


Cours	Cours	
BTS SNIR	Python	
2 ^{ème} année	Les expressions régulières	

Les Expressions Régulières en Python

Les expressions régulières, également connues sous le nom de regex, fournissent un moyen puissant de rechercher et de manipuler des chaînes de caractères. En Python, le module **re** offre des fonctionnalités pour travailler avec des expressions régulières.

1. Importer le module re

Avant d'utiliser des expressions régulières, assurez-vous d'importer le module re :

```
import re
```

2. Créer une Expression Régulière

Une expression régulière est créée à l'aide de la fonction `compile` du module re :

```
pattern = re.compile(r'motif')
```

Le préfixe **r** indique une chaîne brute, ce qui signifie que les caractères d'échappement (comme `\n` pour une nouvelle ligne) ne sont pas interprétés.

3. Rechercher un Motif

La méthode `search` recherche un motif dans une chaîne :

```
result = pattern.search("Texte où le motif peut être trouvé.")
if result:
    print("Motif trouvé :", result.group())
else:
    print("Motif non trouvé.")
```

4. Correspondance au Début de la Chaîne

Utilisez `match` pour vérifier si le motif correspond au début de la chaîne :

```
result = pattern.match("motif au début de la chaîne")
if result:
    print("Motif trouvé au début :", result.group())
else:
    print("Motif non trouvé au début.")
```

5. Trouver Toutes les Correspondances

La méthode `findall` retourne toutes les occurrences d'un motif dans une chaîne sous forme de liste :

```
matches = pattern.findall("Texte avec plusieurs occurrences du motif.")
print("Occurrences trouvées :", matches)
```

6. Remplacer un Motif

La méthode `sub` remplace toutes les occurrences d'un motif par une autre chaîne :

```
new_text = pattern.sub("remplacement", "Texte avec motif à remplacer.")
print("Nouveau texte :", new_text)
```

7. Drapeaux (Flags)

Les drapeaux permettent de modifier le comportement des expressions régulières. Par exemple, `re.IGNORECASE` rend la recherche insensible à la casse :

```
pattern = re.compile(r'motif', re.IGNORECASE)
result = pattern.search("Motif en majuscules ou en minuscules.")
```

8. Caractères Spéciaux

Certains caractères ont une signification spéciale en regex. Pour les utiliser littéralement, échappez-les avec \ :

```
pattern = re.compile(r'\d+') # Recherche de un ou plusieurs chiffres
```

9. Classes de Caractères

Les classes de caractères permettent de spécifier un ensemble de caractères possibles. Par exemple, [aeiou] correspond à n'importe quelle voyelle.

```
vowels_pattern = re.compile(r'[aeiou]')
```

10. Quantificateurs

Les quantificateurs spécifient le nombre d'occurrences d'un motif. Par exemple, * signifie zéro ou plusieurs occurrences, et + signifie une ou plusieurs occurrences.

```
pattern = re.compile(r'\d{3}-\d{2}-\d{4}') # Recherche de  
numéros de sécurité sociale
```

Concepts Avancés des Expressions Régulières en Python

1. Groupes de Captures

Les groupes de captures permettent de capturer des parties spécifiques d'une correspondance. Utilisez des parenthèses pour définir un groupe de capture :

```
pattern = re.compile(r'(\d{3})-(\d{2})-(\d{4})') # Numéro de
sécurité sociale avec groupes
result = pattern.search("Numéro de sécurité sociale : 123-45-
6789")
if result:
    print("Groupe 1 :", result.group(1))
    print("Groupe 2 :", result.group(2))
    print("Groupe 3 :", result.group(3))
```

2. Correspondance Alternative

L'opérateur | permet de spécifier des alternatives dans un motif :

```
pattern = re.compile(r'python|java') # Recherche de "python"
ou "java"
result = pattern.search("J'aime python et java.")
if result:
    print("Motif trouvé :", result.group())
```

3. Ancres

Les ancres définissent des positions spécifiques dans une chaîne. ^ correspond au début de la chaîne et \$ à la fin :

```
pattern = re.compile(r'^Début.*Fin$') # Correspond à une
chaîne qui commence par "Début" et se termine par "Fin"
```

4. Lookahead et Lookbehind

Les assertions lookahead ((?=...)) et lookbehind ((?<=...)) permettent de spécifier des conditions devant ou derrière la correspondance sans les inclure dans la correspondance elle-même :

```
pattern = re.compile(r'\d+(?=%)') # Correspond à un nombre  
suivi de %
```

5. Modificateurs en Ligne

Les modificateurs en ligne permettent de spécifier des options directement dans le motif :

```
pattern = re.compile(r'(?i)motif', re.DOTALL) # Recherche  
insensible à la casse avec prise en compte des sauts de ligne
```

En plus des modificateurs mentionnés précédemment, voici quelques autres modificateurs utiles :

- `re.MULTILINE`: Permet de faire correspondre le début (^) et la fin (\$) de chaque ligne, plutôt que de l'ensemble de la chaîne.
- `re.VERBOSE`: Permet d'écrire des expressions régulières de manière plus lisible en ajoutant des espaces et des commentaires.

```
pattern = re.compile(r'''  
^\d{3}    # Trois chiffres au début  
-        # Trait d'union  
\d{2}    # Deux chiffres au milieu  
-        # Trait d'union  
\d{4}    # Quatre chiffres à la fin  
''', re.VERBOSE)
```

6. Fonction finditer

La fonction finditer retourne un itérable d'objets correspondants, permettant d'itérer sur toutes les occurrences :

```
pattern = re.compile(r'\d+')
for match in pattern.finditer("123 et 456"):
    print("Occurrence trouvée :", match.group())
```

7. Fonction split

La fonction split divise une chaîne en utilisant un motif comme séparateur :

```
pattern = re.compile(r'\s+')
result = pattern.split("Ceci est une phrase.")
print("Résultat de la division :", result)
```

8. Références arrière dans les groupes

Vous pouvez faire référence à des groupes précédemment capturés avec \n où n est le numéro du groupe.

```
pattern = re.compile(r'(\w+) \1') # Correspond à des mots
répétés, comme "chat chat" ou "python python"
```