul `re` ini berfungsi memecah/membagi bagian-bagian string setiap terjadi kecocokan dengan ekspresi reguler dan menempatkannya dalam objek list dan nilai kembalinya berupa objek list tersebut. Metode `split` memiliki dua parameter.

Parameter pertama adalah objek string yang hendak dicocokkan dan parameter kedua bersifat optional untuk menandakan maksimum dibagi berapa bagian. Nilai default untuk parameter kedua ini adalah 0. Misalkan kita mengisi parameter kedua ini dengan nilai 2, maka akan dibagi menjadi dua bagian dan sisanya akan dijadikan satu serta ikut dimasukkan dalam objek list yang terbentuk. Berikut contohnya:

```
>>> import re
>>> pat = re.compile(r'W+')
>>> pat.split("Saya sedang belajar Python")
['Saya', 'sedang', 'belajar', 'Python']
>>> pat.split("Saya sedang belajar Python",2)
['Saya', 'sedang', 'belajar Python']
>>> pat = re.compile(r'aW+')
>>> pat.split("Saya sedang belajar Python")
['S', ' sed', ' bel', ' Python']
>>> pat = re.compile(r'y')
>>> pat.split("Python, Ruby dan Tcl")
['P', 'thon, Rub', ' dan Tcl']
>>> re.split(r'y',"Python, Ruby dan Tcl")
['P', 'thon, Rub', ' dan Tcl']
```

Metode sub() akan mengganti semua substring yang cocok dengan ekspresi reguler yang diberikan dengan string baru yang kita inginkan. Metode sub() juga memiliki tiga parameter. Parameter pertama adalah string pengganti, parameter kedua berupa string yang hendak diganti dan parameter ketiga bersifat opsional untuk menandakan berapa kali akan terjadi pergantian string. Nilai default untuk

parameter ketiga ini adalah 0 menandakan bahwa semua string yang cocok dengan ekspresi reguler akan diganti. Berikut contohnya:

```
>>> pat = re.compile(r'((P|J)+\w*|Ru+\w*)')
>>> s = "Saya belajar Python, Jython dan Ruby"
>>> pat.sub("bahasa scripting",s)
'Saya belajar bahasa scripting, bahasa scripting dan bahasa scripting'
>>> pat.sub("bahasa scripting",s,2)
'Saya belajar bahasa scripting, bahasa scripting dan Ruby'
```

Metode subn() sebenarnya sama dengan metode sub(). Hanya saja nilai kembalian dari metode subn() ini berupa sebuah tuple yang mengandung dua elemen, yakni string perubahan yang dihasilkan dan banyaknya pergantian yang terjadi. Berikut contohnya:

```
>>> pat.subn("bahasa scripting",s)
('Saya belajar bahasa scripting, bahasa scripting dan bahasa scripting', 3)
```

Untuk lebih jelas mengenai penggunaan regex ini, kita akan menggunakan sebuah fungsi untuk membantu Anda mempelajari pola-pola pencocokan dengan regex ini.

```
>>> import re
>>> def re_show(pat, s):
    print re.compile(pat, re.M).sub(
        '{g<0>}',s.rstrip())
```

Fungsi di atas akan menampilkan kata/kalimat yang sesuai dengan pattern regex yang kita berikan. Sekarang saatnya bagi kita untuk menguji kemampuan regex ini.

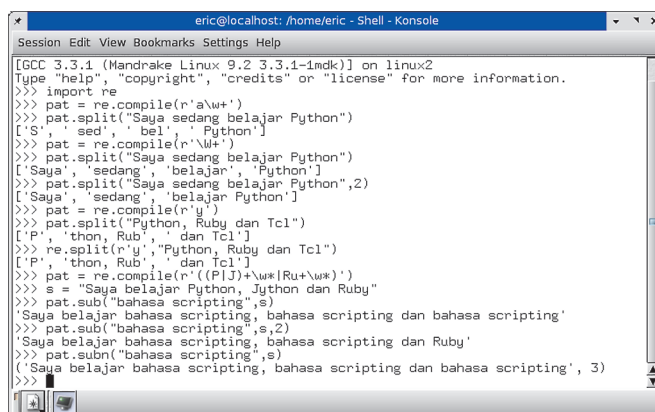
```
>>> s = "Saya sedang belajar Java dan
```

Python dengan Jython."

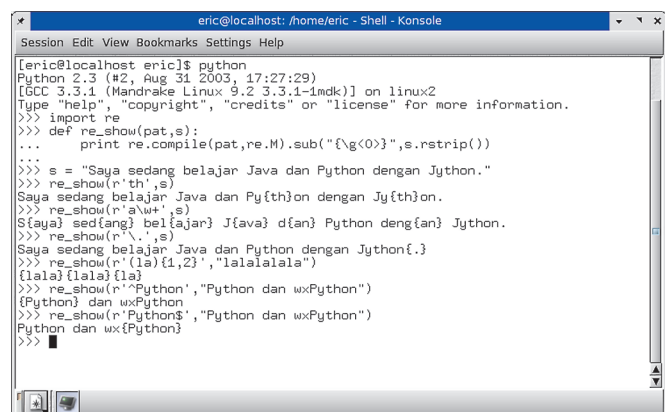
```
>>> re_show(r'th',s)
Saya sedang belajar Java dan Py{th}on
dengan Jy{th}on.
>>> re_show(r'aW+',s)
S{aya} sed{ang} bel{ajar} J{ava} d{an}
Python deng{an} Jython.
>>> re_show(r'\.',s)
Saya sedang belajar Java dan Python
dengan Jython{.}
>>> re_show(r'la){1,2}',"lalalalala")
{lala}{lala}{la}
>>> re_show(r'^Python',"Python dan
wxPython")
{Python} dan wxPython
>>> re_show(r'Python$',"Python dan
wxPython")
Python dan wx{Python}
```

Regular Expression memang menawarkan berbagai kemampuan manipulasi teks yang baik, akan tetapi juga mempunyai keterbatasan. Regular Expression mempunyai batasan di mana tidak bisa melakukan *pattern-matching* pada data yang bersarang dan bertingkat seperti pada HTML dan XML yang menggunakan sistem parsing. Adakalanya ada hal yang bisa dilakukan oleh regex, namun ekspresi regulernya akan menjadi sangat rumit. Bila hal tersebut terjadi, Anda lebih baik membuat sendiri fungsi tersebut dalam bahasa Python yang tentunya akan lebih mudah dimengerti. Dalam hal ini, tentunya Anda bisa memilih kapan kita membutuhkan regular expression. Setiap hal pasti memiliki kelebihan dan kekurangan masing-masing, bukan? ¹

Eric (eric85id@plasa.com)



▲ Gambar 3. Modifikasi string.



▲ Gambar 4. Contoh penggunaan lanjut modul re.