Notes de cours
Semaine 25

Cours Turing

1 Expressions régulières

Les expressions régulières sont un moyen de décrire des ensembles de chaînes de caractères de façon relativement simple et concise. Les expressions régulières décrivent des *patterns* de chaînes de caractères. Des algorithmes permettent ensuite de vérifier si une chaîne de caractères se conforme à une expression régulière, ou encore de trouver des occurrences d'une expression régulière au sein d'un texte.

Il existe de nombreuses variantes d'expressions régulières, de nombreux différents dialectes. Dans ce document, nous allons nous intéresser à la variante adoptée par Python, le langage d'implémentation de notre projet de compilateur.

Ce document n'est qu'une courte introduction à l'utilisation d'expressions régulières en Python. Pour plus de détails, n'hésitez pas à consulter cette excellente ressource en ligne sur le site officiel de Python : `https://docs.python.org/fr/3/howto/regex.html`

1.1 Utilisation en Python

En Python, les expressions régulières sont rendues disponibles via le module `re` (pour *regular expressions*). Ci-dessous vous trouverez un exemple d'utilisation des expressions régulières en Python.

```
import re

regex = re.compile(r"[a-z]+e\b")

results = regex.findall("un petit texte d'exemple.")
for result in results:
    print(result)
```

Le programme commence par créer une expression régulière en appelant la méthode `compile` du module `re` sur une chaîne de caractères appelée un `pattern`. Le langage utilisé pour écrire de tels patterns est le sujet de la majorité de ce document. Le pattern de l'expression régulière spécifie un ensemble de chaînes de caractères à accepter. Dans ce cas précis, il s'agit de toutes les chaînes formées d'au moins une lettre minuscule puis d'un *e* en fin de mot. La méthode `findall` est ensuite utilisée afin de trouver toutes les occurrences de l'expression régulière dans un texte. Chaque occurence est ensuite affichée. Lorsqu'il sera exécuté, le programme affichera le résultat suivant :

```
texte
exemple
```

La méthode `findall` n'est pas la seule méthode disponible sur les expressions régulières en Python. On trouve notamment la méthode `match` pour vérifier si une expression régulière apparaît à une position donnée (par défaut au début), ou encore la méthode `search` pour trouver la prochaine occurrence d'une expression régulière. En cas de succès, le résultat de ces deux méthodes est un objet qui contient non seulement la chaîne acceptée par l'expression régulière, mais aussi sa position au sein du texte.

```
import re

regex = re.compile(r"[a-z]+e\b")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(9, 14), match='texte'>

print(regex.match("un petit texte d'exemple."))
# Affiche: None

print(regex.match("un petit texte d'exemple.", 17))
# Affiche: <re.Match object; span=(17, 24), match='exemple'>
```

Raw strings en Python

Les plus attentifs d'entrevous auront certainement remarqué que dans l'exemple donné la chaîne de caractère Python qui définit le pattern de l'expression régulière était préfixée d'un `r`. Dans le jargon Python, on dit qu'il s'agit d'une *raw string*. Afin de comprendre ce qu'est une *raw string*, il nous faut discuter un instant de la notation de base des chaînes de caractères en Python.

Dans une chaîne de caractères Python normale, il est possible d'utiliser la barre oblique inversée (le *backslash*) afin d'entrer des caractères comme des sauts de ligne `\n` ou encore des apostrophes même si elles correspondent à celles utilisées pour délimiter la chaîne.

```
"Il dit:\n\"Hello tout le monde!\""
```

À cause de cette utilisation spéciale de la barre oblique inversée, si l'on souhaite l'écrire dans une chaîne Python normale, il faut aussi l'échapper à l'aide d'une autre telle barre oblique inversée, comme dans l'exemple ci-dessous :

```
"Mon symbole préféré est: \\"
```

Normalement, cette façon de procéder est très pratique, car elle permet d'ajouter facilement des caractères parfois plus difficiles à saisir. Or, dans notre cas précis, cette façon de procéder est plus dérangeante qu'aidante, pour une raison bien précise : l'utilisation fréquente de la barre oblique inversée.

Comme nous allons le voir, nous ferons de nombreuses fois usage de la barre oblique inversée dans nos expressions régulières. Ce caractère joue en effet un rôle important dans la syntaxe de

ces expressions. Afin de ne pas devoir échapper ces caractères à chaque utilisation, nous allons utiliser les *raw strings* de Python. Cette façon de noter les chaînes de caractères ne permet pas d'entrer des caractères échappés et laisse ainsi intact le sens normal de la barre oblique inversée, comme dans l'exemple suivant, une chaîne bien formée en Python :

```
r"Mon symbole préféré est: \"
```

1.2 Expressions simples

Regardons à présent comment écrire nos propres expressions régulières, en commençant par la façon la plus simple, qui consiste à directement indiquer la séquence de caractères à accepter.

```
import re

regex = re.compile(r"petit")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(3, 8), match='petit'>
```

Dans l'exemple ci-dessous, l'expression régulière spécifie que la chaîne à trouver doit être exactement `petit`. Cette chaîne est trouvée en position 3 du texte.

À quelques exceptions près, il suffit ainsi de tapper un caractère pour indiquer que l'on souhaite ce caractère à l'endroit précisé. Les caractères qui font exception à cette règle sont :

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

Tous ces caractères ont un sens particulier et permettent de décrire des expressions régulières plus complexes, comme nous allons pouvoir le voir dans la suite de ce document. Par exemple, le point, au lieu d'indiquer que l'on souhaite littéralement le caractère `.`, indique que l'on souhaite n'importe quel caractère (à l'exception du saut de ligne).

Pour spécifier que l'on souhaite bien un de ces caractères spéciaux, il suffit de préfixer le caractère d'un \ (barre oblique inversée, *backslash*), comme dans l'exemple ci-dessous :

```
import re

regex = re.compile(r"\.")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(24, 25), match='.'>

# Comparez avec:
regex = re.compile(r".")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(0, 1), match='u'>
```

1.3 Classes de caractères

Parfois, à la place de spécifier exactement un caractère bien précis, on souhaitera permettre un ensemble de caractères différents. Ceci peut être réalisé en listant les différents caractères possibles entre crochets (`[` et `]`), comme dans l'exemple suivant :

```
regex = re.compile(r"[aeiouy]x[aeiouy]")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(17, 20), match='exe'>
```

Le programme ci-dessus cherche la première occurrence d'un `x` entouré de deux voyelles (minuscules), potentiellement différentes. Pour spécifier que l'on souhaite une voyelle à un endroit donné, on utilise dans cet exemple la notation basée sur les crochets vue juste à l'instant.

Classes inverses

Il est aussi possible de spécifier que l'on désire un caractère qui **ne figure pas** parmi les caractères donnés en mettant directement après le crochet ouvrant le caractère `^`. Ainsi, dans l'exemple suivant, on indique que l'on recherche une voyelle, suivie d'un `x`, suivis d'un caractère qui n'est pas une voyelle.

```
regex = re.compile(r"[aeiouy]x[^aeiouy]")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(10, 13), match='ext'>
```

Intervalles

Il est aussi possible d'inscrire un intervalle de symboles à la place d'explicitement lister tous les éléments, et ce en mettant un tiret (`-`) entre les deux bornes de l'intervalle. Cette fonctionnalité est utilisée dans le programme ci-dessous. Le programme cherche dans le texte donné la première occurence d'une lettre minuscule ou majuscule :

```
import re

regex = re.compile(r"[a-zA-Z]")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(0, 1), match='u'>
```

Notez que le tiret (`-`) n'est pas un caractère spécial, malgré son rôle particulier dans la notation utilisée pour les intervalles. Pour dénoter un simple tiret dans une expression régulière, on note simplement `-`. La seul restriction est que, au sein d'une classe, le tiret doit apparaître en premier ou en dernier afin de représenter le caractère `-`. Ainsi, l'expression régulière `[abc-]` représente un caractère parmi `a`, `b`, `c` ou `-`.

Classes nommées

Certaines classes de caractères fréquemment utilisées se sont vues dotées d'un raccourci de notation. Vous trouverez les principales ci-dessous :

`.`	N'importe quel caractère à l'exception du saut de ligne.
`\d`	N'importe quel chiffre.
`\s`	N'importe quel caractère d'espacement (espace, tabulation, saut de ligne, etc.).
`\w`	N'importe quel caractère alphanumérique, ou le tiret bas (`_`).

1.4 Répétitions

Dans une expression régulière, il est possible d'indiquer qu'une partie de l'expression peut, voire doit, se répéter à plusieurs reprises. Il existe pour ceci plusieurs opérateurs que l'on va passer en revue. Le plus basique est l'opérateur `*` (appelé *étoile de Kleene*).

```
import re

regex = re.compile(r"[a-z]*t")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(3, 8), match='petit'>
```

Dans l'exemple ci-dessus, l'étoile de Kleene permet de spécifier que l'on recherche une sous-chaîne formée d'un nombre arbitraire de minuscules suivi d'un `t`. Notez, et c'est important, que l'on accepte que les minuscules ne correspondent pas toutes à la même lettre. Ainsi, dans l'exemple, la correspondance trouvée dans le texte est la chaîne `petit`, qui est formée de plusieurs différentes lettres minuscules avant le dernier `t`.

De même, notez que l'étoile de Kleene a un comportement glouton : le plus possible de caractères seront inclus dans la répétition, sauf s'il s'avère par la suite que cela empêche l'acceptation, auquel cas un comportement de moins en moins gourmand est adopté.

Il existe plusieurs variantes de cet opérateur, qui permettent de spécifier combien de fois l'expression doit être appliquée :

`*`	Au moins 0 fois.
`+`	Au moins 1 fois.
`?`	Entre 0 et 1 fois.
`{`n`}`	Exactement n fois.
`{`n`,}`	Au moins n fois.
`{,`n`}`	Au plus n fois.
`{`n`,`m`}`	Entre n et m fois.

Comme nous allons le voir dans quelques instants, grâce aux groupes, il est possible de répéter non pas uniquement un caractère ou une classe de caractères, mais tout une sous-partie d'une expression régulière.

1.5 Groupes

Les groupes permettent de regrouper une sous-partie d'une expression régulière, généralement dans le but d'y appliquer un opérateur, comme par exemple un opérateur de répétition. Pour former un groupe, il suffit d'entourer l'expression de `(?:` sur la gauche et de `)` sur la droite. Par exemple, dans le programme ci-dessous, l'expression régulière utilise un groupe afin de rechercher une sous-chaîne qui serait constituée de deux fois le symbole `e` suivi soit de `x` soit de `m`.

```
import re

regex = re.compile(r"(?:e[xm]){2}")

print(regex.search("un petit texte d'exemple."))
# Affiche: <re.Match object; span=(17, 21), match='exem'>
```

Dans l'exemple ci-dessus, un groupe est utilisé afin de spécifier que la sous-expression `e[xm]` doit être répétée 2 fois. Sans ce groupe, seule la classe `[xm]` serait sujette à l'opérateur `{2}`.

Groupes de capture

Il est aussi possible d'utiliser les groupes afin de *capturer* des sous-parties de l'expression et ainsi de pouvoir récupérer les caractères qui correspondent à cette partie. Les délimiteurs que l'on utilise pour de tels groupes sont `(` et `)` (à la place de `(?:` et `)`). Afin d'accéder aux caractères capturés par chaque groupe, on utilise la méthode `group` sur l'objet retourné par `search` ou `match`.

```
import re

regex = re.compile(r"t([a-z]+)t")

m1 = regex.search("un petit texte d'exemple.")

print(m1.group(1))
# Affiche: i

m2 = regex.search("un autre texte d'exemple.")

print(m2.group(1))
# Affiche: ex
```

Notez que les groupes sont numérotés à partir de 1. Le groupe 0 correspond à l'ensemble de la chaîne acceptée. Notez que dans l'exemple ci-dessus, il n'y a qu'un seul groupe.

Lorsqu'il y a plusieurs groupes, voire même des groupes imbriqués, l'ordre des groupes est donné par l'ordre d'apparition dans l'expression régulière de la parenthèse ouvrante qui délimite le groupe. Lorsqu'un groupe capture plusieurs sous-chaînes, comme cela peut être le cas lorsqu'un groupe est répété, seule la dernière capture est préservée.

1.6 Alternatives

L'opérateur noté à l'aide d'une barre verticale (`|`) permet de spécifier différentes alternatives dans une expression régulière. Ainsi, l'expression régulière `ABBA|Queen` accepte à la fois la chaîne `"ABBA"` et la chaîne `"Queen"`.

Il est bien entendu possible de combiner cet opérateurs avec les autres constructions vues plus haut, et ce afin de former des expressions régulières arbitrairement complexes, comme dans l'exemple plus bas.

```
import re

regex = re.compile(r"[a-z]*(?:pe|ex)[a-z]*")

for result in regex.findall("un petit texte d'exemple."):
    print(result)
```

Le programme ci-dessus, une fois exécuté, donne le résultat suivant :

```
petit
texte
exemple
```

En effet, le programme recherche les sous-chaînes formées uniquement de lettres minuscules qui contiennent la séquence `pe` ou la séquence `ex`.

1.7 Délimiteurs

Pour être complet, il nous reste à aborder un dernier détail. Certaines expressions permettent de spécifier non pas un caractère dans le texte, mais une position au sein d'un texte. Elles sont listées ci-dessous :

`\A`	Début de la chaîne.
`\Z`	Fin de la chaîne.
`\b`	Fin ou début d'un mot (suite de symboles alphanumériques ou `_`).

Ainsi, si l'on cherche à trouver tous les mots en d'un texte qui sont formés d'une séquence d'au moins une minuscule suivie d'un `e`, il est possible de procéder comme lors de notre tout premier exemple :

```
import re

regex = re.compile(r"[a-z]+e\b")

results = regex.findall("un petit texte d'exemple.")
for result in results:
    print(result)
```

Dans l'exemple ci-dessous, le `\b` permet de spécifier que le `e` est en fin de mot.