Les Expressions Régulières (Regular Expressions - RegEx)

Eric BERTHOMIER

eric.berthomier@free.fr

27 janvier 2016

Ce cours est une adaptation libre de "Regular Expressions in Python" écrit par John R Woodward.

Définition d'une expression régulière

- Regular expression : (abbreviated regex or regexp) une séquence de recherche permettant de réaliser des opérations de type 'chercher et remplacer' sur les chaînes de caractères (strings).

Définition d'une expression régulière

- Regular expression : (abbreviated regex or regexp) une séquence de recherche permettant de réaliser des opérations de type 'chercher et remplacer' sur les chaînes de caractères (strings).
- Chaque caractère dans une expression régulière est
  - soit interprété comme un méta-caractère (donc avec une signification sépciale)
  - soit comme un caractère littéral

Définition d'une expression régulière

- Regular expression : (abbreviated regex or regexp) une séquence de recherche permettant de réaliser des opérations de type 'chercher et remplacer' sur les chaînes de caractères (strings).
- Chaque caractère dans une expression régulière est
  - soit interprété comme un méta-caractère (donc avec une signification sépciale)
  - soit comme un caractère littéral
- Le but des expressions régulières est de savoir si une chaîne de caractères données correspond ou non à une séquence donnée (pattern).

Liste des métacaractères

- `.`
- `+`
- `?`
- `*`
- `^`
- `$`

- `[...]`
- `[^...]`
- `|`
- `()`
- `{m,n}`

. (point)

- Remplace n'importe quel caractère unique (1 caractère).
- Dépendant des plateformes, le . peut ou non remplacer le caractère fin de ligne (0x0A / 0x0D)
- Placé entre [ ] le point représente alors un .
- Il est aussi possible de protéger son interprétation par \.

. (point)

- Remplace n'importe quel caractère unique (1 caractère).
- Dépendant des plateformes, le . peut ou non remplacer le caractère fin de ligne (0x0A / 0x0D)
- Placé entre [ ] le point représente alors un .
- Il est aussi possible de protéger son interprétation par \.

Exemple d'utilisation de .

`a.c` permettra de trouver la chaîne de caractères "abc", "adc", ...
mais ne permetta pas de touver seulement "a", ".", ou "c".

Exemple

regexHello.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

if re.search(r".....", string1):
    print ("RegEx 1 : " + string1 + " has length >= 5")

if re.search(r"....[.]", string1):
    print ("RegEx 2 : " + string1 + " has length >= 5 and ends with a .")

if re.search(r"....\.", string1):
    print ("RegEx 3 : " + string1 + " has length >= 5 and ends with a .")
```

regexHello.txt

```
RegEx 1 : Hello, world. has length >= 5
RegEx 2 : Hello, world. has length >= 5 and ends with a .
RegEx 3 : Hello, world. has length >= 5 and ends with a .
```

+ (plus)

Teste si l'élément qui le précède est présent **une** ou **plusieurs** fois.

+ (plus)

Teste si l'élément qui le précède est présent **une** ou **plusieurs** fois.

Exemple d'utilisation de +

ab+c validera les chaînes "abc", "abbc", "abbbc", et ainsi de suite mais ne validera pas "ac".

+ (plus)

Teste si l'élément qui le précède est présent **une** ou **plusieurs** fois.

Exemple d'utilisation de +

ab+c validera les chaînes "abc", "abbc", "abbbc", et ainsi de suite mais ne validera pas "ac".

regexHelloPlus.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

if re.search(r"l+", string1):
    print (string1 + " contient une ou plusieurs lettres l consecutives.")
```

regexHelloPlus.txt

```
Hello, world. contient une ou plusieurs lettres l consecutives.
```

? (point d'interrogation)

Teste si l'élément qui le précède est présent **zéro** ou **une** fois.

? (point d'interrogation)

Teste si l'élément qui le précède est présent **zéro** ou **une** fois.

regexHelloPi.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

if re.search(r"H.?e", string1):
    print ("Il y a un 'H' et un 'e' separe par 0-1 caracteres (Ex: HeHoe)")
```

regexHelloPi.txt

```
Il y a un 'H' et un 'e' separe par 0-1 caracteres (Ex: HeHoe)
```

* (étoile)

Teste si l'élément qui le précède est présent **zéro** ou **plusieurs** fois.

* (étoile)

Teste si l'élément qui le précède est présent **zéro** ou **plusieurs** fois.

Exemple d'utilisation de *

- `ab*c` validera les chaînes "ac", "abc", "abbbc" ...
- `[xyz]*` validera "", "x", "y", "z", "zx", "zyx", "xyzzy", et ainsi de suite.
- `(ab)*` validera "", "ab", "abab", "ababab" ...

* (étoile)

Teste si l'élément qui le précède est présent **zéro** ou **plusieurs** fois.

Exemple d'utilisation de *

- ab*c validera les chaînes "ac", "abc", "abbbc" ...
- [xyz]* validera "", "x", "y", "z", "zx", "zyx", "xyzzy", et ainsi de suite.
- (ab)* validera "", "ab", "abab", "ababab" ...

regexHelloStar.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

if re.search(r"e(ll)*o", string1):
    print ("'e' suivi de 0 ou plusieurs 'll' suivi de 'o' (eo, ello, ellllo)")
```

regexHelloStar.txt

```
'e' suivi de 0 ou plusieurs 'll' suivi de 'o' (eo, ello, ellllo)
```

^(accent circonflexe)

Teste le début de ligne (fichier texte) ou d'une chaîne de caractère.

^(accent circonflexe)

Teste le début de ligne (fichier texte) ou d'une chaîne de caractère.

regexHelloCirc.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world"

if re.search(r"^Hell", string1):
    print (string1, " est une ligne ou une chaine de caracteres commencant par 'Hell'")
```

regexHelloCirc.txt

```
Hello, world est une ligne ou une chaine de caracteres commencant par 'Hell'
```

$ (dollar)

Teste la fin de ligne (fichier texte) ou d'une chaîne de caractère.

$ (dollar)

Teste la fin de ligne (fichier texte) ou d'une chaîne de caractère.

regexHelloDollar.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world"

if re.search(r"rld$", string1):
    print (string1, " est une ligne ou une chaine de caracteres se terminant par 'rld'")
```

regexHelloDollar.txt

```
Hello, world est une ligne ou une chaine de caracteres se terminant par 'rld'
```

[] (crochets)

Teste un caractère contenu entre les crochets.

[] (crochets)

Teste un caractère contenu entre les crochets.

Exemple d'utilisation de []

- [abc] validera "a", "b", ou "c".
- [a-z] specifie l'ensemble des caractères minuscules de "a" à "z".

[] (crochets)

Teste un caractère contenu entre les crochets.

Exemple d'utilisation de []

- `[abc]` validera "a", "b", ou "c".
- `[a-z]` specifie l'ensemble des caractères minuscules de "a" à "z".

Ces formes peuvent être combinées

- `[abcx-z]` validera "a", "b", "c", "x", "y" ou "z", comme le ferait `[a-cx-z]`.

Spécificités des [] (crochets)

Les caractères - et [

- Le caractère - est traîté comme un caractère littéral si il est le dernier caractère ou le premier de l'ensemble (après le ^, si ce dernier est présent) : [abc-], [-abc].
- Le caractère] peut être inclus dans l'ensemble si c'est le premier caractère (après le ^) : `[]abc]`.

Spécificités des [] (crochets)

Les caractères - et [

- Le caractère - est traîté comme un caractère littéral si il est le dernier caractère ou le premier de l'ensemble (après le ^, si ce dernier est présent) : [abc-], [-abc].
- Le caractère] peut être inclus dans l'ensemble si c'est le premier caractère (après le ^) : `[]abc]`.

Le caractère antislash (\)

Il n'est pas possible d'utiliser le caractère de protection antislash (\) pour protéger un caractère.

Exemple d'utilisation de []

regexHelloBracket.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

if re.search(r"[aeiou]+", string1):
    print (string1, " contient au moins une voyelle.")
```

regexHelloBracket.txt

```
Hello, world. contient au moins une voyelle.
```

[^...]

Vérifie l'**absence** d'un caractère dans l'ensemble défini entre [].

[^...]

Vérifie l'**absence** d'un caractère dans l'ensemble défini entre [].

Exemple d'utilisation de []

- `[^abc]` vérifie que tous les caractères **autre que** "a", "b", or "c".
- `[^a-z]` vérfie l'absence de tout caractère minuscule de "a" à "z".

[^...]

Vérifie l'**absence** d'un caractère dans l'ensemble défini entre [].

Exemple d'utilisation de []

- `[^abc]` vérifie que tous les caractères **autre que** "a", "b", or "c".
- `[^a-z]` vérfie l'absence de tout caractère minuscule de "a" à "z".

Comme auparavent, ces formes peuvent être combinées.

Exemple d'utilisation de [^...]

regexHelloNotBracket.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello World\n"

if re.search(r"[^abc]", string1):
    print ( string1 + " contient au moins un caractere autre que a, b, et c" )
```

regexHelloNotBracket.txt

```
Hello World
 contient au moins un caractere autre que a, b, et c
```

Choix alternatifs (. . . | . . .)

regexHelloOrBracket.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello World."

if re.search(r"(Hello|Hi|Pogo)", string1):
    print ( 'Au moins une des chaines "Hello", "Hi" ou "Pogo" est contenue dans ' + string1)
```

regexHelloOrBracket.txt

```
Au moins une des chaines "Hello", "Hi" ou "Pogo" est contenue dans Hello World.
```

Sous-Expression ()

() définit une sous-expression. La chaîne ainsi trouvée peut être réutilisée par la suite. Une telle chaîne est nommée groupe.

Sous-Expression ()

() définit une sous-expression. La chaîne ainsi trouvée peut être réutilisée par la suite. Une telle chaîne est nommée groupe.

regexHelloGroup.py

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-

import re

string1 = "Hello, world."

m_obj = re.search(r"(H..).(o..)(...)",string1)

if re.search(r"(H..).(o..)(...)",string1):
    print ("Chaine valide : '" + m_obj.group(1) + "' and '" + m_obj.group(2) + "' and '" + m_obj
          .group(3)+ "'")
```

regexHelloGroup.txt

```
Chaine valide : 'Hel' and 'o, ' and 'wor'
```

{m,n}

{m,n} valide si l'élément qui précède est indiqué au moins m fois et au plus n fois.

{m,n}

{m,n} valide si l'élément qui précède est indiqué au moins m fois et au plus n fois. Par exemple, a{3,5} valide seulement les chaînes "aaa", "aaaa", and "aaaaa".

À vous de jouer !