Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2016/2017

Linguaggio Ruby: espressioni regolari (parte I)

Docente del laboratorio: Raffaella Rizzi

Espressione regolare

Un'espressione regolare (o *pattern*) è una sequenza di simboli che rappresenta un insieme L di stringhe (*linguaggio* dell'espressione regolare).

Espressione regolare

Un'espressione regolare (o *pattern*) è una sequenza di simboli che rappresenta un insieme L di stringhe (*linguaggio* dell'espressione regolare).

In Ruby con un'espressione regolare E si può:

- verificare se una determinata stringa S ha un *matching* con E, cioé se S contiene come sottostringa uno degli elementi del suo linguaggio L

Espressione regolare

Un'espressione regolare (o *pattern*) è una sequenza di simboli che rappresenta un insieme L di stringhe (*linguaggio* dell'espressione regolare).

In Ruby con un'espressione regolare E si può:

- verificare se una determinata stringa S ha un *matching* con E, cioé se S contiene come sottostringa uno degli elementi del suo linguaggio L
- effettuare in S la sostituzione della sottostringa di *matching* con una nuova sottostringa.

Espressione regolare

Un'espressione regolare (o *pattern*) è una sequenza di simboli che rappresenta un insieme L di stringhe (*linguaggio* dell'espressione regolare).

In Ruby con un'espressione regolare E si può:

- verificare se una determinata stringa S ha un *matching* con E, cioé se S contiene come sottostringa uno degli elementi del suo linguaggio L
- effettuare in S la sostituzione della sottostringa di *matching* con una nuova sottostringa.

Un'espressione regolare è in Ruby un oggetto della classe `Regexp`.

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* //

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/cat/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "cat". Quindi il suo linguaggio L sarà l'insieme:

```
L={"cat"}
```

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/cat/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "cat". Quindi il suo linguaggio L sarà l'insieme:

```
L={"cat"}
```

Tutte le stringhe che contengono `"cat"` come sottostringa hanno quindi *matching* con il *pattern* `/cat/`.

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/cat/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "cat". Quindi il suo linguaggio L sarà l'insieme:

```
L={"cat"}
```

Tutte le stri
sottostringa

`"catch"` ha *matching* con il *pattern* `/cat/`
`"Catch"` non ha *matching* con il *pattern* `/cat/`

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/t a b/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "t a b". Quindi il suo linguaggio L sarà l'insieme:

```
L={"t a b"}
```

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/t a b/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "t a b". Quindi il suo linguaggio L sarà l'insieme:

```
L={"t a b"}
```

Tutte le stringhe che contengono `"t a b"` come sottostringa hanno quindi *matching* con il *pattern* `/t a b/`.

Il letterale

Il letterale di un'espressione regolare (o *pattern*) è una sequenza di simboli racchiusa tra due *slash* `//`

Ad esempio:

```
/t a b/
```

è il letterale dell'espressione regolare che rappresenta la sola stringa "t a b". Quindi il suo linguaggio L sarà l'insieme:

```
L={"t a b"}
```

Tutte le stringhe che contengono `"t a b"` come sottostringa hanno quindi *matching* con il *pattern* `/t a b/`.

`"table"` non ha *matching* con il *pattern* `/t a b/`

Il letterale

Un espressione regolare (in quanto oggetto di classe `Regexp`) può essere assegnata a una variabile

Il letterale

Un espressione regolare (in quanto oggetto di classe `Regexp`) può essere assegnata a una variabile

```
pt = /cat/
puts pt.class
```

Il letterale

Un espressione regolare (in quanto oggetto di classe `Regexp`) può essere assegnata a una variabile

```
pt = /cat/
puts pt.class
```

```
>Regexp
>
```

Il letterale

Regole fondamentali per scrivere un letterale:

→ tutti i simboli, diversi da `. | ( ) [ ] { } + \ ^ $ * ?`, rappresentano se stessi

→ i simboli `. | ( ) [ ] { } + \ ^ $ * ?` non rappresentano se stessi ma servono per rappresentare "altro" (metasimboli)

→ per fare in modo che i simboli `. | ( ) [ ] { } + \ ^ $ * ?` rappresentino se stessi basta anteporre un *backslash* \

Il letterale

Regole fondamentali per scrivere un letterale:

→ tutti i simboli, diversi da `. | ( ) [ ] { } + \ ^ $ * ?`, rappresentano se stessi

→ i simboli `. | ( ) [ ] { } + \ ^ $ * ?` non rappresentano se stessi ma servono per rappresentare "altro" (metasimboli)

→ per fare in modo che i simboli `. | ( ) [ ] { } + \ ^ $ * ?` rappresentino se stessi basta anteporre un *backslash* \

Ad esempio, il pattern

```
/c.a?t/
```

non rappresenta la stringa `"c.a?t"`

Il letterale

Regole fondamentali per scrivere un letterale:

→ tutti i simboli, diversi da `. | ( ) [ ] { } + \ ^ $ * ?`, rappresentano se stessi

→ i simboli `. | ( ) [ ] { } + \ ^ $ * ?` non rappresentano se stessi ma servono per rappresentare "altro" (metasimboli)

→ per fare in modo che i simboli `. | ( ) [ ] { } + \ ^ $ * ?` rappresentino se stessi basta anteporre un *backslash* \

… mentre il pattern

```
/c\.a\?t/
```

rappresenta la stringa `"c.a?t"`

Il letterale

Nel letterale di un pattern si può inserire il costrutto di sostituzione `#{expr}`

Il letterale

Nel letterale di un pattern si può inserire il costrutto di sostituzione `#{expr}`

```
i = 687
pt = /intero = #{i}/
```

Il letterale

Nel letterale di un pattern si può inserire il costrutto di sostituzione `#{expr}`

```
i = 687
pt = /intero = #{i}/
```

La variabile pt contiene il pattern che rappresenta la stringa “`intero = 687`”.

Il letterale

Nel letterale di un pattern si può inserire il costrutto di sostituzione `#{expr}`

```
i = 687
pt = /intero = #{i}/
```

La variabile pt contiene il pattern che rappresenta la stringa “`intero = 687`”. Ciò è equivalente a scrivere:

```
pt = /intero = 687/
```

Operazione di *matching*

L'operatore di *matching* permette di confrontare una stringa con un pattern e il suo simbolo è =~.
L'espressione:

```
string =~ pattern
```

oppure

```
pattern =~ string
```

restituisce l'offset di inizio della prima occorrenza in *`string`* di una delle stringhe rappresentate da *`pattern`*, oppure `nil` se la stringa non ha *matching* con il pattern.

Operazione di *matching*

L'operatore di *matching* permette di confrontare una stringa con un pattern e il suo simbolo è =~.
L'espressione:

```
string =~ pattern
```

oppure

```
pattern =~ string
```

restituisce l'offset di inizio della prima occorrenza in `string` di una delle stringhe rappresentate da

Un'operazione di *matching* può essere usata in un qualsiasi contesto booleano (ad esempio come condizione in un costrutto `if`)

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
```

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
```

```
>9
>
```

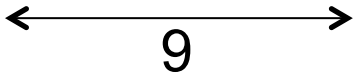
Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
```

```
>9
>
```

“dogs and **cats** and cats”



L’offset di inizio della sottostringa di *matching* coincide con il numero di caratteri che vengono prima di tale sottostringa

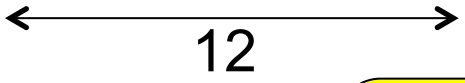
Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
```

```
>9
>
```

“dogs and **cat**s and cats”



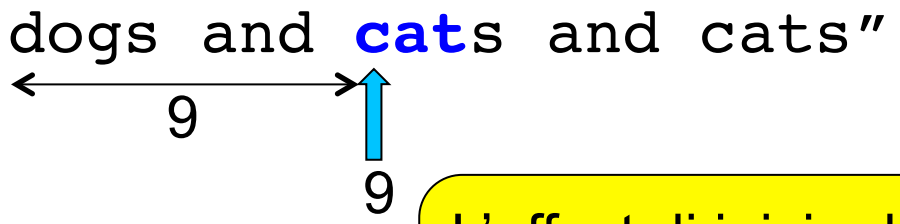
L’offset di fine della sottostringa di *matching* coincide con l’offset di inizio più il numero di caratteri della sottostringa di *matching*

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
```

```
>9
>
```



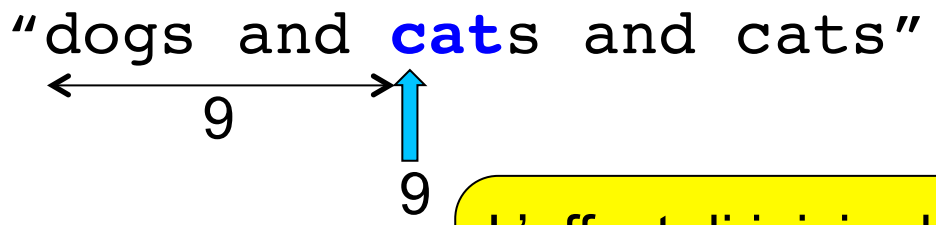
L’offset di inizio della sottostringa di *matching* coincide quindi con la posizione del suo primo simbolo

Operazione di *matching*

Ad esempio:

```
str = “dogs and cats and cats”
pattern = /cat/
puts str[str =~ pattern]
```

```
>c
>
```



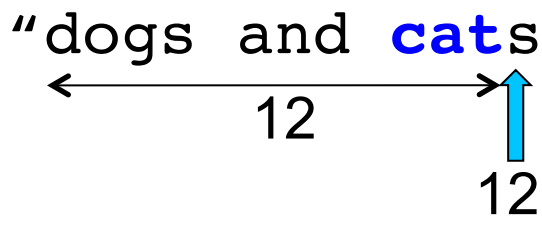
L’offset di inizio della sottostringa di *matching* coincide quindi con la posizione del suo primo simbolo

Operazione di *matching*

Ad esempio:

```
str = “dogs and cats and cats”
pattern = /cat/
puts str[str =~ pattern]
```

```
>c
>
```



L’offset di fine della sottostringa di *matching* coincide quindi con la posizione del primo carattere successivo alla sottostringa di *matching*

Operazione di *matching*

Se cambio il pattern…

```
str = “dogs and cats and cats”
pattern = /Cat/
puts(str =~ pattern)
```

```
>nil
>
```

Operazione di *matching*

Dopo l'esecuzione di un'operazione di *matching* vengono aggiornate le seguenti variabili globali predefinite di Ruby:

- `$&` ➔ contiene la sottostringa di *matching*
- `$`` ➔ contiene il prefisso di stringa che precede la sottostringa di *matching*
- `$'` ➔ contiene il suffisso di stringa che segue la sottostringa di *matching*

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>9
>Match: “cat”
>Before the match: “dogs and “
>After the match: “s and cats”
>
```

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>9
>Match: “cat”
>Before the match: “dogs and “
>After the match: “s and cats”
>
```

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>9
>Match: “cat”
>Before the match: “dogs and “
>After the match: “s and cats”
>
```

Operazione di *matching*

Ad esempio

```
str = “dogs and cats and cats”
pattern = /cat/
puts(str =~ pattern)
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>9
>Match: “cat”
>Before the match: “dogs and “
>After the match: “s and cats”
>
```

I metasimboli

I metasimboli sono simboli che non rappresentano se stessi ma che servono per rappresentare “altre cose”. Precisamente:

- ancore
- classi
- quantificatori
- alternative
- raggruppamenti

Le ancore

Le ancore sono:

- l'inizio di riga → simbolo ^ (cappuccio)
- la fine riga → simbolo `$` (dollaro)
- l'inizio di stringa → `\A`
- la fine di stringa → `\Z` e `\z`
- il confine di parola → `\b`
- il non-confine di parola ("negazione" di `\b`) → `\B`

Le ancore

Le ancore sono:

- l'inizio di riga → simbolo ^ (cappuccio)
- la fine riga → simbolo $ (dollaro)
- l'inizio di stringa → \A
- la fine di stri
- il confine di parola → \b
- il non-confine di parola ("negazione" di \b) → \B

NB: i simboli A, z, Z, b, B con davanti un *backslash* \ diventano metasimboli

Le ancore: il cappuccio ^

Il cappuccio ^ rappresenta l’inizio di una riga

Le ancore: il cappuccio ^

Il cappuccio ^ rappresenta l'inizio di una riga

ad esempio `/cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa\n**cat**aaaa", "aaa**cat**aaa", "aaa**cat**"

Le ancore: il cappuccio ^

Il cappuccio ^ rappresenta l'inizio di una riga

ad esempio `/cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa\n**cat**aaaa", "aaa**cat**aaa", "aaa**cat**"

mentre `/^cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa\n**cat**aaaa"

ma non con "aaa**cat**aaa", "aaa**cat**"

in cui "cat" non compare all'inizio di una riga

Le ancore: il dollaro $

Il dollaro $ rappresenta la fine di una riga

Le ancore: il dollaro $

Il dollaro `$` rappresenta la fine di una riga

ad esempio `/cat/` ha *matching* con le stringhe:

"**cat**aaaa”, "aaaa**cat**\naaaa”, "aaa**cat**aaa", "aaa**cat**”

Le ancore: il dollaro $

Il dollaro `$` rappresenta la fine di una riga

ad esempio `/cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa**cat**\naaaa", "aaa**cat**aaa", "aaa**cat**"

mentre `/cat$/` ha *matching* con le stringhe:

"aaa**cat**", "aaaa**cat**\naaaa"

ma non con "aaa**cat**aaa", "**cat**aaaa"

in cui "cat" non compare alla fine di una riga

Le ancore: \A

`\A` rappresenta l'inizio di una stringa

Le ancore: \A

`\A` rappresenta l'inizio di una stringa

ad esempio `/^cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa\n**cat**aaaa"

Le ancore: \A

`\A` rappresenta l'inizio di una stringa

ad esempio `/^cat/` ha *matching* con le stringhe:

"**cat**aaaa", "aaaa\n**cat**aaaa"

mentre `/\Acat/` ha *matching* con la stringa:

"**cat**aaaa"

ma non con "aaaa\n**cat**aaaa"

in cui "cat" compare all'inizio di una riga ma non all'inizio della stringa

Le ancore: \z

`\z` rappresenta la fine di una stringa

Le ancore: \z

`\z` rappresenta la fine di una stringa

ad esempio `/cat$/` ha *matching* con le stringhe:

"aaaa**cat**", "aaaa**cat**\naaaa", "aaaa**cat**\n"

Le ancore: \z

`\z` rappresenta la fine di una stringa

ad esempio `/cat$/` ha *matching* con le stringhe:

"aaaa**cat**", "aaaa**cat**\naaaa", "aaaa**cat**\n"

mentre `/cat\z/` ha *matching* con la stringa:

"aaa**cat**"

ma non con "aaaa**cat**\naaaa", "aaaa**cat**\n"

in cui "cat" compare alla fine di una riga ma non alla fine della stringa

Le ancore: \Z

`\z` rappresenta la fine di una stringa (eventualmente prima di un *newline*)

Le ancore: \Z

`\Z` rappresenta la fine di una stringa (eventualmente prima di un *newline*)

ad esempio `/cat\z/` ha *matching* con la stringa:

"aaaa**cat**"

ma non con:

"aaaa**cat**\n"

Le ancore: \Z

`\Z` rappresenta la fine di una stringa (eventualmente prima di un *newline*)

ad esempio `/cat\z/` ha *matching* con la stringa:

"aaaa**cat**"

ma non con:

"aaaa**cat**\n"

mentre `/cat\Z/` ha *matching* con le stringhe:

"aaaa**cat**", "aaaa**cat**\n"

Le ancore: \b

`\b` rappresenta un confine di parola

Le ancore: \b

`\b` rappresenta un confine di parola

Parola = sequenza di simboli che possono essere lettere minuscole `a-z`, lettere maiuscole `A-Z`, cifre `0-9` o simbolo *underscore* _

Le ancore: \b

`\b` rappresenta un confine di parola

ad esempio `/\bis/` ha *matching* con la stringa:

"It **is** cat"

ma non con la stringa:

"Th**is** is a cat"

Parola = sequenza di simboli che possono essere lettere minuscole `a-z`, lettere maiuscole `A-Z`, cifre `0-9` o simbolo *underscore* _

Le ancore: \b

`\b` rappresenta un confine di parola

ad esempio `/\bis/` ha *matching* con la stringa:

"It **is** cat"

ma non con la stringa:

"Th**is** is a cat"

E' un confine di parola!!

Parola = sequenza di simboli che possono essere lettere minuscole `a-z`, lettere maiuscole `A-Z`, cifre `0-9` o simbolo *underscore* _

Le ancore: \b

`\b` rappresenta un confine di parola

ad esempio `/\bis/` ha *matching* con la stringa:

"It **is** is cat"

ma non con la stri

"Th**is** is a cat"

Non è un confine di parola!!

Parola = sequenza di simboli che possono essere lettere minuscole `a-z`, lettere maiuscole `A-Z`, cifre `0-9` o simbolo *underscore* `_`

Le ancore: \B

`\B` rappresenta la “negazione” di un confine di parola

Le ancore: \B

`\B` rappresenta la “negazione” di un confine di parola

ad esempio `/\Bis/` ha *matching* con la stringa:

“Th**is** is cat”

ma non con con la stringa:

“It **is** a cat”

Le classi

Una classe rappresenta un insieme di caratteri e viene indicata nel seguente modo:

`[char_list]`

dove *`char_list`* è l'elenco dei caratteri appartenenti alla classe.

Ad esempio:

- `[aeiou]` ➔ classe delle vocali minuscole

Le classi

Una classe rappresenta un insieme di caratteri e viene indicata nel seguente modo:

`[`*`char_list`*`]`

dove *`char_list`* è l'elenco dei caratteri appartenenti alla classe.

Ad esempio:

- `[aeiou]` ➔ classe delle vocali minuscole
- `[AEIOU]` ➔ classe delle vocali maiuscole

Le classi

Una classe rappresenta un insieme di caratteri e viene indicata nel seguente modo:

`[`*`char_list`*`]`

dove *`char_list`* è l'elenco dei caratteri appartenenti alla classe.

Ad esempio:

- `[aeiou]` ➔ classe delle vocali minuscole
- `[AEIOU]` ➔ classe delle vocali maiuscole
- `[.,;:]` ➔ classe di simboli dei punteggiatura (notare che il simbolo `.` rappresenta in questo caso se stesso)

Le classi

Una classe rappresenta un insieme di caratteri e viene indicata nel seguente modo:

`[`*`char_list`*`]`

dove *`char_list`* è l'elenco dei caratteri appartenenti alla classe.

Ad esempio:

- `[.*|?\b]` ➔ classe dei simboli `.` `*` | `?` (notare che in questo caso rappresentano se stessi) e *backspace* `\b` (che in questo caso non è il confine di parola)

Le classi

Una classe rappresenta un insieme di caratteri e viene indicata nel seguente modo:

`[`*`char_list`*`]`

dove *`char_list`* è l'elenco dei caratteri appartenenti alla classe.

Ad esempio:

- `[.*|?\b]` ➔ classe dei simboli `.` `*` | `?` (notare che in questo caso rappresentano se stessi) e *backspace* `\b` (che in questo caso non è il confine di parola)
- `[.*|?\\b]` ➔ classe dei simboli `.` `*` | `?`, *backslash* `\` e lettera minuscola `b`

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole
- `[0-9]` ➔ classe delle cifre da `0` a `9`

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole
- `[0-9]` ➔ classe delle cifre da `0` a `9`
- `[a-zA-Z]` ➔ classe di tutte le lettere

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole
- `[0-9]` ➔ classe delle cifre da `0` a `9`
- `[a-zA-Z]` ➔ classe di tutte le lettere
- `[a-zA-Z0-9_]` ➔ classe dei simboli di parola

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole
- `[0-9]` ➔ classe delle cifre da `0` a `9`
- `[a-zA-Z]` ➔ classe di tutte le lettere
- `[a-zA-Z0-9_]` ➔ classe dei simboli di parola
- `[a\-z]` ➔ classe dei tre simboli `a`, `-` e `z`

Le classi

Il simbolo trattino – all'interno di una classe è il metasimbolo che permette di specificare un intervallo.

Ad esempio:

- `[a-z]` ➔ classe delle lettere minuscole
- `[A-Z]` ➔ classe delle lettere maiuscole
- `[0-9]` ➔ classe delle cifre da `0` a `9`
- `[a-zA-Z]` ➔ classe di tutte le lettere
- `[a-zA-Z0-9_]` ➔ classe dei simboli di parola
- `[a\-z]` ➔ classe dei tre simboli `a`, `-` e `z`
- `[a\-z\[\]]` ➔ classe dei simboli `a`, `-` `z` e parentesi quadre `[]`

Le classi

Il simbolo `^` (messo all'inizio di una classe) serve per negare la lista di caratteri che segue:

`[^`*`char_list`*`]`

è la classe di tutto ciò che non è elencato in *`char_list`*

Ad esempio:

- `[^aeiou]` ➔ classe di tutti i simboli che non sono vocali minuscole

Le classi

Il simbolo ^ (messo all'inizio di una classe) serve per negare la lista di caratteri che segue:

`[^`*`char_list`*`]`

è la classe di tutto ciò che non è elencato in *`char_list`*

Ad esempio:

- `[^aeiou]` ➔ classe di tutti i simboli che non sono vocali minuscole

Attenzione! `[ae^iou]` è la classe delle vocali minuscole e del simbolo cappuccio ^

Le classi

Una classe rappresenta ognuno dei simboli che le appartengono

Ad esempio `/[A-Z]at/` rappresenta le stringhe:

”Cat”, “Bat”, “Rat”, “Fat”

ma non le stringhe ”cat”, “bat”, “rat”, “fat”

mentre `/[a-zA-Z]at/` rappresenta le stringhe:

”Cat”, “Bat”, “Rat”, “Fat”, ”cat”, “bat”, “rat”, “fat”

Oppure `/[A-Z][0-9]at/` rappresenta le stringhe:

”C0at”, “B2at”, “R1at”, “F7at”

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔ metasimbolo `\H`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔ metasimbolo `\H`
- `[␣\t\r\n\f]` ➔ metasimbolo `\s`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔
- `[␣\t\r\n\f]` ➔ metasimbolo `\s`

`\s` → tutto ciò che è spazio

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔ metasimbolo `\H`
- `[␣\t\r\n\f]` ➔ metasimbolo `\s`
- `[^␣\t\r\n\f]` ➔metasimbolo `\S`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔ metasimbolo `\H`
- `[␣\t\r\n\f]` ➔ `\s` → tutto ciò che non è spazio
- `[^␣\t\r\n\f]` ➔metasimbolo `\S`

Le classi

Alcune classi importanti per le quali esiste un metasimbolo equivalente:

- `[0-9]` ➔ metasimbolo `\d`
- `[^0-9]` ➔ metasimbolo `\D`
- `[a-zA-Z0-9_]` ➔ metasimbolo `\w`
- `[^a-zA-Z0-9_]` ➔ metasimbolo `\W`
- `[0-9a-fA-F]` ➔ metasimbolo `\h`
- `[^0-9a-fA-F]` ➔ metasimbolo `\H`
- `[␣\t\r\n\f]` ➔ metasimbolo `\s`
- `[^␣\t\r\n\f]` ➔metasimbolo `\S`
- qualsiasi carattere eccetto `\n` ➔ metasimbolo `.` (punto)

Le classi

Per effettuare l'intersezione tra classi si usa `&&`.

Le classi

Per effettuare l'intersezione tra classi si usa `&&`.

Ad esempio, la classe delle consonanti minuscole è data dalla classe:

```
[a-z&&[^aeiou]]
```

I quantificatori

I quantificatori (di ripetizione) permettono di quantificare il simbolo (o più in generale il raggruppamento) che li precede, e sono:

- l'asterisco `*`
- il simbolo di somma `+`
- il punto di domanda `?`
- il costrutto `{m,n}`
- il costrutto `{m,}`
- il costrutto `{,n}`
- il costrutto `{m}`

I quantificatori: l'asterisco *

L'asterisco `*` specifica 0 o più ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca*t/` rappresenta l'insieme di stringhe composte da `c`, 0 o più ripetizioni di `a` e da `t`, cioè:

{"ct", "cat", "caat", "caaat", …, "caaaaaaaat", …}

I quantificatori: l'asterisco *

L'asterisco `*` specifica 0 o più ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/c[ab]*t/` rappresenta l'insieme di stringhe composte da `c`, 0 o più ripetizioni di `a` oppure di `b`, e da `t`, cioè:

{"ct", "cat", "caat", "caaat", …, "cbt", "cbbt", "cbbbt", …, "cabt", "caababbat", …}

I quantificatori: l'operatore di somma +

L'operatore di somma + specifica 1 o più ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca+t/` rappresenta l'insieme di stringhe composte da `c`, 1 o più ripetizioni di `a` e da `t`, cioè:

{ "cat", "caat", "caaat", …, "caaaaaaaat", …}

cioè l'insieme di `/ca*t/` eccetto la stringa "ct"

I quantificatori: il punto di domanda ?

Il punto di domanda `?` specifica 0 o 1 ripetizione del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca?t/` rappresenta l'insieme di stringhe composte da `c`, 0 o 1 ripetizione di `a` e da `t`, cioè:

{ "ct", "cat"}

I quantificatori: {m,n}

Il costrutto `{m,n}` specifica da `m` a `n` ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca{2,4}t/` rappresenta l'insieme di stringhe composte da `c`, 2 o 3 o 4 ripetizioni di `a` e da `t`, cioè:

{"caat", "caaat", "caaaat"}

I quantificatori: {m,}

Il costrutto `{m,}` specifica `m` o più ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca{2,}t/` rappresenta l'insieme di stringhe composte da `c`, almeno due `a` e da `t`, cioè:

{"caat", "caaat", "caaaat", "caaaaat", …}

I quantificatori: {m,n}

Il costrutto `{,n}` specifica al più `n` ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca{,4}t/` rappresenta l'insieme di stringhe composte da `c`, al più 4 ripetizioni di `a` e da `t`, cioè:

{"ct", "cat", "caat", "caaat", "caaaat"}

I quantificatori: {m}

Il costrutto `{m}` specifica esattamente `m` ripetizioni del simbolo (o del raggruppamento) che lo precede

ad esempio `/ca{4}t/` rappresenta l'insieme di stringhe composte da `c`, 4 ripetizioni di `a` e da `t`, cioè:

{"caaaat"}

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

```
str = “***hello world***”
str =~ /\w+/
puts $&
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

```
str = “***hello world***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

```
str = “***hello world***”
str =~ /\w+\s\w+/
puts $&
```

```
>hello world
>
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

```
str = “***hello           world***”
str =~ /\w+\s\w+/
puts $&
```

```
>nil
>
```

Qualche esempio…

```
str = “***hello***”
str =~ /\w+/
puts $&
```

```
>hello
>
```

```
str = “***hello              world***”
str =~ /\w+\s+\w+/
puts $&
```

```
>hello              world
>
```

Qualche esempio…

```
str = “***hello***”
str =~ /.+/
puts $&
```

Qualche esempio…

```
str = “***hello***”
str =~ /.+/
puts $&
```

```
>***hello***
>
```

Qualche esempio…

```
str = “***hello\n***”
str =~ /.+/
puts $&
```

```
>***hello
>
```

Qualche esempio…

```
str = "***hello\n***"
str =~ /.+/
puts $&
```

```
>***hello
>
```

Il metasimbolo `.` rappresenta qualsiasi carattere diverso da `\n`

Riprendiamo l'operazione di *matching*…

L'operatore di *matching* permette di confrontare una stringa con un pattern e il suo simbolo è =~.
L'espressione:

```
string =~ pattern
```

oppure

```
pattern =~ string
```

restituisce l'offset di inizio della prima occorrenza in `string` di una delle stringhe rappresentate da `pattern`, oppure `nil` se la stringa non ha *matching* con il pattern.

Riprendiamo l'operazione di *matching*…

L'operatore di *matching* permette di confrontare una stringa con un pattern e il suo simbolo è =~.
L'espressione:

```
string =~ pattern
```

oppure

```
pattern =~ string
```

restituisce l'offset di inizio della prima occorrenza in `string` di una delle stringhe rappresentate da `pattern`, oppure `nil` se la stringa non ha *matching* con il pattern.

La sottotringa di *matching* si estende il più possibile a destra (il funzionamento è *greedy*)

Riprendiamo l'operazione di *matching*…

Funzionamento *greedy*

La stringa `string` viene scandita da sinistra a destra (dalla prima all'ultima posizione).

Per ogni posizione i, vengono considerate tutte le sottostringhe che iniziano in i a partire da quella di lunghezza massima (cioè a partire dal suffisso che inizia in posizione i). La prima sottostringa (che si incontra) che appartiene al linguaggio del pattern, è la sottostringa di *matching*.

Qualche esempio…

```
str = “bbbcaaaaaaaat”
pattern = /ca+/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

Qualche esempio…

```
str = “bbbcaaaaaaaat”
pattern = /ca+/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>Match: “caaaaaaaa”
>Before the match: “bbb”
>After the match: “t”
>
```

Qualche esempio…

```
str = “bbbcaaaaaaaat”
pattern = /ca+/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>Match: “caaaaaaaa”
>Before the match: “bbb”
>After the match: “t”
>
```

Operazione di *matching*

Per modificare il comportamento *greedy* del matching si deve usare il simbolo **?** subito dopo il quantificatore.

```
str = “bbbcaaaaaaaat”
pattern = /ca+?/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

Operazione di *matching*

Per modificare il comportamento *greedy* del matching si deve usare il simbolo **?** subito dopo il quantificatore.

```
str = “bbbcaaaaaaaat”
pattern = /ca+?/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>Match: “ca”
>Before the match: “bbb”
>After the match: “aaaaaaat”
>
```

Operazione di *matching*

Per modificare il comportamento *greedy* del matching si deve usare il simbolo **?** subito dopo il quantificatore.

```
str = "bbbcaaaaaaaat"
pattern = /ca+?/
str =~ pattern
puts "Match: \"#{$&}\""
puts "Before the match: \"#{$`}\""
puts "After the match: \"#{$'}\""
```

```
>Match: "ca"
>Before the match: "bbb"
>After the match: "aaaaaaat"
>
```

Le alternative

Il simbolo | permette di specificare due pattern alternativi.

Ad esempio:

`/ab|cd/` rappresenta l'insieme {"ab", "cd"}

Attenzione! Il simbolo | viene "valutato" per ultimo.

Ad esempio, `/cane nero|bianco gatto/` rappresenta solo le stringhe "cane nero" e "bianco gatto" ma non le stringhe "cane bianco" e "nero gatto".

Ad esempio…

```
str = “aaacdbbbbbabaaaa”
pattern = /ab|cd/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

Ad esempio…

```
str = “aaacdbbbbbabaaaa”
pattern = /ab|cd/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>Match: “cd”
>Before the match: “aaa”
>After the match: “bbbbbabaaaa”
>
```

Ad esempio…

```
str = “aaacdbbbbbabaaaa”
pattern = /ab|cd/
str =~ pattern
puts “Match: \”#{$&}\””
puts “Before the match: \”#{$`}\””
puts “After the match: \”#{$’}\””
```

```
>Match: “cd”
>Before the match: “aaa”
>After the match: “bbbbbabaaaa”
>
```

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

Scopo 1: associare un quantificatore a “un qualcosa” di più lungo di un solo simbolo.

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 1: associare un quantificatore a “un qualcosa” di più lungo di un solo simbolo.

Ad esempio,

`/cab+/` rappresenta l’insieme {“cab”, “cabb”, “cabbb”, “cabbbb”, …}

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 1: associare un quantificatore a “un qualcosa” di più lungo di un solo simbolo.

Ad esempio,

`/cab+/` rappresenta l’insieme {“cab”, “cabb”, “cabbb”, “cabbbb”, …}

mentre:

`/c(ab)+/` rappresenta l’insieme {“cab”, “cabab”, “cababab”, “cababbab”, …}

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

Scopo 2: alterare la precedenza del simbolo | nelle alternative.

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 2: alterare la precedenza del simbolo `|` nelle alternative.

Ad esempio,

`/cane nero|bianco/` rappresenta l'insieme {"cane nero", "bianco"}

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 2: alterare la precedenza del simbolo `|` nelle alternative.

Ad esempio,

`/cane nero|bianco/` rappresenta l'insieme {"cane nero", "bianco"}

mentre

`/cane (nero|bianco)/` rappresenta l'insieme {"cane nero", "cane bianco"}

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

I *matching* relativi ai raggruppamenti (eventualmente) presenti nel pattern vengono recuperati da sinistra a destra. Per riferirsi ad essi si devono usare:

- le variabili globali predefinite `$1`, `$2`, `$3`, …, `$n` se il riferimento avviene all'esterno del pattern
- `\1`, `\2`, `\3`, …, `\n` se il riferimento avviene all'interno del pattern

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto cane”
pattern = /(\w+)\s(\w+)/
str =~ pattern
puts $1
puts $2
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto cane”
pattern = /(\w+)\s(\w+)/
str =~ pattern
puts $1
puts $2
```

```
>gatto
>cane
>
```

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto cane”
pattern = /(\w+)\s(\w+)/
str =~ pattern
puts $1
puts $2
```

```
>gatto
>cane
>
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto cane”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: \”#{$&}\””
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto cane”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: \”#{$&}\””
```

Non c’è *matching* tra stringa e pattern in quanto `\1` nel pattern si riferisce alla parte catturata dal raggruppamento a sinistra ed equivale quindi alla ripetizione di `gatto` anche a destra

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto gatto”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: ^#{$&}^”
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “gatto gatto”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: ^#{$&}^”
```

```
>gatto gatto
>
```

In questo caso c’è *matching* tra stringa e pattern

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “cane cane”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: ^#{$&}^”
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “cane cane”
pattern = /(\w+)\s\1/
str =~ pattern
puts “Match: ^#{$&}^”
```

```
>cane cane
>
```

Anche in questo caso c’è *matching* tra stringa e pattern

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “Mississippi”
pattern = /(\w+)\1/
str =~ pattern
puts $1
puts “Match: ^#{$&}^”
```

I raggruppamenti

Le parentesi tonde `( )` permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “Mississippi”
pattern = /(\w+)\1/
str =~ pattern
puts $1
puts “Match: \”#{$&}\””
```

```
>iss
>Match: “ississ”
>
```

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “Mississippi”
pattern = /(\w+)\1/
str =~ pattern
puts $1
puts “Match: \”#{$&}\””
```

```
>iss
>Match: “ississ”
>
```

I raggruppamenti

Le parentesi tonde ( ) permettono di specificare raggruppamenti.

Scopo 3: recuperare parti della sottostringa di *matching* (*back reference*).

```
str = “Mississippi”
pattern = /(\w+)\1/
str =~ pattern
puts $1
puts “Match: \”#{$&}\””
```

```
>iss
>Match: ”ississ”
>
```

Operazione di *matching*

L'operatore di *matching* `!~` permette di confrontare una stringa con un pattern

```
string !~ pattern
```

```
pattern !~ string
```

e viene restituito `true` se non esiste un *matching*, altrimenti viene restituito `false`.