HOWTO sulle Espressioni Regolari

Release 0.05

A.M. Kuchling

11 febbraio 2004

amk@amk.ca

Sommario

Questo documento è un vademecum introduttivo sull'uso delle espressioni regolari in Python con il modulo `re`. Fornisce un'introduzione più semplice rispetto alla corrispondente sezione della Library Reference.

Questo documento è disponibile presso http://www.amk.ca/python/howto.

Traduzione in italiano

La presente traduzione è stata realizzata in collaborazione, ha coordinato il lavoro Ferdinando Ferranti zap[at]zonapython.it ed hanno contribuito a tradurre: Davide Benini dbenini[at]qubica.net, Diego Olerni info[at]olerni.it, Emanuele Olivetti olivetti[at]itc.it, Marco Marconi azazel_arms[at]yahoo.it, Matteo Giacomazzi matteo.giacomazzi[at]email.it, Matteo Bertini matteo[at]naufraghi.it, Mauro Morichi mauro[at]teppisti.it, Nicola Vitale nivit[at]email.it, Paolo Caldana verbal[at]teppisti.it, Paolo Massei paoafr[at]tin.it, Paolo Mossino mox79[at]gmx.it, Pierluigi Fabbris pierluigi.fabbris[at]email.it.

Indice

1 Introduzione

Il modulo `re` fu aggiunto in Python 1.5 e mette a disposizione dei modelli per le espressioni regolari simili a quelli del Perl. Versioni di Python precedenti venivano rilasciate con il modulo `regex`, che mette a disposizione dei modelli nello stile di Emacs. I modelli simili a quelli di Emacs sono leggermente meno leggibili e non mettono a disposizione altrettante funzionalità, dunque non vi è motivo di utilizzare il modulo `regex` quando si scrive del nuovo codice, tuttavia potreste incontrare del vecchio codice che ne fa uso.

Le espressioni regolari (o RE) sono essenzialmente un piccolo ed altamente specializzato linguaggio di programmazione incorporato in Python e reso disponibile attraverso il modulo `re`. Utilizzando questo piccolo linguaggio, si specificano le regole per l'insieme di possibili stringhe che di cui si vogliono trovare corrispondenze; tale insieme potrebbe contenere frasi in inglese o indirizzi di posta elettronica o comandi TeX o qualunque altra cosa vogliate. Potete quindi porre domande tipo "Questa stringa ha corrispondenza in questo modello?" o "Esiste una corrispondenza con questo modello da qualche parte nella stringa?". Potete utilizzare le RE anche per modificare una stringa o per spezzarla in diversi modi.

I modelli delle espressioni regolari sono compilati in una serie di bytecode che sono eseguiti da un motore di ricerca scritto in C. Per un uso avanzato, potrebbe essere necessario prestare una certa attenzione al modo in cui il motore esegue una data RE e scrivere la RE in un certo modo, ottenendo così del bytecode che viene eseguito più velocemente. L'ottimizzazione non viene trattata in questo documento perché richiede una buona conoscenza del comportamento interno del motore di ricerca.

Il linguaggio delle espressioni regolari è relativamente piccolo e ristretto, quindi non tutti i compiti di elaborazione di stringhe possono essere realizzati per mezzo delle espressioni regolari. Vi sono anche dei compiti che *pur potendo* essere svolti dalle espressioni regolari danno luogo a delle espressioni regolari estremamente complicate. In questi casi, fareste meglio a scrivere del codice Python per l'elaborazione; nonostante il codice Python sia più lento di una complessa espressione regolare, probabilmente sarà più comprensibile.

2 Semplici modelli

Cominceremo imparando le espressioni regolari più semplici. Siccome le espressioni regolari sono utilizzate per operare sulle stringhe, inizieremo con il caso più comune: individuare caratteri.

Per una spiegazione dettagliata della scienza informatica sottostante le espressioni regolari (automi a stati finiti deterministici e non-deterministici), potete fare riferimento a quasi qualunque libro sulla scrittura di compilatori.

2.1 Individuare la corrispondenza con i caratteri

La maggior parte delle lettere e dei caratteri semplicemente corrispondono a sé stessi. Per esempio, l'espressione regolare `test` corrisponderà esattamente alla stringa '`test`'. (Potete abilitare una modalità non sensibile alle differenze tra maiuscolo e minuscolo che farebbe corrispondere questa RE anche a '`Test`' o '`TEST`'; troverete maggiori dettagli in seguito.)

Vi sono eccezioni a questa regola; alcuni caratteri sono speciali e non corrispondono a sé stessi. Segnalano invece che alcune cose fuori dall'ordinario dovrebbero trovare corrispondenza o influenzare altre porzioni della

RE ripetendole. Gran parte di questo documento è dedicato alla discussione di vari metacaratteri ed al loro compito.

Questa è una lista completa dei metacaratteri; il loro significato verrà discusso nel resto di questo HOWTO.

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

I primi metacaratteri che guarderemo sono '`[`' e '`]`'. Sono utilizzati per specificare classi di caratteri che sono un insieme di caratteri di cui si desidera ottenere una corrispondenza. I caratteri possono essere elencati individualmente o tramite un intervallo di caratteri che può essere indicato tramite due caratteri e separati con un '`-`'. Per esempio ⌈`[abc]`⌋ troverà corrispondenza con ognuno dei caratteri '`a`', '`b`' o '`c`'; il medesimo comportamento per ⌈`[a-c]`⌋, che utilizza un intervallo per esprimere il medesimo insieme di caratteri. Se volete trovare le corrispondenze solamente delle lettere minuscole la vostra RE potrebbe essere ⌈`[a-z]`⌋.

I metacaratteri non sono attivi nelle classi. Per esempio, ⌈`[akm$]`⌋ troverà corrispondenza con ognuno dei caratteri '`a`', '`k`', '`m`' o '`$`'; '`$`' è solitamente un metacarattere ma, in una classe di caratteri, viene privato della sua natura speciale.

Potete far corrispondere i caratteri non presenti in un intervallo creando il *complemento* dell'insieme. Questo si indica includendo un '`^`' come primo carattere della classe; '`^`' in ogni altro luogo corrisponderà semplicemente al carattere '`^`'. Per esempio `[^5]` darà corrispondenza ad ogni carattere eccetto il '`5`'.

Forse il metacarattere più importante è il backslash '`\`'. Come nelle stringhe costanti (NdT: string literals) di Python il backslash può essere seguito da diversi caratteri per segnalare varie sequenze speciali. Viene utilizzato anche per effettuare la protezione di tutti i metacaratteri in modo da poterne trovare corrispondenza nei modelli; per esempio, se dovete trovare la corrispondenza con una '`[`' o una '`\`' potete farle precedere da un backslash per rimuovere il loro significato speciale: ⌈`\[`⌋ o ⌈`\\`⌋.

Alcune delle sequenze speciali che iniziano con '`\`' rappresentano degli insiemi predefiniti di caratteri che sono spesso utili, come l'insieme delle cifre, l'insieme delle lettere o l'insieme di tutto ciò che non è spazio vuoto. Sono disponibili le seguenti sequenze predefinite:

`\d` Corrisponde ad ogni cifra decimale; equivale alla classe ⌈`[0-9]`⌋.

`\D` Corrisponde ad ogni carattere che non sia una cifra; è equivalente alla classe `[^0-9]`.

`\s` Corrisponde ad ogni carattere di spaziatura; è equivalente alla classe ⌈`[ \t\n\r\f\v]`⌋.

`\S` Corrisponde ad ogni carattere che non sia uno spazio vuoto; è equivalente alla classe `[^ \t\n\r\f\v]`.

`\w` Corrisponde ad ogni carattere alfanumerico; è equivalente alla classe ⌈`[a-zA-Z0-9_]`⌋.

`\W` Corrisponde ad ogni carattere che non sia alfanumerico; è equivalente alla classe `[^a-zA-Z0-9_]`.

Tali sequenze possono essere incluse in una classe di caratteri. Per esempio ⌈`[\s,.]`⌋ è una classe di caratteri che corrisponde ad ogni carattere di spazio o ad una '`,`' o ad un '`.`'.

L'ultimo metacarattere in questa sezione è il ⌈`.`⌋. Corrisponde a qualunque carattere salvo il carattere di a fine riga ed esiste una modalità (`re.DOTALL`) in cui corrisponde anche al carattere di fine riga. '`.`' viene spesso usato quando volete trovare corrispondenza con "ogni carattere".

2.2 Cose che si ripetono

Il fatto di essere capaci di individuare di caratteri variabili è la prima cosa che le espressioni regolari possono fare e che non era possibile con i metodi disponibili per le stringhe. Però, se questa fosse l'unica nuova risorsa offerta dalle RE, non sarebbe un gran guadagno. Un'altra caratteristica importante è quella di poter specificare parti della RE che devono essere ripetute un certo numero di volte.

Il primo metacarattere che vedremo per esprimere ripetizioni è ⌈`*`⌋. L'asterisco ⌈`*`⌋ non cerca il carattere '`*`'; serve invece a specificare che il carattere che lo precede può presentarsi zero o più volte e non esclusivamente una sola volta.

Per esempio, ⌜`ca*t`⌟ troverà corrispondenza '`ct`' (0 caratteri '`a`'), '`cat`' (1 '`a`'), '`caaat`' (3 caratteri '`a`') e così via. Il motore delle RE ha alcune limitazioni derivanti dalle dimensioni dei tipi `int` in C, questo vi impedirà di trovare corrispondenze con stringhe con più di 2 miliardi di '`a`'; ma probabilmente non avete abbastanza memoria per costruire una stringa tanto grande, quindi non dovreste mai raggiungere questo limite.

Ripetizioni come ⌜`*`⌟ sono *golose*; quando si cercano ripetizioni in una RE il motore cerca quella più lunga possibile. Se poi alcune parti del modello non corrispondono, il motore torna indietro e riprova con meno ripetizioni.

Un esempio analizzato passo passo renderà le cose più chiare. Consideriamo l'espressione ⌜`a[bcd]*b`⌟. Questa espressione cerca una stringa che inizia con la lettera '`a`', prosegue con zero o più lettere della classe `[bcd]` e termina con '`b`'. Immaginiamo adesso di provare questa RE con la stringa '`abcbd`'.

Passo	Modello	Spiegazione
1	`a`	La ⌜`a`⌟ nella RE corrisponde.
2	`abcbd`	Il motore cerca ⌜`[bcd]*`⌟ e va avanti il più possibile, cioè fino alla fine della stringa.
3	*Fallimento*	Il motore cerca la ⌜`b`⌟, ma la posizione corrente è alla fine della stringa e quindi fallisce.
4	`abcb`	Torna indietro, adesso ⌜`[bcd]*`⌟ corrisponde ad un carattere in meno.
5	*Fallimento*	Cerca di nuovo ⌜`b`⌟, ma nella posizione corrente trova il carattere '`d`'.
6	`abc`	Torna indietro, in questo momento ⌜`[bcd]*`⌟ corrisponde solamente con '`bc`'.
6	`abcb`	Cerca di nuovo ⌜`b`⌟. Ma stavolta il carattere nella posizione corrente è '`b`', la ricerca ha successo.

Il lavoro della RE è terminato e corrisponde con '`abcb`'. Questo mostra come opera l'algoritmo di ricerca, che prima va avanti il più possibile e poi, se non trova la giusta corrispondenza, torna progressivamente indietro riprovando più e più volte. Sarebbe tornato indietro fino a cercare zero corrispondenze con ⌜`[bcd]*`⌟ ed al passo successivo avrebbe fallito, concludendo che la stringa non corrisponde per niente alla RE.

Un altro metacarattere per le ripetizioni è ⌜`+`⌟, ed evidenzia una o più occorrenze. È necessario fare attenzione alla differenza tra ⌜`*`⌟ e ⌜`+`⌟; ⌜`*`⌟ cerca zero o più occorrenze, quindi tutto quello che può essere ripetuto può anche essere del tutto assente, mentre con ⌜`+`⌟ è necessaria almeno *una* occorrenza. Per usare un esempio simile al precedente, ⌜`ca+t`⌟ troverà '`cat`' (1 '`a`'), '`caaat`' (3 '`a`'), ma non troverà '`ct`'.

Esistono altri due qualificatori di ripetizioni. Il carattere punto interrogativo, ⌜`?`⌟, cerca una o zero ripetizioni; potete pensarlo come un marcatore per qualcosa che è facoltativo. Ad esempio, ⌜`home-?brew`⌟ troverà sia '`homebrew`' che '`home-brew`'.

Il qualificatore di ripetizioni più complicato è ⌜`{m,n}`⌟, dove *m* e *n* sono numeri decimali interi. Significa che devono esserci almeno *m* ripetizioni e non più di *n*. Ad esempio ⌜`a/{1,3}b`⌟ corrisponderà con '`a/b`', '`a//b`' e '`a///b`'. Però non troverà '`ab`', che non ha barre, o '`a////b`', che ne ha quattro.

È possibile omettere *m* o *n*; nel caso viene assunto un valore ragionevole al posto di quello mancante. L'omissione di *m* viene interpretata come un limite inferiore posto a 0, mentre l'omissione di *n* mette il limite superiore a infinito, o meglio, al limite di 2 miliardi menzionato in precedenza, che possiamo considerare infinito.

Lettori con un'inclinazione riduzionista potrebbero notare che i primi tre qualificatori potrebbero essere espressi usando questa notazione. ⌜`{0,}`⌟ equivale a ⌜`*`⌟, ⌜`{1,}`⌟ equivale a ⌜`+`⌟ e ⌜`{0,1}`⌟ equivale a ⌜`?`⌟. Però è meglio usare ⌜`*`⌟, ⌜`+`⌟ e ⌜`?`⌟ quando possibile, semplicemente perché sono più compatti e più facili da leggere.

3 Utilizzo delle espressioni regolari

Ora che sono state introdotte alcune semplici espressioni regolari, come si utilizzano in pratica in Python? Il modulo `re` fornisce una interfaccia verso il motore di gestione delle espressioni regolari, consentendo la compilazione delle RE in oggetti ed il loro successivo utilizzo nella ricerca delle corrispondenze.

3.1 Compilare le Espressioni Regolari

Le espressioni regolari vengono compilate in istanze `RegexObject`, che contengono i metodi per svariate operazioni, come la ricerca di un determinato modello di corrispondenza all'interno di una stringa o la sua sostituzione.

```
>>> import re
>>> p = re.compile('ab*')
>>> print p
<re.RegexObject instance at 80b4150>
```

`re.compile()` accetta anche un argomento *facoltativo*, utilizzato per abilitare varie funzionalità speciali e variazioni di sintassi. In seguito verranno illustrate le configurazioni disponibili, ma partiamo ora con un esempio:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

La RE è passata al metodo `re.compile()` come stringa. Le RE vengono trattate come stringhe perché non fanno parte del nucleo di sviluppo del linguaggio Python e non è stata sviluppata alcun tipo di sintassi particolare per esprimerle. (Esistono applicazioni che non necessitano dell'utilizzo delle RE, quindi non c'è la necessità di appesantire le specifiche del linguaggio includendole). Invece, il modulo `re` è semplicemente un modulo di estensione, scritto in C, incluso in Python, proprio come il modulo `socket` o `zlib`.

Considerare le RE come stringhe permette di mantenere il linguaggio Python più semplice, ma presenta uno svantaggio, che sarà l'argomento del prossimo paragrafo.

3.2 Il problema del backslash

Come spiegato in precedenza, le espressioni regolari fanno uso del carattere backslash ('`\`') per indicare particolari forme o per consentire di usare i caratteri speciali privati del loro proprio significato speciale. Questo crea un conflitto con l'uso in Python dello stesso carattere per il medesimo impiego nelle stringhe costanti.

Supponiamo che vogliate scrivere una RE che indichi le corrispondenze della stringa '`\section`', tipico comando che si può trovare in un file LaTeX. Per capire che cosa scrivere nel codice del programma, si parte dalla stringa di cui si cerca la corrispondenza. Poi dovrete proteggere ogni backslash, ed ogni metacarattere, anteponendovi un backslash, il risultato sarà la stringa '`\\section`'. La stringa da passare a `re.compile()` sarà quindi `\\section`. Però, per esprimere questa come una stringa in Python, entrambi i backslash devono essere preceduti da *ancora* un altro backslash.

Carattere	Scenario
`\section`	Stringa di testo di cui cercare la corrispondenza
`\\section`	Protezione del backslash per `re.compile`
`\\\\section`	Protezione del backslash per una stringa costante

In breve, per trovare la corrispondenza del carattere letterale di backslash, dovrete scrivere la stringa RE `'\\\\'`, perché la RE deve essere '`\\`' e ad ogni backslash deve essere espresso come '`\\`' all'interno di una corretta stringa costante di Python. Nelle RE che fanno largo uso del carattere backslash, questo porta a ripetere molti backslash e rende la stringa risultante difficile da comprendere.

La soluzione è usare la notazione di Python per le stringhe, raw, ed utilizzarla per le espressioni regolari; i backslash non sono gestiti in nessun modo speciale se si trovano in una stringa costante che ha il prefisso '`r`', quindi `r\n` indica una stringa di 2 caratteri, contenente '`\`' e '`n`', mentre `\n` indica una stringa con il solo carattere di fine riga. Spesso le espressioni regolari sono espresse in Python adottando la notazione raw delle stringhe.

Stringa ordinaria	Stringa raw
`ab*`	`rab*`
`\\\\section`	`r\\section`
`\\w+\\s+\\1`	`r\w+\s+\1`

3.3 Eseguire confronti

Dopo aver ottenuto un oggetto rappresentante un'espressione regolare compilata, cosa ve ne fate? L'istanza `RegexObject` possiede alcuni metodi ed attributi. Ci occuperemo solo del più significativi; consultate la libreria di riferimento per un elenco dettagliato.

Metodi/Attributi	Scopo
`match()`	Determina se la RE corrisponde all'inizio della stringa.
`search()`	ricerca all'interno di una stringa, trovando tutte le posizioni corrispondenti alla RE.
`findall()`	Trova tutte le sottostringhe corrispondenti alla RE, e le restituisce in una lista.
`finditer()`	Trova tutte le sottostringhe corrispondenti alla RE, e le restituisce in un iteratore.

`match()` e `search()` restituiscono `None` se nessuna corrispondenza viene trovata. Se hanno successo, viene restituita un'istanza `MatchObject`, contenente informazioni riguardo alla corrispondenza: dove inizia e dove finisce, la sottostringa a cui corrisponde e altro.

Potete imparare molto al riguardo sperimentando interattivamente con il modulo `re`. Se avete Tkinter disponibile, potete anche dare un'occhiata a 'Tools/scripts/redemo.py', un programma dimostrativo incluso nella distribuzione Python. Vi permette di inserire RE e stringhe, e vi dice se la RE corrisponde o meno. 'redemo.py' può essere abbastanza utile quando provate a debuggare una RE complicata. Kodos di Phil Schwartz è anche uno strumento interattivo per sviluppare e testare modelli RE. Questo HOWTO utilizzerà l'interprete standard Python per i suoi esempi.

Prima cosa, lanciate l'interprete Python, importate il modulo `re` e compilate una RE:

```
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 80c3c28>
```

Adesso, potete provare a far corrispondere varie stringhe alla RE `[a-z]+`. Una stringa vuota non dovrebbe corrispondere, visto che `+` significa una o più ripetizioni. `match()` dovrebbe restituire `None` in questo caso, che non permetterà di stampare nulla all'interprete. Potete stampare esplicitamente il risultato di `match()` per renderlo più chiaro.

```
>>> p.match("")
>>> print p.match("")
None
```

Adesso, proviamolo su una stringa che dovrebbe corrispondere, come '`tempo`'. In questo caso, `match()` restituirà `MatchObject`, per cui dovreste salvare il risultato in una variabile per un uso futuro.

```
>>> m = p.match('tempo')
>>> print m
<_sre.SRE_Match object at 80c4f68>
```

Ora potete interrogare `MatchObject` per avere informazioni riguardo la stringa corrispondente. L'istanza `MatchObject` possiede inoltre alcuni metodi ed attributi; i più importanti sono:

Metodo/Attributo	Scopo
`group()`	restituisce la stringa corrispondente alla RE
`start()`	restituisce la posizione iniziale della corrispondenza
`end()`	restituisce la posizione finale della corrispondenza
`span()`	restituisce una tupla contenente la (start, end) posizione della corrispondenza

Provando questi metodi chiarirete presto il loro significato:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` restituisce la sottostringa che corrispondeva alla RE. `start()` ed `end()` restituiscono l'indice iniziale e finale della corrispondenza. `span()` restituisce entrambi gli indici in una singola tupla. Visto che il metodo `match` controlla solo l'inizio di una stringa, `start()` sarà sempre zero. Comunque, il metodo `search` dell'istanza `RegexObject` ricerca all'interno della stringa, quindi la corrispondenza non può iniziare da zero in quel caso.

```
>>> print p.match('::: messaggio')
None
>>> m = p.search('::: messaggio') ; print m
<re.MatchObject instance at 80c9650>
>>> m.group()
'messaggio'
>>> m.span()
(4, 11)
```

Nei programmi reali, lo stile più comune è salvare `MatchObject` in una variabile, e controllare se sia `None`. Solitamente si presenta così:

```
p = re.compile( ... )
m = p.match( 'la stringa va qui' )
if m:
    print 'Corrispondenza trovata: ', m.group()
else:
    print 'Nessuna corrispondenza'
```

Due metodi `RegexObject` restituiscono tutte le corrispondenze per un campione. `findall()` restituisce una lista di stringhe corrispondenti.

```
>>> p = re.compile('\d+')
>>> p.findall('12 batteristi, 11 trombettisti, 10 saltimbanchi')
['12', '11', '10']
```

`findall()` deve creare l'intera lista prima che possa essere restituita come risultato. In Python 2.2, è disponibile anche il metodo `finditer()`, che restituisce una sequenza di istanze `MatchObject` sottoforma di iteratore.

```
>>> iterator = p.finditer('12 batteristi, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x401833ac>
>>> for match in iterator:
...     print match.span()
...
(0, 2)
(22, 24)
(29, 31)
```

3.4 Funzioni a livello di modulo

Non dovete produrre un `RegexObject` e chiamare i suoi metodi; il modulo `re` fornisce anche funzioni di alto livello chiamate `match()`, `search()`, `sub()` e così via. Queste funzioni prendono gli stessi argomenti del corrispondente metodo `RegexObject`, con la stringa RE aggiunta come primo argomento e restituiscono ancora come risultato sia `None` che un'istanza `MatchObject`.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.MatchObject instance at 80c5978>
```

Sotto la superficie, queste funzioni producono semplicemente per voi un `RegexObject` e richiamano su di esso il metodo appropriato. Inoltre, memorizzano l'oggetto compilato in una cache, in modo da rendere più veloci future chiamate che utilizzino la stessa RE.

Dovreste usare funzioni a livello di modulo oppure dovreste prendere la `RegexObject` e chiamare da soli i suoi metodi? La scelta qui dipende sia da quanto frequentemente verrà usata la RE, sia dal personale stile di programmazione. Se una RE verrà utilizzata solo in un punto del codice, allora saranno probabilmente più convenienti le funzioni del modulo. Se un programma contiene molte espressioni regolari o le riutilizza in diverse occasioni, potrebbe diventare più conveniente mantenere tutte le definizioni in un posto solo, nella sezione di codice che compila tutte le RE in anticipo. Per vedere un esempio dalla libreria standard, ecco un estratto dalla 'xmllib.py':

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

In genere è preferibile lavorare con oggetti compilati, perfino se si usano una sola volta, ma altre persone saranno di sicuro più puriste, riguardo questo argomento.

3.5 Opzioni di compilazione

Le opzioni di compilazione modificano alcuni aspetti di come le espressioni regolari lavorano. Le opzioni sono disponibili, nel modulo `re` sotto due nomi, un nome lungo come `IGNORECASE`, e uno corto (una lettera) come `I`. (Se avete familiarità con i modificatori di modello del Perl, i caratteri speciali usano le stesse lettere; La forma corta ad una lettera di `re.VERBOSE` è `re.X`, per esempio.) Opzioni multiple possono essere specificate per ogni singolo bit; `re.I | re.M` impostano entrambe le opzioni `I` e `M`, per esempio.

Qui c'è una tavola delle opzioni disponibili, seguite da una più dettagliata spiegazione di ognuna di esse.

Opzione	Significato
`DOTALL`, `S`	Rende `.` cerca la corrispondenza di ogni carattere, includendo i fine riga
`IGNORECASE`, `I`	Cerca la corrispondenza dei caratteri senza fare distinzione tra maiuscolo e minuscolo
`LOCALE`, `L`	Cerca la corrispondenza usando le impostazioni locali
`MULTILINE`, `M`	Cerca la corrispondenza nelle linee attraverso `^` e `$`
`VERBOSE`, `X`	Abilita la prolissità delle RE, potrebbe essere organizzato in modo più pulito e comprensibile.

`I`
`IGNORECASE`

Effettua una ricerca della corrispondenza senza fare distinzione tra maiuscolo e minuscolo; in classi di caratteri e stringhe costanti si ricercheranno lettere ignorando se maiuscole o minuscole. Per esempio, `[A-Z]` ricercherà anche caratteri minuscoli, e `Spam` ricercherà '`Spam`', '`spam`', o '`spAM`'. Questo tipo di ricerca della corrispondenza non prende la localizzazione dell'account; la prenderà se verrà impostata anche l'opzione `LOCALE`.

L
LOCALE

Crea ⌈\w⌋, ⌈\W⌋, ⌈\b⌋, e ⌈\B⌋, in funzione della corrente localizzazione.

Le localizzazioni sono delle caratteristiche della libreria del C intese ad aiutare nella scrittura di programmi che tengono conto delle differenze di linguaggio. Per esempio, se state processando del testo francese, vorreste essere capaci di scrivere ⌈\w+⌋ per cercare la corrispondenza delle parole, ma ⌈\w⌋ troverà solo la classe di caratteri ⌈[A-Za-z]⌋; non troverà 'é' o 'ç'. Se il vostro sistema è propriamente configurato e la localizzazione francese è selezionata, certe funzioni C diranno al programma che 'é' dovrebbe essere considerata una lettera. Impostando l'opzione LOCALE quando compilate una espressione regolare verrà restituito un oggetto risultante compilato che usa queste funzioni C per ⌈\w⌋; questo è più lento, ma abilita anche ⌈\w+⌋ per cercare parole francesi come vi aspettereste.

M
MULTILINE

(⌈^⌋ e ⌈$⌋ non sono stati ancora spiegati; saranno introdotti nella sezione 4.1.)

Solitamente ⌈^⌋ cerca solo all'inizio della stringa, e ⌈$⌋ cerca solo alla fine della stringa ed immediatamente prima del fine riga (se c'è) alla fine della stringa. Quando questa opzione è specificata, ⌈^⌋ cerca all'inizio della stringa e all'inizio di ogni riga dentro la stringa, seguendo immediatamente ogni fine riga. In modo simile il metacarattere ⌈$⌋ cerca sia alla fine della stringa che alla fine di ogni riga (immediatamente precedente ogni fine riga).

S
DOTALL

Il carattere speciale '.' cerca ogni carattere, incluso il fine riga, senza questa opzione, '.' cercherà tutto *eccetto* il fine riga.

X
VERBOSE

Questa opzione permette di scrivere espressioni regolari più leggibili per garantire una maggiore flessibilità nel modo di formattarle.

Quando questa opzione è stata specificata, gli spazi vuoti in una stringa RE sono ignorati, eccetto quando gli spazi sono una classe di caratteri o sono preceduti da un backslash non protetto. Questo permette di organizzare ed indentare le RE in modo più chiaro. Inoltre vi permette di inserire commenti all'interno di una RE che saranno ignorati dal motore di ricerca delle corrispondenze; i commenti sono contrassegnati dal simbolo '#' che non è in nessuna classe di caratteri, né tantomeno deve essere protetto da un backslash.

Per esempio, qui c'è una RE che usa re.VERBOSE; vedete com'è molto più semplice da leggere?

```
charref = re.compile(r"""
 &[#]      # Un riferimento ad un'entità numerica
 (
   [0-9]+[^0-9]      # Forma decimale
   | 0[0-7]+[^0-7]   # Forma ottale
   | x[0-9a-fA-F]+[^0-9a-fA-F] # Forma esadecimale
 )
""", re.VERBOSE)
```

Senza l'impostazione prolissa, la RE sarebbe apparsa così:

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

Nell'esempio precedente, la concatenazione automatica di stringhe costanti in Python è stata usata per spezzare la RE in piccoli pezzi, ma è molto più difficile da capire che nella versione re.VERBOSE.

4 Ulteriori potenzialità dei modelli

Fin qui abbiamo esplorato solo una parte delle caratteristiche delle espressioni regolari. In questa sezione vedremo alcuni nuovi metacaratteri e come usare i gruppi per recuperare porzioni del testo che corrispondono alla ricerca.

4.1 Ancora sui metacaratteri

Ci sono alcuni metacaratteri che non abbiamo ancora discusso. La maggior parte di essi sarà illustrata in questa sezione.

Alcuni dei metacaratteri che devono ancora essere discussi sono *asserzioni di lunghezza zero*. Queste non provocano un avanzamento del motore attraverso la stringa; al contrario, non consumano affatto alcun carattere, semplicemente falliscono o hanno successo. Ad esempio, `\b` è una asserzione che indica che la posizione corrente è situata al limite di una parola; la posizione non viene modificata da `\b`. Questo significa che le asserzioni a lunghezza zero non dovrebbero mai essere ripetute, perché se corrispondono una prima volta per una determinata posizione, ovviamente possono farlo per un numero infinito di volte.

`|` Alternanza, o anche l'operatore "or". Se A e B sono espressioni regolari, `A|B` troverà corrispondenza in qualunque stringa che corrisponda ad '`A`' oppure a '`B`'. `|` ha un ordine di precedenza molto basso per permettere che funzioni ragionevolmente quando si alternano stringhe multi-carattere. `Crow|Servo` corrisponderà sia a '`Crow`' che a '`Servo`', non a '`Cro`', ad una '`w`' o ad una '`S`', né ad '`ervo`'.

Per una corrispondenza letterale con '`|`', si usi `\|`, o lo si racchiuda in una classe di caratteri, come `[|]`.

`^` Trova la corrispondenza all'inizio delle righe. A meno che non sia stata impostata l'opzione `MULTILINE`, questo metacarattere troverà la corrispondenza solo all'inizio della stringa. In modalità `MULTILINE` troverà la corrispondenza anche immediatamente dopo ogni fine riga all'interno della stringa.

Ad esempio, se si desidera trovare la corrispondenza della parola '`From`' solo all'inizio della riga, la RE da usare è `^From`.

```
>>> print re.search("^Da", "Da qui all'inifinto")
<re.MatchObject instance at 80c1520>
>>> print re.search('^Da', 'Recitare Da Copione')
None
```

`$` Trova la corrispondenza alla fine di una riga, definita sia come la fine della stringa, sia come ogni posizione seguita da un codice di controllo di fine riga.

```
>>> print re.search('}$', '{block}')
<re.MatchObject instance at 80adfa8>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<re.MatchObject instance at 80adfa8>
```

Per una corrispondenza letterale con '`$`', usate `\$`, o racchiudetelo in una classe di caratteri, come `[$]`.

`\A` Trova la corrispondenza solo all'inizio della stringa. Quando non si è in modalità `MULTILINE` `\A` e `^` sono a tutti gli effetti la stessa cosa. In modalità `MULTILINE` invece essi sono differenti; `\A` continua a trovare la corrispondenza solo all'inizio della stringa, mentre `^` può corrispondere a qualunque posizione all'interno della stringa che segua un carattere di fine riga.

`\Z` Trova la corrispondenza solo alla fine della stringa.

`\b` Limite di una parola. Questa è una asserzione a lunghezza zero che trova la corrispondenza solo all'inizio o alla fine di una parola. Una parola è definita come una sequenza di caratteri alfanumerici, perciò la fine di una parola è indicata da un carattere spazio o da un carattere non alfanumerico.

Il seguente esempio trova ‘`class`’ solo quando questa è una parola completa; non troverà corrispondenza quando essa è contenuta all’interno di un’altra parola.

```
>>> p = re.compile(r'\bclassi\b')
>>> print p.search('niente classi e tutte')
<re.MatchObject instance at 80c8f28>
>>> print p.search('algoritmo declassificato')
None
>>> print p.search('molte sottoclassi')
None
```

Ci sono due punti critici che è necessario ricordare usando questa sequenza speciale. Per prima cosa, in Python questo è il peggiore conflitto tra stringhe costanti ed espressioni regolari. Nelle stringhe costanti di Python, ‘`\b`’ è il carattere backspace, valore ASCII 8. Se non usaste stringhe di tipo raw, allora Python convertirà ‘`\b`’ in un backspace, e la vostra RE non troverà la corrispondenza come vi aspettereste. L’esempio seguente sembra lo stesso della nostra precedente RE, ma omette la ‘`r`’ all’inizio della stringa della RE.

```
>>> p = re.compile('\bclassi\b')
>>> print p.search('niente classi e tutte')
None
>>> print p.search('\b' + 'classi' + '\b')
<re.MatchObject instance at 80c3ee0>
```

In secondo luogo, all’interno di una classe di caratteri, dove non è utilizzabile questa asserzione, ⌈`\b`⌋ rappresenta il carattere backspace, per compatibilità con le stringhe costanti di Python.

⌈`\B`⌋ Un’altra asserzione a lunghezza zero, questa è l’opposto di ⌈`\b`⌋, trova la corrispondenza solo quando la posizione corrente non è al limite di una parola.

4.2 Riunire i gruppi

Frequentemente avrete bisogno di ottenere più informazioni rispetto al solo sapere se la RE corrisponde o meno. Le espressioni regolari sono spesso usate per suddividere stringhe scrivendo una RE divisa in diversi sottogruppi che corrispondono a differenti componenti. Per esempio, una riga di intestazione RFC-822 è divisa in un nome ed un valore separati da un ‘`:`’. Questa riga può essere analizzata scrivendo una espressione regolare che corrisponde con un’intera intestazione, un gruppo ne riconosce la parte del nome ed un altro gruppo ne verifica il valore.

I gruppi sono identificati con questi metacaratteri ‘`(`’, ‘`)`’. ‘`(`’ e ‘`)`’ rappresentano più del significato che hanno nelle espressioni matematiche. Riuniscono insieme le espressioni contenute al loro interno. Per esempio, potete ripetere il contenuto di un gruppo con un qualificatore di ripetizione, come ⌈`*`⌋, ⌈`+`⌋, ⌈`?`⌋, o ⌈`{m,n}`⌋. Per esempio, ⌈`(ab)*`⌋ verificherà zero o più ripetizioni di ‘`ab`’.

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```

I gruppi indicati con ‘`(`’, ‘`)`’ catturano anche l’inizio e la fine dell’indice del testo che verificano; Questi possono essere recuperati passando un argomento a `group()`, `start()`, `end()` e `span()`. I gruppi vengono numerati partendo da 0. Il gruppo 0 è sempre presente; lo è nell’intera RE, così tutti i metodi `MatchObject` hanno il gruppo 0 come loro argomento predefinito. Successivamente vedremo come esprimere gruppi che non catturino la sezione di testo corrispondente.

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

I sottogruppi sono numerati da sinistra a destra, da 1 in poi. I gruppi possono essere annidati; per determinarne il numero, semplicemente contate il numero di parentesi aperte, procedendo da sinistra a destra.

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

Al metodo `group()` possono essere passati più valori di gruppo per volta, in questo caso restituirà una tupla contenente il corrispondente valore dei gruppi indicati.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

Il metodo `groups()` restituisce una tupla contenente la stringa di tutti i sottogruppi, da 1 in poi, quanti essi siano.

```
>>> m.groups()
('abc', 'b')
```

I parametri in un modello permettono di specificare che il contenuto di un precedente gruppo catturante deve anche essere trovato nella posizione corrente all'interno della stringa. Per esempio, `\1` avrà successo se l'esatto contenuto del gruppo 1 si troverà nella posizione corrente, altrimenti fallirà. Ricordatevi che le stringhe costanti di Python includono il backslash seguito da numeri per consentire l'inclusione di caratteri arbitrari, pertanto accertatevi di usare una stringa semplice quando includerete parametri in una RE.

Per esempio, la seguente RE intercetta parole doppie all'interno di una stringa.

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Parigi in in estate').group()
'in in'
```

Parametri come questi non sono poi così utili per cercare semplicemente attraverso una stringa – ci sono pochi formati di testo che ripetono le parole come in questo modo – ma presto scoprirete che sono *molto* più utili per effettuare delle sostituzioni nelle stringhe.

4.3 Gruppi nominati e non-catturanti

Le RE elaborate possono utilizzare molti gruppi, sia per catturare sottostringhe di interesse, che per raggruppare e strutturare la RE stessa. Nelle RE complesse, diventa difficile tenere traccia del numero dei gruppi. Ci sono due funzionalità che possono aiutarvi con questo problema. Entrambe usano una sintassi comune per le espressioni regolari, perciò ora vi daremo un'occhiata.

Perl 5 aggiunge alcune ulteriori caratteristiche per le espressioni regolari standard, ed il modulo `re` di Python ne supporta parecchie. Sarebbe stato difficile scegliere nuovi metacaratteri a singola battitura o nuove sequenze speciali inizianti con '\' per rappresentare la nuova funzionalità senza rendere le espressioni regolari del Perl confusamente differenti dallo standard delle RE. Se sceglievate '&' come nuovo metacarattere, per esempio, le vecchie espressioni avrebbero considerato che '&' fosse un carattere regolare e non lo avrebbero protetto scrivendo ⌜`\&`⌟ o ⌜`[&]`⌟.

La soluzione scelta dagli sviluppatori Perl fu di usare ⌜`(?...)`⌟ come sintassi di estensione. '?' immediatamente dopo una parentesi era un'errore di sintassi perché il '?' non avrebbe avuto nulla da ripetere, perciò questo non introdusse nessun problema di compatibilità. I caratteri immediatamente dopo il '?' indicano quale estensione viene utilizzata, quindi ⌜`(?=foo)`⌟ è una cosa (un'asserzione lookahead positiva) mentre ⌜`(?:foo)`⌟ è qualcos'altro (un gruppo non-catturante contenente la sottoespressione ⌜`foo`⌟).

Python aggiunge una estensione di sintassi a quella del Perl. Se il primo carattere dopo il punto interrogativo è una '`P`', sappiate che è un'estensione specifica di Python. Al momento esistono due di queste estensioni: ⌜`(?P<`*nome*`>...)`⌟ definisce un gruppo con nome, e ⌜`(?P=`*nome*`)`⌟ è un retroriferimento ad un gruppo con nome. Se le versioni future di Perl 5 aggiungeranno simili funzionalità usando una differente sintassi, il modulo `re` sarà modificato per supportare la nuova sintassi, mentre verrà preservata la sintassi specifica per Python per un riguardo alla compatibilità.

Adesso che abbiamo dato un'occhiata alle estensioni della sintassi generale, possiamo ritornare alle funzionalità che semplificano il lavoro con i gruppi nelle RE complesse. Visto che i gruppi sono numerati da sinistra a destra ed un'espressione complessa può sfruttare molti gruppi, può divenire difficile tenere traccia della corretta numerazione, e la modifica di una RE complessa del genere è seccante. Inserite un nuovo gruppo vicino all'inizio, e modificate i numeri di tutto ciò che segue.

A volte vorrete usare un gruppo per raccogliere parte di una espressione regolare, ma non siete interessati a recuperare il contenuto del gruppo. Potete rendere esplicito questo fatto usando un gruppo non-catturante: ⌜`(?:...)`⌟, dovrete inserire ogni altra espressione regolare tra le parentesi.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Eccetto per il fatto che non potete recuperare il contenuto di ciò a cui il gruppo corrisponde, un gruppo non-catturante si comporta esattamente come un gruppo non-catturante; potete inserirvi ogni cosa, ripetendola con un metacarattere di ripetizione come '`*`', e nidificarlo con altri gruppi (catturanti o non-catturanti). ⌜`(?:...)`⌟ è particolarmente utile quando modificate un gruppo esistente, dato che potete aggiungere nuovi gruppi senza cambiare il modo in cui tutti gli altri gruppi sono numerati. Dovrebbe essere menzionato che non vi è alcuna differenza di prestazioni nella ricerca fra gruppi catturanti e non-catturanti; e nemmeno che una forma risulta più veloce dell'altra.

In secondo luogo, e più significante caratteristica, sono i gruppi con nome; invece che riferirsi ad essi tramite numeri, a questi gruppi vi si può riferire con un nome.

La sintassi per un gruppo con nome è una delle estensioni specifiche di Python: ⌜`(?P<`*nome*`>...)`⌟. *nome* è, ovviamente, il nome del gruppo. Eccetto per l'associazione del nome ad un gruppo, i gruppi con nome si comportano in modo identico anche per i gruppi catturanti. Il metodo `MatchObject` che si occupa di tutti i gruppi catturanti accetta o interi, per riferirsi ai gruppi tramite numero, o stringhe contenenti il nome del gruppo. Gruppi con nome restituiscono ancora numeri, in modo che possiate ottenere informazioni circa un gruppo in due modi:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Molta punteggiatura )))' )
>>> m.group('word')
'Molta'
>>> m.group(1)
'Molta'
```

I gruppi con nome sono comodi perché vi lasciano utilizzare nomi facili da ricordare, invece di ricordare numeri. Ecco un esempio di RE dal modulo `imaplib`:

```
InternalDate = re.compile(r'INTERNALDATE "'
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
r'(?P<year>[0-9][0-9][0-9][0-9])'
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
        r'"')
```

È ovviamente più facile recuperare `m.group('zonem')`, invece di dover ricordare di recuperare il gruppo 9.

Visto che la sintassi per i retroriferimenti, in una espressione come `(...)\1`, si riferisce al numero del gruppo, esiste una variante naturale che usa il nome del gruppo al posto del numero. Questa è anche un'estensione Python: `(?P=`*nome*`)` indica che il contenuto del gruppo chiamato *nome* dovrebbe essere trovato ancora al punto corrente. L'espressione regolare per trovare parole doppie, `(\b\w+)\s+\1` può anche essere scritta come `(?P<word>\b\w+)\s+(?P=word)`:

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Parigi in in estate').group()
'in in'
```

4.4 Asserzioni lookahead

Un'altra asserzione a lunghezza zero è l'asserzione lookahead. Le asserzioni lookahead sono disponibili in entrambe le forme, positiva e negativa, e si presentano così:

`(?=...)` Asserzione lookahead positiva. Questa riesce se il contenuto dell'espressione regolare, qui rappresentato da `...`, corrisponde con successo alla corrente posizione e fallisce in ogni altro caso. Ma, una volta che l'espressione contenuta è stata verificata, il motore di ricerca delle corrispondenze non avanza più; il resto del modello è ritenuto giusto finché l'asserzione è verificata.

`(?!...)` Asserzione lookahead negativa. Questa è l'opposto della asserzione positiva; è verificata se il contenuto dell'espressione *non* corrisponde alla posizione corrente nella stringa.

Un esempio aiuterà a concretizzare con una dimostrazione un caso dove un lookahead è utile. Consideriamo un semplice modello per verificare un nome di file e dividerlo in nome ed estensione, il separatore è il '`.`'. Per esempio, in '`news.rc`', '`news`' è il nome ed '`rc`' è l'estensione del file.

Il modello che verifica questa espressione è veramente semplice:

`.*[.].*$`

Notare che il punto '`.`' è trattato in modo speciale perché è un metacarattere; è stato inserito in una classe di caratteri. Notare anche il `$` a fine riga; questo è stato aggiunto per assicurarsi che tutto il resto della stringa sia stata inclusa nell'estensione. Questa espressione regolare corrisponde con '`foo.bar`', '`autoexec.bat`', '`sendmail.cf`' e '`printers.conf`'.

Ora consideriamo un problema un po' più complicato; cosa succede se volete verificare un nome di file dove l'estensione non è '`bat`'? Alcuni tentativi errati:

```
.*[.][^b].*$
```

Il primo tentativo prova ad escludere '`bat`' richiedendo che il primo carattere dell'estensione non sia una '`b`'. Questo è sbagliato, perché il modello non corrisponde anche con '`foo.bar`'.

```
⌜.*[.]([^b]..|.[^a].|..[^t])$⌟
```

L'espressione diviene più disordinata quando si prova ad aggiustare la prima soluzione richiedendo uno dei seguenti casi da verificare: il primo carattere dell'estensione non è '`b`'; il secondo carattere non è '`a`'; o il terzo carattere non è '`t`'. Questo accetta '`foo.bar`' e rifiuta '`autoexec.bat`', ma richiede una estensione con tre lettere e non vuole accettare nomi di file con estensioni a due lettere come '`sendmail.cf`'. Complichiamo ancora il modello sforzandoci di risolvere il problema.

```
⌜.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$⌟
```

Nel terzo tentativo, la seconda e terza lettera sono state poste in facoltativamente per consentire la corrispondenza di estensioni più corte di tre caratteri come '`sendmail.cf`'.

Il modello è realmente complicato adesso, perché è diventato difficile da leggere e capire. Se il problema cambia e si esclude sia '`bat`' che '`exe`' come estensioni, il modello diventerà ancora più complicato e confuso.

Un lookahead negativo risolve tutto questo:

```
⌜.*[.](?!bat$).*$⌟
```

Il lookahead significa: se l'espressione ⌜`bat`⌟ non corrisponde a questo punto, prova il resto del modello; se ⌜`bat$`⌟ corrisponde, il resto del modello fallirà. Il carattere ⌜`$`⌟ è richiesto per assicurarsi che qualcosa simile a '`sample.batch`', dove l'estensione del file riconoscerebbe solo l'inizio con '`bat`', sia consentita.

Escludendo altre estensioni di nomi di file è adesso facile semplicemente aggiungerle come un'alternativa nell'asserzione. Il seguente modello esclude nomi di file che finiscono con '`bat`' o '`exe`':

```
⌜.*[.](?!bat$|exe$).*$⌟
```

5 Modificare le stringhe

Fino a questo punto, abbiamo semplicemente effettuato ricerche su una stringa statica. Le espressioni regolari sono anche utilizzate per modificare una stringa statica in vari modi, utilizzando i seguenti metodi di `RegexObject`:

Metodo/Attributo	Scopo
`split()`	Divide la stringa in una lista, spezzandola dove la RE corrisponde
`sub()`	Trova tutte le sottostringhe dove la RE corrisponde e le sostituisce con una stringa differente
`subn()`	Si comporta come la `sub()`, ma restituisce la nuova stringa e il numero delle sostituzioni effettuate

5.1 Suddividere le stringhe

Il metodo `split()` di `RegexObject` suddivide una stringa dove l'espressione regolare corrisponde e restituisce una lista delle varie parti. È simile al metodo `split()` delle stringhe, ma permette di specificare in modo più generico i delimitatori con cui spezzare la stringa; `split()` consente di suddividere la stringa solo tramite spazi vuoti o una stringa fissa. Come potreste aspettarvi, esiste anche una funzione a livello di modulo `re.split()`.

`split`(*string* [, *maxsplit* = *0*])

Suddivide *string* usando le corrispondenze dell'espressione regolare. Se nell'espressione regolare sono utilizzate dei sottogruppi catturanti (quindi delimitati da parentesi tonde) il loro contenuto verrà restituito come parte della lista risultante. Se *maxsplit* non è zero, vengono eseguite tutte le suddivisioni possibili.

È possibile limitare il numero delle divisioni effettuate, assegnando un valore a *maxsplit*. Se *maxsplit* non è zero, vengono effettuate tutte le suddivisioni possibili e la parte rimanente della stringa viene restituita come elemento finale della lista. Nell'esempio seguente, il delimitatore è costituito da qualunque sequenza di caratteri non alfanumerici.

```
>>> p = re.compile(r'\W+', re.LOCALE)
>>> p.split('Trattasi di una una prova, breve e piacevole, di split().')
['Trattasi', 'di', 'una', 'una', 'prova', 'breve', 'e', 'piacevole', 'di', 'split', '']
>>> p.split('Trattasi di una una prova, breve e piacevole, di split().', 3)
['Trattasi', 'di', 'una', 'una prova, breve e piacevole, di split().']
```

Qualche volta non sarete interessati solamente al testo tra i delimitatori, ma avrete anche la necessità di conoscere esattamente il delimitatore. Se dei sottogruppi sono utilizzati nella RE il loro valore verrà restituito come parte della lista. Confrontate le seguenti chiamate:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('Trattasi di... una prova.')
['Trattasi', 'di', 'una', 'prova', '']
>>> p2.split('Trattasi di... una prova.')
['Trattasi', ' ', 'di', '... ', 'una', ' ', 'prova', '.', '']
```

La funzione a livello di modulo `re.split()` aggiunge la RE come primo parametro, per il resto è identica.

```
>>> re.split('[\W]+', 'Parole, parole, parole.')
['Parole', 'parole', 'parole', '']
>>> re.split('([\W]+)', 'Parole, parole, parole.')
['Parole', ', ', 'parole', ', ', 'parole', '.', '']
>>> re.split('[\W]+', 'Parole, parole, parole', 1)
['Parole', 'parole, parole.']
```

5.2 Ricerca e sostituzione

Un altro comune compito è la ricerca di tutte le corrispondenze di un modello e la sostituzione con una stringa differente. Il metodo `sub()` prende il valore da sostituire, che può essere una stringa o una funzione, e la stringa che deve essere elaborata.

sub(*sostituzione*, *stringa*[, *count* = *0*])

Restituisce la stringa ottenuta sostituendo le occorrenze più a sinistra non sovrapposte alla RE nella *stringa* di sostituzione *sostituzione*. Se il modello non è stato trovato, viene restituita la *stringa* immutata.

L'argomento opzionale *count* è il massimo numero di occorrenze del modello da sostituire. *count* deve essere un numero intero positivo. Il valore predefinito è 0 che significa sostituire tutte le occorrenze.

Questo è un semplice esempio dell'uso del metodo `sub()`. Sostituisce i nomi dei colori con la parola '`colore`':

```
>>> p = re.compile( '(blu|bianco|rossa)')
>>> p.sub( 'colore', 'calzino blu e scarpa rossa')
'calzino colore e scarpa colore'
>>> p.sub( 'colore', 'calzino blu e scarpa rossa', count=1)
'calzino colore e scarpa rossa'
```

Il metodo `subn()` fa lo stesso lavoro, ma restituisce una tupla con due indici, il nuovo valore della stringa ed il numero delle sostituzioni che sono state eseguite:

```
>>> p = re.compile( '(blu|bianco|rossa)')
>>> p.subn( 'colore', 'calzino blu e scarpa rossa')
('calzino colore e scarpa colore', 2)
>>> p.subn( 'colore', 'nessun colore per tutti')
('nessun colore per tutti', 0)
```

Corrispondenze vuote sono sostituite solo quando non sono adiacenti ad una precedente corrispondenza.

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

Se *sostituzione* è una stringa, ogni backslash di protezione presente viene processato. Ad esempio '`\n`' è convertito in un singolo carattere di fine riga, '`\r`' è convertito in un carattere a capo e così via. Caratteri di protezione come '`\j`' andranno a sinistra da soli. Riferimenti all'indietro come '`\6`', sono sostituiti con la sottostringa corrispondente nel relativo gruppo della RE. Questo permette di incorporare porzioni del testo originale nella risultante stringa di sostituzione.

Questo esempio confronta le parole '`sezione`' seguite da una stringa racchiusa tra '`{`', '`}`' e cambia '`sezione`' in '`sottosezione`':

```
>>> p = re.compile('sezione{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'sottosezione{\1}','sezione{Prima} sezione{seconda}')
'sottosezione{Prima} sottosezione{seconda}'
```

Questa è anche una sintassi per riferirsi a gruppi con nome come definito dalla sintassi `(?P<nome>...)`. '`\g<nome>`' sarà usato dalla stringa corrispondente dal gruppo con nome '`nome`' e '`\g<`*numero*`>`' usato dal corrispondente gruppo numerato. '`\g<2>`' è perciò equivalente a '`\2`' ma è ambiguo in una sostituzione di stringa come '`\g<2>0`'. ('`\g<2>0`' verrebbe interpretato come un riferimento al gruppo 20, non a riferimento al gruppo 2 seguito da una costante '`0`'.) Le seguenti sostituzioni sono tutte equivalenti, ma usano tutte e tre le varianti della sostituzione di stringa.

```
>>> p = re.compile('sezione{ (?P<nome> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'sottosezione{\1}','sezione{Prima}')
'sottosezione{Prima}'
>>> p.sub(r'sottosezione{\g<1>}','sezione{Prima}')
'sottosezione{Prima}'
>>> p.sub(r'sottosezione{\g<nome>}','sezione{Prima}')
'sottosezione{Prima}'
```

sostituzione può anche essere una funzione che fornisce più controllo. Se *sostituzione* è una funzione, è chiamata per ogni occorrenza del *modello* che non si sovrappone. In ogni chiamata la funzione è passata all'argomento `MatchObject` per la corrispondenza e può usare questa informazione per calcolare la desiderata stringa di sostituzione da restituire.

Nel seguente esempio, la funzione di sostituzione traduce decimali in esadecimali:

```
>>> def hexrepl( match ):
...     "Restituisce una stringa esadecimale per numeri decimali"
...     value = int( match.group() )
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Chiama 65490 per stampare, 49152 per codice utente.')
'Chiama 0xffd2 per stampare, 0xc000 per codice utente.'
```

Quando usate la funzione `re.sub()` a livello di modulo, il modello è passato come primo argomento. Il modello potrebbe essere una stringa o un `RegexObject`; se c'è necessità di specificare delle opzioni in espressioni regolari dovrete usare `RegexObject` come primo parametro o modificatori incorporati del modello, ad esempio `sub((?i)b+, x, bbbb BBBB)` restituisce `'x x'`.

6 Problemi comuni

Le espressioni regolari sono uno strumento potente per alcune applicazioni, ma il loro comportamento non è intuitivo e certe volte non agiscono nel modo che ci attendiamo. In questa sezione evidenzieremo alcuni tra i più comuni errori in cui si può cadere.

6.1 Usare i metodi delle stringhe

Talvolta utilizzare il modulo `re` è un errore. Se stiamo cercando la corrispondenza di una singola stringa, o di un singolo carattere e non stiamo utilizzando alcuna funzionalità delle `re` come l'opzione `IGNORECASE`, allora non è necessario l'utilizzo della potenza delle espressioni regolari. Le stringhe hanno parecchi metodi per eseguire operazioni basate su stringhe fisse e sono di solito molto efficienti, poiché l'implementazione è un singolo e semplice ciclo scritto in linguaggio C che è stato ottimizzato per quello scopo, contrariamente ad un motore per le espressioni regolari che è di grandi dimensioni e progettato per utilizzi più generali.

Un esempio potrebbe essere la sostituzione di un singola stringa prefissata con un'altra; per esempio, se vogliamo sostituire la parola '`word`' con '`deed`', `re.sub()` sembrerebbe essere lo strumento giusto da utilizzare, ma possiamo prendere in considerazione anche il metodo `replace()` delle stringhe. Ricordiamo però che `replace()` sostituirebbe '`word`' anche come sottostringa dentro eventuali parole che la contengano, per esempio trasformando '`swordfish`' in '`sdeedfish`', ma anche la semplice espressione regolare `word` avrebbe fatto lo stesso. (Per evitare la sostituzione di sottostringhe di una parola, l'espressione regolare è `\bword\b`, che richiede che la parola '`word`' sia delimitata (da uno spazio, tab ecc. in entrambi i lati.) Questo compito quindi va oltre le possibilità di `replace`).

Un'altra attività piuttosto comune è quella di eliminare ogni occorrenza di un singolo carattere o sostituirla con un altro carattere. Si può fare questo con qualcosa tipo `re.sub('\n', ' ', S)`, ma `translate()` può fare entrambe le cose ed essere più veloce di qualsiasi operazione tramite espressioni regolari.

In breve, prima di rivolgervi al modulo `re`, prendete in considerazione se il problema può essere risolto con un semplice e più rapido metodo delle stringhe.

6.2 match() contro search()

La funzione `match()` controlla unicamente se l'espressione regolare combacia con l'inizio della stringa, mentre `search()` procederà lungo la stringa fino alla prima occorrenza. È importante aver chiara in mente questa distinzione. Ricordate, `match()` avrà successo solo se l'occorrenza parte dal primo carattere; se questa iniziasse oltre, `match()` non la troverebbe.

```
>>> print re.match('super', 'superstizione').span()
(0, 5)
>>> print re.match('super', 'insuperabile')
None
```

Diversamente, `search()` procederà lungo la stringa restituendo la prima corrispondenza che trova.

```
>>> print re.search('super', 'superstizione').span()
(0, 5)
>>> print re.search('super', 'insuperabile').span()
(2, 7)
```

Talvolta potreste essere tentati di utilizzare comunque `re.match()` e semplicemente aggiungere ⌜`.*`⌟ all'inizio della RE. Resistete alla tentazione e utilizzate `re.search()` al suo posto. Il compilatore di RE esegue alcune analisi per rendere più veloce il processo di ricerca di una corrispondenza. Una di queste analisi si basa su quale deve essere il primo carattere di una corrispondenza positiva; per esempio, un modello che inizia con ⌜`Crow`⌟ troverà corrispondenza con una '`C`'. L'analisi fa in modo che il motore proceda rapidamente attraverso la stringa cercando quel carattere, tentando la corrispondenza completa solo quando viene effettivamente trovata una '`C`'.

Aggiungere ⌜`.*`⌟ impedisce questa ottimizzazione, richiedendo invece di procedere fino alla fine della stringa per poi tornare indietro per cercare una corrispondenza con il resto della RE. Utilizzate quindi `re.search()`.

6.3 Qualificatori ingordi contro non-ingordi

In un'espressione regolare con qualificatori di ripetizione, come ⌜`a*`⌟, l'azione risultante è quella di applicare il modello o lo schema di ricerca quanto più possibile. Spesso questo comportamento non è desiderabile quando si stanno cercando le corrispondenze di una coppia opposta di delimitatori, come possono essere ad esempio le parentesi angolari che delimitano un tag HTML. Il modello più semplice per discernere un semplice tag HTML non funziona a causa della natura, per così dire, ingorda (Ndt greedy) di ⌜`.*`⌟.

```
>>> s = '<html><head><title>Titolo</title>'
>>> len(s)
33
>>> print re.match('<.*>', s).span()
(0, 33)
>>> print re.match('<.*>', s).group()
<html><head><title>Titolo</title>
```

La RE discerne il carattere '`<`' in '`<html>`', e la parte ⌜`.*`⌟ consuma il resto della stringa. Tuttavia è rimasto ancora qualcosa nella RE, e il carattere ⌜`>`⌟ non può corrispondere alla fine della stringa, perciò il motore dell'espressione regolare deve indietreggiare carattere dopo carattere finché non trova una corrispondenza per ⌜`>`⌟. La corrispondenza finale si estende così dal carattere '`<`' in '`<html>`' fino a '`>`' in '`</title>`', il che non è ciò che si vuole.

In questo caso, la soluzione è di usare i qualificatori non-ingordi ⌜`*?`⌟, ⌜`+?`⌟, ⌜`??`⌟, oppure ⌜`{m,n}?`⌟, che selezionano la minore quantità possibile di testo. Nell'esempio precedente, la parentesi '`>`' è cercata immediatamente dopo la prima corrispondenza di '`<`', e in caso di insuccesso, il meccanismo di ricerca avanza di un carattere alla volta ricercando ad ogni passo il carattere '`>`'. Questo produce proprio il risultato corretto:

```
>>> print re.match('<.*?>', s).group()
<html>
```

(Da evidenziare che l'analisi di un testo HTML o XML mediante le espressioni regolari è piuttosto difficile. Modelli di ricerca grossolani e sbrigativi gestiranno i casi più comuni, ma HTML ed XML includono casi che violeranno l'espressione regolare ovvia; e anche quando se ne scrivesse una per gestire tutti i casi possibili, i

modelli risulterebbero *veramente* complicatissimi. Per tali compiti è preferibile usare un modulo che implementa un parser HTML o XML.)

6.4 Non usare re.VERBOSE

Avrete ormai probabilmente notato che le espressioni regolari sono una notazione molto compatta, ma anche di difficile lettura. RE di moderata complessità possono diventare insiemi molto lunghi di backslash, parentesi e metacaratteri, rendendole difficili da leggere e capire.

Per tali RE, può essere di aiuto specificare l'opzione `re.VERBOSE` al momento della compilazione, perché permette di scrivere l'espressione regolare in una forma più chiara.

L'opzione `re.VERBOSE` ha diversi effetti. Uno spazio bianco, che non appartenga ad una classe di caratteri, in una espressione regolare viene ignorato. Ciò significa che un'espressione come `cane | gatto` è equivalente alla meno leggibile `cane|gatto`; tuttavia `[a b]` corrisponderà ancora ai caratteri '`a`', '`b`' oppure ad uno spazio. Inoltre, potrete anche inserire dei commenti in una RE; i commenti si estendono da un carattere '#' fino al successivo fine riga. Se usato con stringhe delimitate da tre virgolette, permette di formattare le RE in maniera più elegante:

```
pat = re.compile(r"""
 \s*                 # Salta gli spazi iniziali
 (?P<header>[^:]+)   # Parola header
 \s* :               # Spazio bianco e due punti
 (?P<value>.*?)      # Valore di header  -- *? è usato per
                     # trascurare gli spazi finali che seguono
 \s*$                # Spazi in coda fino a fine riga
""", re.VERBOSE)
```

Questo è molto più leggibile di:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

7 Suggerimenti sul documento

Le espressioni regolari sono un argomento complicato. Vi è servito questo documento per comprenderle meglio? C'erano parti non chiare, o problemi, che avete incontrato, che non sono stati trattati qui? In tal caso inviate pure all'autore i vostri suggerimenti per migliorarlo.

Il libro più completo sulle espressioni regolari è quasi certamente Jeffrey Friedl's *Mastering Regular Expressions*, pubblicato da O'Reilly. Sfortunatamente, si concentra esclusivamente sui tipi di espressioni regolari in Perl e Java e non contiene materiale su Python, perciò non vi sarà utile come riferimento per la programmazione in Python. (La prima edizione del libro trattava il modulo, ormai obsoleto, `regex` di Python e non vi sarà di grande aiuto). Considerate l'opportunità di consultarlo in biblioteca.