Flik 11: Reguljära uttryck

Reguljära uttryck

- Ett reguljärt uttryck är ett mönster för att matcha en eller flera strängar
- Ett reguljärt uttryck består av en följd av villkor som alla ska vara uppfyllda för att få en match
- Modulen *re* används i Python för reguljära uttryck

Beståndsdelarna i regexp

- Reguljära uttryck byggs upp med fyra olika tekniker:
  - Förankringar
  - Multiplikatorer
  - Atomer
  - Register

Atomer

`.` Matchar ett (1) godtyckligt tecken

`a.b` Matchar alla strängar med 'a' följt av godtyckligt tecken, följt av 'b'

`...` Matchar alla strängar med tre tecken

Vissa tecken, som exempelvis punkten ovan, har en specialbetydelse (“metatecken”). Alla tecken som inte har en specialbetydelse är atomer som matchar sig själva, som ”a” och ”b” ovan.

Förankringar – matchar noll tecken

`^` Först i strängen, efterföljande uttryck måste finnas först i strängen

`^abc` Matchar alla strängar som inleds med 'abc'

`$` Sist i strängen, uttrycket innan måste stå sist i strängen

`abc$` Matchar alla strängar som slutar med 'abc'

Reguljära uttryck - multiplikatorer

`*` Föregående uttryck kan upprepas noll eller flera gånger

`X1*$` Alla rader som slutar med ett `X` följt av noll till oändligt många `1`

`?` Föregående uttryck kan upprepas noll eller en gång

`Mr?s` Matchar `Ms` eller `Mrs`

`+` Föregående uttryck kan upprepas en eller oändligt antal gånger

`X1+$` Matchar en sträng bestående av ett `X` följt av minst en `1`

Reguljära uttryck - multiplikatorer

`{m,n}` Matchar strängar där föregående uttryck förekommer mellan *m* och *n* gånger efter varandra

`{m}` Matchar strängar där föregående uttryck förekommer exakt *m* gånger i följd

`a{8}` En sträng med åtta 'a' efter varandra

- Obs `*, +, ?, {m,n}` är "giriga" - dvs matchar så mycket som möjligt
- Använd `*?, +?, ??, {m,n}?` för att matcha minimalt

Atomer – teckenmängd

- `[tecken]` - På platsen för uttrycket får endera av de tecken som anges i *tecken* stå
- `tecken` kan anges som uppräkning (*abcdxyz*), som intervall (*0-9*) eller som kombination av uppräkningar och intervall (*axz0-9FPU*)
- Exempel:
  - `^[abc]` **-** Alla rader som inleds med 'a', 'b' eller 'c'
  - `^[_A-Za-z][_A-Za-z0-9]*$` - Matchar variabelnamn i bl.a. C
- Om första tecknet är '^' så negeras villkoret, dvs `[^abc]` matchar strängar som *inte* har 'a', 'b' eller 'c' i aktuell position
- Specialtecken - se vidare *kapitel 12 - Regular Expressions*
  - `\d` samma som `[0-9]`
  - `\D` samma som `[^0-9]`
  - `\w` matchar en bokstav, siffra eller understrykningstecken.
  - `[^\W\d_]` matchar en godtycklig bokstav (inklusive åäö...)
  - `\s` osynligt tecken, samma som `[ \n\t\r\v\f]`

Reguljära uttryck – register

- `(uttryck)` – Grupperar, dvs gör uttryck till en atom
- `|` (pipe-tecken) - matchar flera uttryck separerade av |
- Exempel:

  `Mr|Ms` matchar en sträng som innehåller `Mr` eller `Ms`

  `(ka|pe|o)lle` matchar kalle, pelle, olle
- `\` - Ta bort specialbetydelsen av metatecken:

  `\*\+` matchar en sträng som innehåller `*+`
- Parenteserna har ytterligare en effekt: matchad delsträng sparas i ett ”kom ihåg-register”
- Innehållet i registret kan plockas fram med `\1`
- `(\w)\1` matchar Kalle, Nisse och Anna, men inte Urban eller Eva.
- Varje nytt parentespar motsvarar ett nytt register, plockas fram med `\2`, `\3` osv.

Reguljära uttryck - exempel

- Ett heltal:
  ```
  [0-9]+
  ```
- Tomma rader och rader med enbart blanktecken i:
  ```
  ^ *$
  ```
- Rader med enbart versaler:
  ```
  ^[A-Z]*$
  ```
- Rader som inleds med två asterisker följt av en punkt:
  ```
  ^\*\*\.
  ```
- Rader som inte avslutas med 'a':
  ```
  [^a]$
  ```

Reguljära uttryck - match()

- *re.match()* matchar bara från början på en sträng

```
>>> import re
>>> s = 'abc'
>>> bool(re.match('a',s))
True
>>> bool(re.match('a.c',s))
True
>>> bool(re.match('b',s))
False
```

Reguljära uttryck - search()

- *re.search()* matchar godtycklig del (jmf sed, awk, perl etc)

```
>>> import re
>>> s = 'abc'
>>> bool(re.search('a',s))
True
>>> bool(re.search('.c',s))
True
>>> bool(re.search('b',s))
True
```

Reguljära uttryck - group()

- *group()*

```
>>> import re
>>> s = 'abc'
>>> re.search('b.',s).group()
'bc'
>>> re.search('b$',s).group()
Traceback ...
Attribute error...

>>> m = re.search('b$',s)
>>> if m:
>>>     print(m.group())
>>> else:
>>>     print('miss')
...
miss
```

Reguljära uttryck - gruppering

- För att extrahera ut en del av ett uttryck kan gruppering med parenteser användas.
- *group(N)* - returnerar sub-grupper, dvs uttryck inom parenteser

```
>>> import re
>>> s = 'abc-123'
>>> m = re.search(r'(\w{3})-(\d{3})',s)
>>> m.group()
'abc-123'
>>> m.group(1)
'abc'
>>> m.group(2)
'123'
```

- Alternativ med namngivna grupper - perl-syntax (?P<key>)

```
>>> m = re.search(r'(?P<alpha>\w{3})-(?P<number>\d{3})',s)
>>> m.group('alpha')
'abc'
```

- För att referera till grupper används antingen \1 för att referera till grupp 1 osv eller perl-syntaxen (?P=key)

Reguljära uttryck - gruppering

- *groups()* - returnerar alla sub-grupper som en tupel av strängar

```
>>> m.groups()
('abc', '123')
>>> m = re.search(r'(\w{3})',s)
>>> m.groups()
('abc',)
```

- `r'str'` - använd raw-strings!! `'\b'` tolkas annars t.ex. som ascii 8

Reguljära uttryck - findall() och finditer()

- findall() returnerar alla matchande mönster i en sträng

```
>>> s = 'abc-123 bcd-234'
>>> re.findall(r'\w{3}-\d{3}',s)
['abc-123', 'bcd-234']
>>> re.findall(r'(\w{3})-(\d{3})', s)
[('abc', '123'), ('bcd', '234')]
```

- finditer() returnerar en iterator som går att loopa över
  - Obs! Returnerar ett match-objekt

```
>>> for m in re.finditer(r'\w{3}-\d{3}',s):
...     print(m.group())
...
abc-123
bcd-234
```

- Observera skillnaden mellan *search/match* och *findall/finditer*

```
>>> s = 'abc-123 bcd-234'
>>> re.search(r'\w{3}-\d{3}',s).group()
'abc-123'      # dvs enbart första träffen
```

Reguljära uttryck - compile() och flaggor

- *compile()* - förkompilera reguljärt uttryck - mer effektivt om det ska användas flera gånger i ett program

```
>>> import re
>>> s = 'abc-123'
>>> pat = re.compile(r'(\w{3})-(\d{3})')
>>> m = pat.search(s)  # Även pat.findall(s) t.ex.
>>> m.group()
'abc-123'
```

- *compile(str [, flags]), search(pat, str [, flags]), match(pat, str [, flags])*
  - Exempel på flaggor som modifierar beteendet:
    - re.I => Ignorera case
    - re.X => Verbose
    - Går att kombinera med pipe-tecken: re.I|re.X

Reguljära uttryck - split() och sub()

- *split()* - gör uppdelning av sträng baserat på reguljärt uttryck

```
>>> import re
>>> s = '234   123 456    789        1234'
>>> re.split(r'\s+',s)    # Ett eller flera blanktecken
['234', '123', '456', '789', '1234']

# Alternativt med compile
>>> pat = re.compile(r'\s+')
>>> pat.split(s)
['234', '123', '456', '789', '1234']
```

- *sub()* - sök och ersätt - ersätter alla matchningar av uttryck med sträng

```
>>> pat.sub(':',s)
'234:123:456:789:1234'
>>> re.sub(r'\s+', ':', s)      # Ej kompilerat
'234:123:456:789:1234'
```

- *subn()* - returnerar istället en tupel med sträng + antal ersättningar

Övning

- **ovn1101/hittaord.py** Skapa funktionen `allaord` som tar ett filnamn som argument. Funktionen ska returnera en sträng bestående av alla ord från filen, sorterade i svensk bokstavsordning, separerade av blanktecken.
  - Med ”ord” menas här en följd av bokstäver (ej kommatecken, siffror, kolon osv.)
  - Förutsätt att filen är kodad i utf-8.
  - Versaler och gemener ska betraktas som olika.
  - Om filen inte kan läsas ska None returneras.
  - Om det inte finns några ord i filen ska en tom sträng returneras.

```
>>> print(open("fil.txt").read(), end='')
Ett och två, tre
och tre och fyra.
>>> import hittaord
>>> hittaord.allaord("fil.txt")
'Ett fyra och tre två'
>>> hittaord.allaord("finns.ej")
>>>
```