Corso di **Python**

Lezione 12

Pattern matching

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Cercare stringhe

- Se lo scopo è quello di cercare una semplice parola forse il metodo migliore non è quello di usare espressioni regolari
- Le espressioni regolari spesso non sono la risposta giusta:
  - Performance peggiori
  - Difficili da fare il debug e da mantenere
- Funzioni base delle stringhe:
  - `str.find()`
  - `str.replace()`
  - `str.startswith()`
  - `str.split()`

La libreria `re`

- Per il pattern matching Python ha la libreria **`re`**
- Il motore di pattern matching di questa libreria è scritto in linguaggio C
- I pattern sono tradotti in byte code e interpretati direttamente dal motore scritto in C, rendendo l'elaborazione più veloce
- Il pattern matching non è caratteristica inclusa nelle basi del linguaggio (come ad esempio c'è in Perl) ma può essere usata come una qualsiasi funzionalità di libreria
- Documentazione ufficiale:
  - Libreria **re**: https://docs.python.org/3.5/library/re.html
  - Tutorial: https://docs.python.org/3/howto/regex.html

Utilizzo della libreria `re`

- Le ricerche con espressioni regolari si possono fare in 2 modi:
  - Con l'uso diretto della funzioni di ricerca fornite dal modulo ***`re`***
  - Mediante compilazione di espressioni regolari come oggetti (istruzione *compile*)
- La seconda modalità è più veloce, in quanto compila negli oggetti il relativo byte code necessario ad effettuare lo specifico match

Sintassi delle espressioni regolari

Simbolo	***Matching***	***Simbolo***	***Matching***
`.`	Qualsiasi carattere	`[`*chars*`]`	Insieme di caratteri
`^`	Inizio stringa	`|`	Match alternativo
`$`	Fine stringa	`(...)`	Gruppo di matching
`*`	Ripetizione di zero o più volte	`(?:...)`	Gruppo che non cattura
`+`	Ripetizione di una o più volte	`(?P<`*name*`>...)`	Named group
`?`	Ripetizione di zero o una volta	`(?#...)`	Commento
`{m}`	Ripetizione di *m* volte	`(?!...)`	Match invertito
`{m,n}`	Ripetizione da *m* a *n* volte	`\b`	Stringa vuota se è a fine parola
`{m,n}?`	Ripetizione minimo da m a n volte	`\B`	Stringa vuota se non è a fine parola
`\`	Carattere di escape	`\d`	Cifra numerica

Funzioni di ricerca di `re`

- Ci sono 5 metodi per effettuare le ricerche:

  `re.search(pattern, string, flags=0)`
  - Cerca il pattern nella stringa, indipendentemente dalla posizione. Ritorna oggetto di matching.

  `re.match(pattern, string, flags=0)`
  - Cerca il pattern ad inizio della stringa. Ritorna un oggetto di matching.

  `re.fullmatch(pattern, string, flags=0)`
  - Introdotta dalla 3.4. Ritorna un oggetto di matching solo se la corrispondenza è sull'intera stringa.

  `re.findall(pattern, string, flags=0)`
  - Ritorna lista con stringhe o lista con coppie stringa + numero

  `re.finditer(pattern, string, flags=0)`
  - Ritorna un iteratore con elementi che sono oggetti di matching

- Ogni metodo ha il suo utilizzo specifico
- I flag specificano un diverso tipo di matching

Funzioni *search* e *match*

- Funzioni simili solo che *re.match* è limitato ai match che coincidono con l'inizio della stringa
  - Questa limitazione rende *re.match* più veloce di *re.search*
- Se non c'è match con il pattern queste funzioni ritornano *None*
- Esempi:

```
>>> print(re.search('^From', 'From Here to Eternity'))
<_sre.SRE_Match object; span=(0, 4), match='From'>
>>> print(re.match('^From', 'From Here to Eternity'))
<_sre.SRE_Match object; span=(0, 4), match='From'>


>>> print(re.search('^From', 'Reciting From Memory'))
None
>>> print(re.search('From', 'Reciting From Memory'))
<_sre.SRE_Match object; span=(9, 13), match='From'>
>>> print(re.match('From', 'Reciting From Memory'))
None
```

L'oggetto SRE_Match

- L'oggetto di match contiene tutti i dettagli del matching:

```
>>> match = re.search(pattern, string)
>>> [attr for attr in dir(match) if attr[0] != '_']
['end', 'endpos', 'expand', 'group', 'groupdict', 'groups',
'lastgroup', 'lastindex', 'pos', 're', 'regs', 'span', 'start',
'string']
```

- Può essere testato come se fosse un boolean:

```
match = re.search(pattern, string)
if match:
    process(match)
```

- Alcuni attributi sono legati alla chiamata con cui il match è stato generato:

`match.re` : regular expression utilizzata per il matching

`match.string` : stringa utilizzata per il matching

`match.pos`, `match.endpos` : parametri di inizio e fine del range di azione del motore di RE

Uso delle *raw string*

- La notazione raw string (`r"text"`) consente di semplificare espressioni regolari in cui si fa uso del backslash (`'\'`), evitando che questi caratteri debbano essere raddoppiati per effettuarne il matching:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
```

- Per effettuare il match di un carattere backslash con la raw string bastano due backslash (`r'\\'`), senza ne servirebbero quattro (`'\\\\'`):

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
```

Gruppi del match

`match.group([group1, ...])` : ritorna tutti i gruppi di matching

`match.groups(default=None)` : ritorna una tupla con tutti i gruppi (default è il valore messo ai gruppi che non partecipano al match)

`match.start([group]), match.end([group])` : indici inizio e fine del match del gruppo

`match.span([group])` : accoppiamento dei due attributi precedenti

`match.lastindex` : ultimo indice del gruppo di match o *None* se non ci sono gruppi

`match.groupdict(default=None)` : ritorna un dizionario contentente tutti i *named group* (i gruppi che sono definiti con la forma `'(?P<name>pattern)'`). Il parametro opzionale *default* è il riempimento per gruppi con nome che non rientrano nel match.

`match.lastgroup` : nome dell'ultimo named group o None se non ce ne sono

Esempi:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # >>> Il gruppo 0 è il match, che c'è sempre <<<
'Isaac Newton'
>>> m.group(1)       # Primo sottogruppo
'Isaac'
>>> m.group(2)       # Secondo sottogruppo
'Newton'
>>> m.group(1, 2)    # Specificando più indici si ottiene una tupla
('Isaac', 'Newton')

>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()       # Estrazione di tupla con tutti i gruppi
('24', '1632')

>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # In questo caso il secondo gruppo è None.
('24', None)
>>> m.groups('0')   # Per assegnare al secondo gruppo un valore numerico opportuno
('24', '0')
```

Quando usare i gruppi con nome

- I gruppi con nome sono leggermente più lenti ma sono molto utili per evitare gli errori quando si estrae il match

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)",
"Davide Brunato")
>>> m.groupdict()
{'first_name': 'Davide', 'last_name': 'Brunato'}
```

```
>>> log_line = "Sep 22 12:09:32 posta postfix/cleanup[5905]:
920FFD08069: resent-message-
id=<30140922100827.31CAD3F004@posta.sissa.it>"
>>> m = re.search(": (?P<thread>[A-Z,0-9]{9,14}): resent-
message-id=<(?P<message_id>.+)>", log_line)
>>> m.groupdict()
{'thread': '920FFD08069', 'message_id':
'30140922100827.31CAD3F004@posta.sissa.it'}
```

Funzione *findall*

- Si usa quando il pattern prevede dei match multipli

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']

>>> re.findall('\d+', '12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']

>>> str = "To: dave@example.com, maury@example.com"
>>> emails = re.findall(r'[\w\.-]+@[\w\.-]+', str)
>>> for email in emails:
...     print(email)
...
dave@example.com
maury@example.com
```

Funzione *findall* e i gruppi

- Quando il pattern prevede dei gruppi la funzione *findall* ritorna una lista di tuple:

```
>>> str = 'blah dave@example.com, blah blah
maury@example.com blah blah'
>>> tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
>>> print(tuples)
[('dave', 'example.com'), ('maury', 'example.com')]
>>> for tuple in tuples:
...     print("User: %s, Domain: %s" % tuple)
...
User: dave, Domain: example.com
User: maury, Domain: example.com
```

Funzione *findall* e i file

- Quando il pattern deve essere cercato sulle linee di un file può essere più conveniente leggere tutto il file e applicare findall una volta sola:

```
>>> from io import StringIO
>>> f = StringIO("blah dave@example.com blah\nblah
blah\nblah maury@example.com blah \nblah")
>>> emails = re.findall(r'[\w\.-]+@[\w\.-]+',
f.read())
>>> emails
['dave@example.com', 'maury@example.com']
```

- Questo approccio può avere anche un impatto significativo sulla velocità di elaborazione degli script

Funzione *finditer*

- Questa funzione si applica come *findall*, ritorna degli oggetti match ed è basata su un generatore invece che su una lista
- Rispetto a *findall*:
  - È meno onerosa perché usa un generatore per ritornare i matching
  - Permette un'elaborazione più accurata dei gruppi
  - È leggermente più lenta
- Esempio:

```
>>> text = "He was carefully disguised but captured quickly
by police."
>>> for m in re.finditer(r"\w+ly", text):
...     result = (m.start(), m.end(), m.group(0))
...     print('%02d-%02d: %s' % result)
...
07-16: carefully
40-47: quickly
```

I flag di ricerca

- I flag servono per implementare una diversa modalità di matching
- Sono definiti come costanti del modulo:

Flag	Significato
`re.ASCII`, `re.A`	Limitare l'azione di \w, \b, \s e \d al set ASCII
`re.DOTALL`, `re.S`	Il . corrisponde a tutti i caratteri compreso i newline
`re.IGNORECASE`, `re.I`	Il match è case-insensitive
`re.LOCALE`, `re.L`	Abilita un match differenziato sui caratteri speciali della lingua selezionata
`re.MULTILINE`, `re.M`	Effettua un match multi-linea, modificando il senso di ^ e $
`re.VERBOSE`, `re.X`	Abilita la modalità estesa per regex più leggibili

Uso dei flag

- I flag possono essere usati singolarmente:

```
>>> str = 'blah dave@EXAMPLE.COM, maury@example.com'
>>> re.search(r'[\w\.-]+@example\.com', str, flags=re.I)
<_sre.SRE_Match object; span=(5, 21), match='dave@EXAMPLE.COM'>
>>> re.search(r'[\w\.-]+@example\.com', str)
<_sre.SRE_Match object; span=(23, 40), match='maury@example.com'>
```

- Oppure combinati con l'operatore OR a livello di bit (' | '):

```
>>> str = 'blah blah blah\ndave@EXAMPLE.COM'
>>> re.search(r'^[\w\.-]+@example\.com', str, flags=re.I)
>>> re.search(r'^[\w\.-]+@example\.com', str, flags=re.I|re.M)
<_sre.SRE_Match object; span=(15, 31), match='dave@EXAMPLE.COM'>
```

Sostituzione di stringhe

- La libreria **re** prevede paio di funzioni per la sostituzione di stringhe:

  `re.sub(`*`pattern`*`, `*`repl`*`, `*`string`*`, `*`count=0`*`, `*`flags=0`*`)` Sostituisce le occorrenze del pattern nella stringa. Viene ritornata la nuova stringa. Il parametro *count* specifica il numero max di sostituzioni da effettuare (il default significa che rimpiazza tutte le occorrenze). Se il pattern non viene trovato allora viene restituita la stringa iniziale.

  `re.subn(`*`pattern`*`, `*`repl`*`, `*`string`*`, `*`count=0`*`, `*`flags=0`*`)` Come *re.sub* ma ritorna una coppia con elementi la nuova stringa e il numero di sostituzioni effettuate.

- Esempi:

```
>>> str = 'blue socks and red shoes'
>>> re.sub('(blue|white|red)', 'colour', str)
'colour socks and colour shoes'
>>> re.sub('(blue|white|red)', 'colour', str, count=1)
'colour socks and red shoes'
>>> re.subn('(blue|white|red)', 'colour', str)
('colour socks and colour shoes', 2)
```

- I match di lunghezza zero sono sostituiti solo se non sono adiacenti:

```
>>> re.sub('x*', '-', 'abxd')
'-a-b-d-'
```

Split di stringhe con pattern

- È disponibile una funzione per effettuare la divisione in sottostringhe mediante un pattern:

  `re.`**`split`**`(`*`pattern`*`, `*`string`*`, `*`maxsplit=0`*`, `*`flags=0`*`)` : Effettua lo divisione della stringa mediante il pattern specificato. Ritorna una lista contenente le parti. Se si specifica un parametro maxsplit > 0 questo determina il numero massimo di divisioni effettuate. Se si specificano dei gruppi espliciti per il matching questi vengono ritornati nella lista nelle posizioni corrispondenti.

- Esempi:

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
>>> re.split('\W+', '...word...word..') # Ci sono anche le stringhe vuote
['', 'word', 'word', '']
```

Split con matching vuoti

- La funzione split non include i matching vuoti:

```
>>> re.split('x*', 'axbc')
['a', 'bc']
>>> re.findall('x*', 'axbc')
['', 'x', '', '', '']
```

- Pattern che effettuano solo match vuoti non effettuano nessuna divisione sulle stringhe:

```
>>> re.split("^$", "foo\n\nbar\n", flags=re.M)
['foo\n\nbar\n']
```

- Dalla versione 3.5 tentare uno split con un pattern che prevede solo matching vuoti genera un errore:

```
>>> re.split("^$", "foo\n\nbar\n", flags=re.M)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
ValueError: split() requires a non-empty pattern match.
```

Compilazione dei pattern

- I pattern di ricerca sono compilabili in oggetti specifici (*regex*) per velocizzare ricerche multiple con lo stesso pattern:

  `re.compile(pattern, flags=0)` : Compila una stringa che specifica un pattern. Il parametro *flags* specifica il tipo di matching da effettuare. Restituisce un oggetto *regex*.

- Esempi:

```
>>> re.compile("x*", flags=0)
re.compile('x*')
>>> type(re.compile("x*", flags=0))
<class '_sre.SRE_Pattern'>
>>> pattern = re.compile('[a-z]+')
```

Funzioni dei pattern

- Gli oggetti pattern hanno le medesime di ricerca del modulo, con la differenza che non includono i parametri *pattern* e *flags:*

```
regex.split(string, maxsplit=0)
regex.sub(repl, string, count=0)
```

```
>>> pattern = re.compile("o[gh]")
>>> pattern.findall("dog")
['og']
>>> pattern = re.compile("\W+")
>>> pattern.split("Words, words, words.")
['Words', 'words', 'words', '']
```

- I metodi di ricerca dei pattern includono un paio di parametri posizionali per indicare i limiti di azione del pattern:

```
regex.search(string[, pos[, endpos]])
regex.match(string[, pos[, endpos]])
```

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match all'inizio della stringa
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # Non c'è match perché esamina a partire dalla 'o'
```

Errori nei pattern

- La libreria **`re`** prevede una sua eccezione:

  `exception re.error(`*`msg`*`, `*`pattern=None`*`, `*`pos=None`*`)` :
  - *pattern* è la stringa utilizzata per definire il pattern e *pos* è l'indice all'interno del pattern che ha generato l'errore

- Viene sollevata quando la stringa che definisce il pattern non rappresenta una espressione regolare valida:

```
>>> re.compile(r"[", flags=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.4/re.py", line 223, in compile
    return _compile(pattern, flags)
  File "/usr/lib64/python3.4/re.py", line 294, in _compile
    p = sre_compile.compile(pattern, flags)
  File "/usr/lib64/python3.4/sre_compile.py", line 568, in compile
    p = sre_parse.parse(p, flags)
  File "/usr/lib64/python3.4/sre_parse.py", line 760, in parse
    p = _parse_sub(source, pattern, 0)
  File "/usr/lib64/python3.4/sre_parse.py", line 370, in _parse_sub
    itemsappend(_parse(source, state))
  File "/usr/lib64/python3.4/sre_parse.py", line 496, in _parse
    raise error("unexpected end of regular expression")
sre_constants.error: unexpected end of regular expression
```

Altre funzioni della libreria

`re.escape(string)` : Effettua un escape di tutti i caratteri del pattern ad eccezione degli alfanumerici ASCII (e il carattere *underscore* dalla versione 3.3). Utile se si vuole effettuare un match con un letterale stringa che può includere metacaratteri di espressione regolare.

```
>>> re.escape("([abc])+")
'\\(\\[abc\\]\\)\\+'
```

```
$ python2
Python 2.7.10 (default, Sep  8 2015, 17:20:17)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> re.escape("abc_")
'abc\\_'
```

`re.purge()` : Ripulisce la cache utilizzata per il matchin delle espressioni regolari

Misurare performance - 1

- In generale per misuare le perfomance di script possiamo usare il comando Unix **time**:

```
$ time python myprogram.py


real  0m1.184s
user  0m1.171s
sys   0m0.014s
```

  - `real`: Tempo totale effettivo
  - `user`: Tempo CPU al di fuori del codice del kernel
  - `sys`: tempo CPU speso all'interno di funzioni del kernel

- Vantaggi di questo metodo:
  - Semplice da usare
  - Permette un confronto tra script fatti in linguaggi differenti
- Svantaggi:
  - Si misura l'intero processo, che magari può dipendere anche strettamente dal contesto (es. infomazioni in cache)

Misurare performance - 2

- In un sorgente si possono misurare le tempistiche creando degli oggetti Timer e facendo uso di contesti:

```
import time
class MyTimer(object):
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.secs = self.end - self.start
        self.msecs = self.secs * 1000  # millisecs

with MyTimer() as t:
    "-".join(str(n) for n in range(100))
print("Elapsed time: %s s" % t.secs)
```

Misurare performance - 3

- Per misurare direttamente la velocità di costrutti Python è meglio usare invece la libreria **timeit**
- La si può usare direttamente dalla CLI:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 24 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 20.1 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
100000 loops, best of 3: 15.1 usec per loop
```

- Oppure la si può usare nell'interprete:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.2808827519984334
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.20406439500220586
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.15624191800088738
```

API di **timeit**

- Due API basate su classe `timeit.Timer()`:

```
timeit.timeit(stmt='pass', setup='pass',
timer=<default timer>, number=1000000)
```

  - Crea una classe Timer che testa il codice definito dal parametro *stmt*.

    Il parametro *number* indica il numeri di ripetizioni per lo statement, *setup* è il codice di inizializzazione eseguito una volta sola.

    Il parametro *timer* specifica la funzione di misura del tempo da utilizzare (la funzione di default dipende dalla piattaforma e dalla versione di Python: in Unix il default è *`time.time()`* in Python 2 e *`time.perf_counter()`* per Python 3).

```
timeit.repeat(stmt='pass', setup='pass',
timer=<default timer>, repeat=3, number=1000000)
```

  - Come la precedente ma ripete completamente *timeit* per il numero indicato nel parametro *repeat*

Libreria **timeit** negli script

- La libreria *timeit* si può usare anche negli script ma solo per valutare singoli statement
- Esempio per provare:

```
import re
from timeit import timeit


s = '0' * 1000 + 'foo' + '0' * 1000
regex = re.compile(r'foo')
setup = 'from __main__ import s, regex'
print(timeit('"foo" in s', setup = setup))
print(timeit('regex.search(s).group(0)', setup = setup))
```

Performance di **re.compile**

- Un paio di run per un confronto effettivo tra le possibili modalità di ricerca e nei miglioramenti di Python 3.4 rispetto a Python 2.7:

```
$ python2.7 timing_search.py
### Performance test di ricerca stringhe ###
Operatore '<stringa> in s': 0.807714939117s
Funzione 's.find(<stringa>)': 0.914547204971s
Pattern matching con re.compile: 0.80514383316s
Pattern matching con re.search: 1.29183888435s

$ python3.4 timing_search.py
### Performance test di ricerca stringhe ###
Operatore '<stringa> in s': 0.768925852000393 2s
Funzione 's.find(<stringa>)': 0.900404608000826 6s
Pattern matching con re.compile: 0.685267993998422 8s
Pattern matching con re.search: 1.333512760000303s
```

Performance Python VS ...

- Per una comparazione partite da:
  - http://benchmarksgame.alioth.debian.org /
- Quindi:
  - Dipende da quello che dovete fare e se le performance massime sono essenziali
  - Python, Ruby, Perl hanno comunque velocità abbastanza paragonabili
  - Java è già più veloce (è un linguaggio statico ...)
  - Si può sempre migliorare (ad esempio PHP 7 è decisamente più veloce di PHP 5.6 ...)

Python 2 VS 3 speed

- Inizialmente la v. 3.0 era più lenta di Python 2.7:
  - Nuove feature non erano ottimizzate
- Successive release hanno migliorato le performance di Python 3
  - Gia con la v3.3 la velocità era migliore della v2.7:

    https://speakerdeck.com/pyconslides/python-3-dot-3-trust-me-its-better-than-python-2-dot-7-by-dr-brett-cannon

- Per fare un test completo usare la suite ufficiale di test:
  - https://hg.python.org/benchmarks

1) Scaricare e scompattare l'archivio:

   https://hg.python.org/benchmarks/archive/tip.tar.bz2

2) All'interno della directory estratta eseguire lo script perf.py, indicando come ultimi parametri la versione di Python che controlla il test e la versione da testare:

```
$ python perf.py -r -b default /usr/bin/python /usr/bin/python3
```