Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2016/2017

Linguaggio Ruby: espressioni regolari (parte II)

Docente del laboratorio: Raffaella Rizzi

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole

Il pattern `/cat/i` rappresenta l'insieme {"cat", "Cat", "cAt", "caT", "CaT", "CAt", "cAT", "CAT"}

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole
- `m` ➔ *multiline* → include `\n` nella classe rappresentata dal punto `.`

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole
- `m` ➔ *multiline* → include `\n` nella classe rappresentata dal punto `.`
- `o` ➔ *substitute once* → la sostituzione di #{} nel letterale è effettuata una volta sola (la prima volta che il letterale viene valutato per creare il pattern)

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modifica… *matching*

- `i` → …che le lette…
- `m` → …esentata dal …
- `o` → …el letterale è ef… …il letterale vien…

```
3.times do |i|
    p = /current i: #{i}/
end
```

Senza opzione `o` il pattern alle varie iterazioni è:

per i = 0 → `/current i: 0/`

per i = 1 → `/current i: 1/`

per i = 2 → `/current i: 2/`

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modifica… *matching*

- `i` → …che le lette…
- `m` → …esentata dal …
- `o` → …el letterale è ef… …il letterale vien…

```
3.times do |i|
   p = /current i: #{i}/o
end
```

Con opzione `o` il pattern alle varie iterazioni è:

per i = 0 → `/current i: 0/`

per i = 1 → `/current i: 0/`

per i = 2 → `/current i: 0/`

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole
- `m` ➔ *multiline* → include `\n` nella classe rappresentata dal punto `.`
- `o` ➔ *substitute once* → la sostituzione di #{} nel letterale è effettuata una volta sola (la prima volta che il letterale viene valutato per creare il pattern)
- `x` ➔ *extended mode* → ignora gli spazi e permette i commenti all'interno del letterale

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* `/` terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole
- `m` ➔ *multiline* → include `\n` nella classe rappresentata dal punto `.`
- `o` ➔ *substitute once* → la sostituzione di #{} nel letterale è effettuata una volta sola (la prima volta che il letterale viene valutato per creare il pattern)
- `x` ➔ *ex* Il pattern `/c t/x` equivale al pattern `/ct/` te i commenti all'interno del letterale

Opzioni di *matching*

Le opzioni di *matching* sono simboli che si possono aggiungere dopo lo *slash* / terminale del letterale per modificare il comportamento dell'operazione di *matching*

- `i` ➔ *case insensitive* → viene ignorato il fatto che le lettere siano maiuscole o minuscole
- `m` ➔ *multiline* → include `\n` nella classe rappresentata dal punto `.`
- `o` ➔ *substitute once* → la sostituzione di #{} nel letterale è effettuata una volta sola (la prima volta che il letterale viene v
- `x` ➔ *ex* tte i commenti all'interno del letterale

Il pattern `/ciao#ciao/x` equivale al pattern `/ciao/`

La classe `Regexp`

Un pattern è un oggetto appartenente alla classe `Regexp`, e può quindi (in alternativa al letterale) essere costruito attraverso il costruttore di classe:

```
pattern = Regexp.new(pattern_string)
```

dove *`pattern_string`* è la stringa che (valutata) fornisce il pattern. Ad esempio:

```
pt = Regexp.new("ca+b")
```

è equivalente ad assegnare a `pt` il letterale:

```
pt = /ca+b/
```

La classe `Regexp`

La classe `Regexp` mette a disposizione il metodo `match` che (in alternativa all'operatore =~) effettua il *matching* con una stringa e restituisce un oggetto di tipo `MatchData` (o `nil` se non c'è *matching*)

```
match_obj = pattern.match(string)
```

`string` è la stringa con cui si intende effettuare il *matching*

La classe `Regexp`

La classe `Regexp` mette a disposizione il metodo `match` che (in alternativa all'operatore =~) effettua il *matching* con una stringa e restituisce un oggetto di tipo `MatchData` (o `nil` se non c'è *matching*)

```
match_obj = pattern.match(string)
```

`string` è la stringa con cui si intende effettuare il *matching*

```
str = "gatto cane"
p = Regexp.new("\\w+\\s\\w+")
m = p.match(str)
puts "#{p.class}"
puts "#{m.class}"
```

La classe `Regexp`

La classe `Regexp` mette a disposizione il metodo `match` che (in alternativa all'operatore =~) effettua il *matching* con una stringa e restituisce un oggetto di tipo `MatchData` (o `nil` se non c'è *matching*)

```
match_obj = pattern.match(string)
```

`string` è la stringa con cui si intende effettuare il *matching*

Equivale a `p = /\w+\s\w+/`

```
str = "
p = Regexp.new("\\w+\\s\\w+")
m = p.match(str)
puts "#{p.class}"
puts "#{m.class}"
```

La classe `Regexp`

La classe `Regexp` mette a disposizione il metodo `match` che (in alternativa all'operatore =~) effettua il *matching* con una stringa e restituisce un oggetto di tipo `MatchData` (o `nil` se non c'è *matching*)

```
match_obj = pattern.match(string)
```

```
str
mat
```

```
>Regexp
>MatchData
>
```

```
str
p = Regexp.new("\\w+\\s\\w+")
m = p.match(str)
puts "#{p.class}"
puts "#{m.class}"
```

La classe `MatchData`

Un oggetto di tipo `MatchData` incapsula tutte le informazioni del *matching* e funziona come un array; i suoi elementi sono quindi accessibile con la solita sintassi:

```
match_obj[index]
```

Gli elementi sono (nell'ordine) la sottostringa di *matching* e gli (eventuali) n *matching* dovuti ai raggruppamenti. La lunghezza dell'array è n+1.

```
str = "TH145 657FV"
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
```

La classe `MatchData`

Un oggetto di tipo `MatchData` incapsula tutte le informazioni del *matching* e funziona come un array; i suoi elementi sono quindi accessibile con la solita sintassi:

```
match_obj[index]
```

Gli elementi
matching e
raggruppam

```
str = "TH14
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
```

Nell'oggetto `m` sono contenuti cinque elementi di cui il primo è la sottostringa di *matching* "TH145 657FV" e gli altri sono i *matching* catturati dai quattro raggruppamenti presi da sinistra verso destra.

La classe MatchData

```
str = “TH145 657FV”
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts “L=#{len}”
puts “Match: \”#{m[0]}\””
groups = m[1..len]
groups.each {|g|
  puts g
}
```

La classe MatchData

```
str = “TH145 657FV”
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts “L=#{len}”
puts “Match: \”#{m[0]}\””
groups = m[1..len]
groups.each {|g|
  puts g
}
```

```
>L=5
>Match: “TH145 657FV”
>TH14
>5
>657
>FV
>
```

La classe MatchData

```
str = "TH145 657FV"
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts "L=#{len}"
puts "Match: \"#{m[0]}\""
groups = m[1..len]
groups.each {|g|
  puts g
}
```

```
>L=5
>Match: "TH145 657FV"
>TH14
>5
>657
>FV
>
```

La classe MatchData

```
str = "TH145 657FV"
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts "L=#{len}"
puts "Match: \"#{m[0]}\""
groups = m[1..len]
groups.each {|g|
  puts g
}
```

```
>L=5
>Match: "TH145 657FV"
>TH14
>5
>657
>FV
>
```

La classe `MatchData`

```
str = “TH145 657FV”
p = /(\w+)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts “L=#{len}”
puts “Match: \”#{m[0]}\””
groups = m[1..len]
groups.each {|g|
  pu
}
```

Come si deve modificare il pattern `p` per poter recuperare le stringhe “TH”, “145”, “657”, “FV”?

```
>L=5
>Match: “TH145 657FV”
>TH14
>5
>657
>FV
>
```

La classe MatchData

```
str = “TH145 657FV”
p = /(\w+?)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts “L=#{len}”
puts “Match: \”#{m[0]}\””
groups = m[1..len]
groups.each {|g|
  puts g
}
```

```
>L=5
>Match: “TH145 657FV”
>TH
>145
>657
>FV
>
```

La classe MatchData

```
str = “TH145 657FV”
p = /(\w+?)(\d+)\s(\d+)(\w+)/
m = p.match(str)
len=m.length
puts “L=#{len}”
puts “Match: \”#{m[0]}\””
groups = m[1..len]
groups.each {|g|
  puts g
}
```

```
>L=5
>Match: “TH145 657FV”
>TH
>145
>657
>FV
>
```

La classe `MatchData`

La classe `MatchData` mette a disposizione dei metodi per accedere alle informazioni di *matching*, tra i quali:

- `length` ➔ restituisce il numero di elementi contenuti
- `begin(`*`index`*`)` ➔ restituisce l'offset di inizio (all'interno della stringa) dell'elemento di indice *`index`*
- `end(`*`index`*`)` ➔ restituisce l'offset di fine (all'interno della stringa) dell'elemento di indice *`index`*
- `offset(`*`index`*`)` ➔ restituisce, in un array di lunghezza 2, gli offset di inizio e di fine dell'elemento di indice *`index`*

La classe `MatchData`

…

- `pre_match` ➔ restituisce il prefisso di stringa prima della sottostringa di *matching*
- `post_match` ➔ restituisce il suffisso di stringa dopo la sottostringa di *matching*

La classe MatchData

```
str = “TH145 657FV”
p = /(\d+)\s(\d+)/
m = p.match(str)
puts “First group: ^#{m[1]}^”
puts “Start offset: #{m.begin(1)}”
puts “End offset: #{m.end(1)}”
puts “The match: ^m[0]^”
puts “Before the match: ^m.pre_match^”
puts “After the match: ^m.post_match^”
```

La classe MatchData

```
str = “TH145 657FV”
p = /(\d+)\s(\d+)/
m = p.match(str)
puts “First group: ^#{m[1]}^”
puts “Start offset: #{m.begin(1)}”
puts “End offset: #{m.end(1)}”
puts “The match: ^m[0]^”
puts “Before the match: ^m.pre_match^”
puts “After the match: ^m.post_match^”
```

```
>First group: ^145^
>Start offset: 2
>End offset: 5
>The match: ^145 657^
>Before the match: ^TH^
>After the match: ^FV^
>
```

La classe MatchData

```
str = "TH145 657FV"
p = /(\d+)\s(\d+)/
m = p.match(str)
puts "First group: ^#{m[1]}^"
puts "Start offset: #{m.begin(1)}"
puts "End offset: #{m.end(1)}"
puts "The match: ^m[0]^"
puts "Before the match: ^m.pre_match^"
puts "After the match: ^m.post_match^"
```

```
>First group: ^145^
>Start offset: 2
>End offset: 5
>The match: ^145 657^
>Before the match: ^TH^
>After the match: ^FV^
>
```

La classe MatchData

```
str = “TH145 657FV”
p = /(\d+)\s(\d+)/
m = p.match(str)
puts “First group: ^#{m[1]}^”
puts “Start offset: #{m.begin(1)}”
puts “End offset: #{m.end(1)}”
puts “The match: ^m[0]^”
puts “Before the match: ^m.pre_match^”
puts “After the match: ^m.post_match^”
```

```
>First group: ^145^
>Start offset: 2
>End offset: 5
>The match: ^145 657^
>Before the match: ^TH^
>After the match: ^FV^
>
```

La classe MatchData

```
str = "TH145 657FV"
p = /(\d+)\s(\d+)/
m = p.match(str)
puts "First group: ^#{m[1]}^"
puts "Start offset: #{m.begin(1)}"
puts "End offset: #{m.end(1)}"
puts "The match: ^m[0]^"
puts "Before the match: ^m.pre_match^"
puts "After the match: ^m.post_match^"
```

```
>First group: ^145^
>Start offset: 2
>End offset: 5
>The match: ^145 657^
>Before the match: ^TH^
>After the match: ^FV^
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Il metodo `sub` della classe `String` sostituisce con la stringa *`s_string`* la prima occorrenza di un *matching* con *`pattern`* nella stringa invocante `string`.

NB: il metodo lascia inalterata la stringa invocante *`string`* e restituisce una copia contenente la sostituzione.

Se nessuna sostituzione viene effettuata, allora il metodo restituisce una copia della stringa invocante *`string`*.

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.sub(/\d+/, “NNN”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.sub(/\d+/, “NNN”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>THNNN 657FV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *`s_string`* passata come argomento, si possono anche usare i *back references* `\1`, `\2`, …, `\n`

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si possono anche usare i *back references* \1, \2, …, \n

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\1”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si possono anche usare i *back references* `\1`, `\2`, …, `\n`

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\1”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH145FV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si possono anche usare i *back references* `\1`, `\2`, …, `\n`

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\1”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH145FV
>
```

Sostituisce la sottostringa di *matching* con quanto è stato catturato dal primo raggruppamento

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *`s_string`* passata come argomento, si può anche usare il metasimbolo `\&` che rappresenta la sottostringa di *matching*

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo `\&` che rappresenta la sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “|\\&|”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo `\&` che rappresenta la sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “|\\&|”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH|145 657|FV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \+ che rappresenta il *last matched group*.

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \+ che rappresenta il *last matched group*.

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\+”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \+ che rappresenta il *last matched group*.

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\+”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH657FV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \` che rappresenta il prefisso prima della sottostringa di *matching*

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \` che rappresenta il prefisso prima della sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\`”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo \` che rappresenta il prefisso prima della sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\`”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>THTHFV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *`s_string`* passata come argomento, si può anche usare il metasimbolo `\'` che rappresenta il suffisso dopo la sottostringa di *matching*

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo `\'` che rappresenta il suffisso dopo la sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\’”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern, s_string)
```

Nella stringa *s_string* passata come argomento, si può anche usare il metasimbolo `\'` che rappresenta il suffisso dopo la sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/(\d+)\s(\d+)/, “\\’”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>THFVFV
>
```

Sostituzione in una stringa

```
new_string = string.sub(pattern) block
```

Al metodo `sub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

Sostituzione in una stringa

```
new_string = string.sub(pattern) block
```

Al metodo `sub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/\d+/){ |m_str|
  “***”
}
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.sub(pattern) block
```

Al metodo `sub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.sub(/\d+/){ |m_str|
  “***”
}
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH*** 657FV
>
```

Sostituzione in una stringa

```
new_string = string.gsub(pattern, s_string)
```

Il metodo `gsub` della classe `String` sostituisce *con la stringa* `s_string` tutte le occorrenze di un *matching* con `pattern` della stringa invocante `string`.

Il metodo lascia inalterata la stringa invocante `string` e restituisce una copia contenente la sostituzione.

Se nessuna sostituzione viene effettuata, allora il metodo restituisce una copia della stringa invocante `string`.

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.gsub(/\d+/, “***”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.gsub(/\d+/, “***”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH*** ***FV
>
```

Sostituzione in una stringa

```
new_string = string.gsub(pattern) block
```

Al metodo `gsub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

Sostituzione in una stringa

```
new_string = string.gsub(pattern) block
```

Al metodo `gsub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.gsub(/\d+/){ |m_str|
  “***”
}
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
new_string = string.gsub(pattern) block
```

Al metodo `gsub` può anche essere associato un blocco a cui viene passata la sottostringa di *matching*. Il valore restituito dal blocco è sostituito alla sottostringa di *matching*

```
str = “TH145 657FV”
new_str = str.gsub(/\d+/){ |m_str|
  “***”
}
puts “#{str}“
puts “#{new_str}“
```

```
>TH145 657FV
>TH*** ***FV
>
```

Sostituzione in una stringa

Esercizio1: trasformare in minuscolo tutte le vocali di una stringa.

Sostituzione in una stringa

Esercizio1: trasformare in minuscolo tutte le vocali di una stringa.

```
str = “AATH145 657FVOO”
new_str = str.gsub(/[AEIOU]/){|m_str|
  m_str.downcase
}
puts “#{str}“
puts “#{new_str}“
```

```
>AATH145 657FVOO
>aaTH145 657FVoo
>
```

Sostituzione in una stringa

```
copy_string = string.sub!(pattern, s_string)
```

```
copy_string = string.gsub!(pattern, s_string)
```

I metodi `sub!` e `gsub!` della classe `String` funzionano come `sub` e `gsub`. L'unica differenza è che alterano la stringa invocante *`string`* e ne restituiscono una copia.

Se nessuna sostituzione viene effettuata, allora il metodo restituisce `nil`.

Anche `sub!` e `gsub!` possono essere invocati in associazione a un blocco.

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.gsub!(/(\d+/, “***”)
puts “#{str}“
puts “#{new_str}“
```

Sostituzione in una stringa

```
str = “TH145 657FV”
new_str = str.gsub!(/(\d+/, “***”)
puts “#{str}“
puts “#{new_str}“
```

```
>TH*** ***FV
>TH*** ***FV
>
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 1: se il pattern non contiene raggruppamenti, allora l'array restituito contiene tutte le sottostringhe di *matching*

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 1: se il pattern non contiene raggruppamenti, allora l'array restituito contiene tutte le sottostringhe di *matching*

```
str = "123 145 657 678 999 121"
m_array = str.scan(/\d+\s\d+/)
m_array.each {|m_str|
 puts m_str
}
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 1: se il pattern non contiene raggruppamenti, allora l'array restituito contiene tutte le sottostringhe di *matching*

```
str = "123 145 657 678 999 121"
m_array = str.scan(/\d+\s\d+/)
m_array.each {|m_str|
 puts m_
}
```

```
m_array = ["123 145", "657 678", "999 121"]
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 1: se il pattern non contiene raggruppamenti, allora l'array restituito contiene tutte le sottostringhe di *matching*

```
str = "123 145 657 678 999 121"
m_array = str.scan(/\d+\s\d+/)
m_array.each {|m_str|
 puts m_str
}
```

```
m_array = ["123 145", "657 678", "999 121"]
```

```
>123 145
>657 678
>999 121
>
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 2: se il pattern contiene raggruppamenti, allora ogni elemento di *`array`* è a sua volta un array che contiene i *matching* relativi ai raggruppamenti

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 2: se il pattern contiene raggruppamenti, allora ogni elemento di *`array`* è a sua volta un array che contiene i *matching* relativi ai raggruppamenti

```
str = “123 145 657 678 999 121”
m_array = str.scan(/(\d+)\s(\d+)/)
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 2: se il pattern contiene raggruppamenti, allora ogni elemento di *`array`* è a sua volta un array che contiene i *matching* relativi ai raggruppamenti

```
str = "123 145 657 678 999 121"
m_array = str.scan(/(\d+)\s(\d+)/)
```

Global *matching*

```
array = string.scan(pattern)
```

Il metodo `scan` della classe `String` restituisce un array con tutti i *matching* di *`string`* con *`pattern`*

Caso 2: se il pattern contiene raggruppamenti, allora ogni elemento di *`array`* è a sua volta un array che contiene i *matching* relativi ai raggruppamenti

```
str = "123 145 657 678 999 121"
m_array = str.scan(/(\d+)\s(\d+)/)
```

```
m_array = [["123","145"], ["657","678"], ["999","121"]]
```