Les chaines de caractères 

Utiliser les expressions régulières

Pascal Burkhard
LDDR 2020-2021

Exercice 1

Consignes

- ouvrez le fichier `adresses_facile.csv` dans Excel
- observez les numéros de téléphone dans la colonne `Tel`
- modifiez les numéros pour qu'ils correspondent au format `0123456789`

Questions

- peut-on modifier tous les numéros pour obtenir un format `012 345 67 89` pour les numéros ?
- et un format `+41 12 345 67 89`

Exercice 2

Consignes

- importez les données du fichier `adresses_facile.csv` dans une liste de dictionnaires (en utilisant le module `csv` comme indiqué ci-dessous)

```
import csv

adresses_dict = csv.DictReader(open("chemin/vers/adresses_facile.csv"))
adresses_dict = list(adresses_dict)
```

- en utilisant vos connaissances actuelles, modifiez le format des numéros pour correspondre à `+41 12 345 67 89`
- répétez l'exercice avec le fichier `adresses_intermediaire.csv`
- observez le fichier `adresses_expert.csv` réfléchissez à une procédure pour modifier les numéros de téléphone comment pourrait-on procéder ?

Les expressions régulières à la rescousse 🦸

Source : https://xkcd.com/208/

Les expressions régulières...

... c'est quoi ?

Les **expressions régulières** (*regular expression* ou ***regex*** en anglais) ou parfois aussi *expressions rationnelles* permettent de décrire un ensemble de caractères au moyen d'un **motif** (*pattern* en anglais). Ce motif peut être plus ou moins complexe.

Les expressions régulières peuvent être considérées comme un langage de programmation en soit. Elles permettent d'effectuer des tâches spécialisées de **recherche** et de **remplacement**.

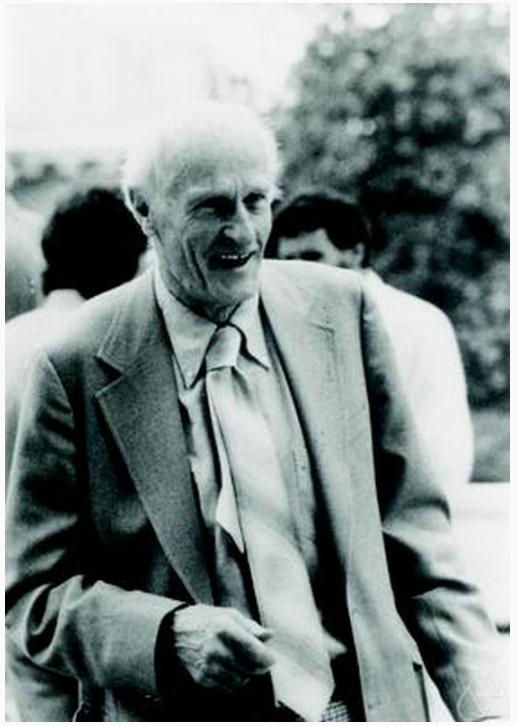
Les expressions régulières sont présentes dans un très grand nombre de langages de programmation, intégrées directement (comme dans *Perl* par exemple) ou à travers une extension (comme avec le module *re* dans *Python* 🐍).

Les expressions régulières sont également intégrées dans un grand nombre d'éditeurs de texte et d'environnements de développement (IDE).

Un peu d'histoire

La théorie qui donne naissance aux expressions régulières émerge des travaux de Warren McCulloch et Walter Pitts qui décrivent le fonctionnement des neurones dans le système neveux dans les années 1940. En ce basant sur ces travaux, le logicien **Stephen Cole Kleene** décrit les modèles imaginés sous forme d'ensembles réguliers et d'automates. Il est considéré comme le père des ***expressions régulières***.

Les expressions régulières font leur apparition dans le monde de l'informatique dans les années 1980 lorsqu'elles sont intégrées dans le programme *ed* – un éditeur de texte du système d'exploitation UNIX. Elles seront ensuite intégrées dans d'autre petits logiciels ou utilitaires (comme *vi*, *awk*, *grep*, *Emacs*) et dans divers langages de programmation (comme *Perl*, *Java* ou encore *Python* 🐍).



Stephen Cole Kleene
Source: Wikimedia

L'implémentation

Divers langages de programmation implémentent les expressions régulières. Ces implémentations varient *malheureusement* d'un langage à un autre. Même s'il existe des standards (comme le standard *POSIX* du monde *UNIX*, ou le standard *PCRE* implémenté dans le langage *Perl*), de petites divergences subsistent.

Les expressions régulières dans *Python* 🐍 suivent le standard *POSIX* avec quelques petites différences. Dans le cadre de cette introduction, ces différences n'ont aucun impact. Il est cependant prudent de tester son expression régulière dans un moteur (voir *Ressources utiles* dans l'aide-mémoire) qui correspond au standard utilisé !

Comment ça marche ?

Une expression régulière est un **motif** définissant un ensemble de chaines de caractères. Le motif d'une expression régulière est composé de différents **symboles** :

- des **caractères littéraux**
  des symboles qui expriment *littéralement* le caractère (le symbole correspond à lui-même), par exemple le symbole `a` correspond à la lettre *a*

- des **métacaractères**
  des symboles qui ne correspondent pas directement à eux-mêmes – ces symboles ont une signification particulière décrite par la suite
  dans les expressions régulières de *Python* 🐍 ils sont au nombre de 14 : `. ^ $ * + ? { } [ ] \ | ( )`

Les caractères littéraux

La plupart des symboles d'une expression régulière sont des *caractères littéraux* : ils se décrivent eux-mêmes.

Le motif `in` sera par exemple détecté de la manière suivante dans la chaine de caractères ci-dessous :

```
L'informatique n'est qu'un outil, comme un pinceau ou un crayon.
```

Les métacaractères

Les métacaractères utilisés dans les expressions régulières sont utilisés pour les applications suivantes :

- des ancrages
  définir un point précis dans une chaine de caractères, tel que le début de la chaine ou la limite d'un mot par exemple

- des classes de caractères et des groupes
  définir un groupe de caractères à faire correspondre

- des quantifieurs
  définir le nombre de fois qu'une partie du motif de l'expression régulière est répété

Les métacaractères permettent également d'autres fonctions avancées, qui dépassent le cadre de ce cours.

Les métacaractères

Les ancrages

Les ancrages permettent de fixer un motif à un point précis de la chaine de caractères. Ces métacaractères permettent de se fixer sur des éléments *invisibles* d'une chaine de caractères. Les ancrages principaux sont :

- `^` : correspond au début d'une ligne (avec des chaines de caractères multilignes)
- `$` : correspond à la fin d'une ligne (avec des chaines de caractères multilignes)
- `\A` : correspond au début d'une chaine de caractères
- `\Z` : correspond à la fin d'une chaine de caractères
- `\b` : correspond à la limite d'un mot (début ou fin)
- `\B` : correspond à l'opposé de `\b` (toute position sauf le début ou la fin d'un mot)

Les métacaractères

Les ancrages – quelques exemples

Observez comment l'ancre `\A` restreint le motif `\Aun programme` :

```
un programme sans bug est un programme qui n'a pas été suffisament testé
```

Observez la différence entre les motifs `\Bqu` et `\bqu` :

```
Comme la Hongrie, le monde informatique a une langue qui lui est propre. Mais il y a une différence. Si
vous restez assez longtemps avec des Hongrois, vous finirez bien par comprendre de quoi ils parlent.
```

Les métacaractères

Les classes de caractères

Les classes de caractères permettent de définir une collection de caractères. Les crochets permettent de définir des classes de caractères. Par exemple le motif `[abc]` peut correspondre à la lettre *a* comme *b* ou *c*. Il est également possible d'inverser une classe de caractère en utilisant le symbole `^` au début de l'ensemble. Par exemple `[^abc]` va correspondre à tous les symboles sauf *a*, *b* et *c*. Certaines classes de caractères sont définies avec des *raccoucis*. Quelques classes de caractères utilisées fréquemment :

- `.` : correspond à n'importe quel symbole
- `[0-9]` ou `\d` : correspondent à tous les chiffres
- `[^0-9]` ou `\D` : correspondent à tout sauf les chiffres
- `[a-z]` : correspond aux lettres minuscules *a* à *z*
- `\s` : correspond aux espaces
- `\S` : correspond à tout sauf aux espaces
- `\w` : correspond aux lettres, chiffres et au symbole `_` (équivalent à `[A-Za-z0-9_]`)
- `\W` : correspond à tout sauf aux lettres, chiffres et au symbole `_` (équivalent à `[^A-Za-z0-9_]`)

Les métacaractères

Les classes de caractères

Le standard *POSIX* définit par ailleurs des classes avec un format `[:mot:]`. Quelques exemples (non exhaustif) :

- `[:upper:]` : Lettres majuscules
- `[:lower:]` : Lettres minuscules
- `[:alpha:]` : Lettres majuscules et minuscules
- `[:alnum:]` : Lettres et chiffres
- `[:digit:]` : Chiffres (correspond à `\d` ou `[0-9]`)
- `[:punct:]` : Ponctuation

Les classes `[:upper:]`, `[:lower:]`, `[:alpha:]` et `[:alnum:]` ont l'énorme avantage (en français surtout) d'inclure les caractères accentués (ce que `[A-Z]` ou `[a-z]` ne fait pas) !

Les classes de caractères – quelques exemples

Observez les occurrences des motifs `\s`, `\d` et `[pl]` dans la chaine de caractères ci-dessous :

```
2 n'est pas égal à 3, même pour de grandes valeurs de 2 ou de petites valeurs de 3.
```

Et le motif `[A-Z]` dans la phrase ci-dessous :

```
Si vous mettez 30 novices devant un ordinateur équipé de Windows, vous aurez 30 manières différentes de planter Windows.
```

Notez la différence entre le motif `[A-Za-z]` et le motif `[:alpha:]`.

```
Tout ordinateur est obsolète au plus tard à son déballage.
```

```
Tout ordinateur est obsolète au plus tard à son déballage.
```

Les métacaractères

Les groupes

Les métacaractères `(` et `)` permettent de définir un groupe dans un motif. Il existe deux types de groupes, les groupes *capturant* et *non-capturant*. Cette distinction devient particulièrement pertinente lorsqu'on procède à des remplacements en utilisant des expressions régulières.

- `(bug)` : groupe capturant avec le motif `bug`
- `(?:bug)` : groupe non capturant avec le motif `bug`

Les groupes sont particulièrement pertinents lorsqu'on utilise le métacaractère `|` (opérateur logique *OU*). Quelques exemples :

- `(Tom|Fred)` : groupe capturant avec le motif `Tom` ou le motif `Fred`
- `ser(ons|ez|ont)` : motif correspondant aux formes pluriels du futur simple du verbe être

Les quantifieurs

Les quantifieurs permettent d'exprimer un nombre de fois qu'une partie du motif (un caractère littéral, un groupe ou une classe de caractères) peut être répétée :

- `?` : 0 ou 1 fois
- `*` : 0 fois ou plus
- `+` : 1 fois ou plus
- `{2}` : 2 fois (exactement)
- `{6}` : 6 fois (exactement)
- `{3,5}` : 3, 4 ou 5 fois
- `{4,}` : 4 fois ou plus

Les quantifieurs sont *gourmands* par défaut. Cela signifie qu'ils engloberont toujours le maximum de symboles possibles. En rajoutant un `?` aux quantifieurs ci-dessus (par exemple `??`, `*?`, `+?`, `{2,}?` ou `{3,6}?`), ceux-ci deviendront *minimalistes* et engloberont le moins de symboles possibles.

Les quanfitieurs sont généralement combinés avec une classe de caractères ou un groupe, mais ils peuvent aussi s'appliquer à un caractère littéral.

Les quantifieurs – quelques exemples

Observez le résultat du motif `(très\s)+` ci-dessous :

```
Les composantes d'un microprocesseur sont très très très très petites!
```

et comparez avec le résultat du motif `(très\s)+?` ci-après :

```
Les composants d'un microprocesseur sont très très très très petits!
```

notez la différence entre un quantifieur *gourmand* et *minimaliste* !

Les quantifieurs – quelques exemples

Les quantifieurs peuvent également s'avérer très utiles en combinaison avec des classes de caractère. Le motif `\A[^,]*` identifie la première partie de la phrase ci-dessous jusqu'à rencontrer une virgule :

```
Grâce à l'ordinateur, on peut faire plus rapidement des choses qu'on n'aurait pas eu besoin de faire sans ordinateur.
```

On peut également appliquer les quantifieurs à des raccourcis (comme `\S` dans l'exemple ci-après). Le motif `\b\S{5,}\b` correspond aux mots de 5 caractères et plus :

```
Rappelez-vous que ce n'est pas un hasard si l'informatique et la théorie du chaos se sont développées simultanément.
```

Les métacaractères

Échapper un métacaractère

Le métacaractère `\` (*barre oblique inverse* ou *backslash* en anglais) permet d'*échapper* les métacaractères.

Par exemple, lorsqu'on souhaite utiliser le symbole `.` dans une expression régulière, il faut l'*échapper*. On obtiendra le motif suivant `\.` :

```
Un programme informatique fait ce que vous lui avez dit de faire, pas ce que vous voulez qu'il fasse.
```

On notera également que dans une classe de caractères (entre crochets `[]`), seul le crochet fermant `]` et le circonflexe `^` (lorsqu'il est au début) doivent être échappés.

Les expressions régulières dans *Python* 🐍

Lorsqu'on utilise des expressions régulières dans *Python* 🐍, on se trouve rapidement confronté au conflit du métacaractère `\` utilisé dans les expressions régulières pour échapper les métacaractères, mais également dans les chaines de caractères de *Python* 🐍. Par exemple lorsqu'on délimite une chaine de caractères avec des apostrophes, la *barre oblique inverse* permet d'échapper l'apostrophe : `'aujourd\'hui'`.

On se retrouverait donc à devoir échapper doublement les `\` pour que ceux-ci fonctionnent correctement. Le motif `\b\S{5,}\b` présenté précédemment, deviendrait `\\b\\S{5,}\\b` : cela devient vite un cauchemard !

Fort heureusement *Python* 🐍 nous offre une solution simple à ce problème : les *chaines de caractères brutes* (*raw strings* en anglais). Cette structure permet d'éviter l'échappement par Python. Une *chaine de caractères brute* est créée en préfixant la chaine de caractères d'une lettre *r* (ou *R*) :

```
r'chaine brute\ dans laquelle la barre oblique est maintenue telle quelle'
r"deuxième chaine brute\ avec la même barre oblique"
```

Seule limitation de ces chaines brutes : les apostrophes ou guillemets (selon ce qui est utilisé) peuvent être échappés. On ne pourra donc pas avoir la chaine brute suivante : `r'\'` !

Les expressions régulières dans *Python* 🐍

Pour travailler avec des expressions régulières dans Python il faut importer le module *re* :

```
import re
```

Le module *re* propose plusieurs méthodes pour travailler avec des expressions régulières dans *Python* 🐍. Nous allons nous concentrer sur trois méthodes :

- re.**findall**(*pattern*, *string*, *flags=0*)
- re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)
- re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

La méthode `findall()` permet de trouver toutes les occurrences du motif (*pattern*) dans la chaine (*string*). Cette méthode retourne une liste des occurrences trouvées.

La méthode `split()` permet de diviser la chaine (*string*) en coupant là où elle trouve le motif (*pattern*). Le paramètre *maxsplit* permet de spécifier un nombre de morceaux maximal. Cette méthode retourne également une liste.

La méthode `sub()` permet de remplacer une dans une chaine (*string*) les motifs (*pattern*) par la chaine spécifiée par le paramètre *repl*.

Les expressions régulières dans *Python* 🐍

re.**findall**(*pattern, string, flags=0*)

Quelques exemples :

```
re.findall(r"\d+", "Un ordinateur fait autant d'erreur en 2 secondes que 20 humains en 20 ans.")
```

```
## ['2', '20', '20']
```

```
re.findall(r"\b\S{5,}\b", "Rappelez-vous que ce n'est pas un hasard si l'informatique et la théorie du
chaos se sont développées simultanément.")
```

```
## ['Rappelez-vous', "n'est", 'hasard', "l'informatique", 'théorie', 'chaos', 'développées', 'simultanément']
```

Les expressions régulières dans *Python* 🐍

re.**split**(*pattern, string, maxsplit=0, flags=0*)

Quelques exemples :

```
re.split(r"/", "un/deux/trois/quatre/cinq/six/sept")
```

```
## ['un', 'deux', 'trois', 'quatre', 'cinq', 'six', 'sept']
```

```
re.split(r"\W", "C'est quand votre ordinateur est éteint qu'il fonctionne le mieux.")
```

```
## ['C', 'est', 'quand', 'votre', 'ordinateur', 'est', 'éteint', 'qu', 'il', 'fonctionne', 'le', 'mieux', '']
```

Les expressions régulières dans *Python* 🐍

re.**sub**(*pattern, repl, string, count=0, flags=0*)

```
re.sub(r"à[^,]*", "aux expressions régulières", "Quand on se met à l'informatique, il vaut mieux avoir
BEAUCOUP d'amis.")
```

```
## 'Quand on se met aux expressions régulières, il vaut mieux avoir BEAUCOUP d'amis.'
```

Dans le cadre des remplacements, les groupes *capturant* prennent tout leur sens. On peut en effet utiliser des *références* à ces groupes dans l'argument *repl*. Ces références prennent la forme `\1` pour le premier groupe, `\2` pour le deuxième groupe et ainsi de suite. Notez l'utilisation d'une chaine brute pour l'argument de remplacement :

```
re.sub(r"0(\d{2})\s(\d{3})\s(\d{2})\s(\d{2})", r"+41 \1 \2 \3 \4", "012 345 67 89")
```

```
## '+41 12 345 67 89'
```

Quelques ressources utiles

Références

- Guides et références (en anglais) : https://www.regular-expressions.info
- Guides et références (axé sur le langage PHP, en français) : http://www.expreg.com
- Exemples d'expressions fréquentes : https://projects.lukehaas.me/regexhub/
- Documentation *regex* de *Python* 🐍 (en français) : https://docs.python.org/fr/3/howto/regex.html

Moteurs d'expressions régulières

- https://regexr.com
- https://regex101.com
- https://debuggex.com

Exercice 3

Consignes

- revisitez le fichier `adresses_facile.csv`
- en utilisant le modèle `exercice3_exemple.py`, modifiez l'expression régulière (l'expression actuelle `(.*)` ne change rien aux numéros de téléphone) pour obtenir un format `+41 12 345 67 89`.
- lorsque vous y êtes parvenu avec le fichier `adresses_facile.csv` essayez `adresses_intermediaire.csv` puis `adresses_expert.csv`

Conseil

Faites usage d'un des moteurs d'expressions régulières évoqués dans l'aide-mémoire !

Références

Courtois, C. (2021). *Les Lois de Murphy*. URL: https://www.coindeweb.net/murphy/murphy.html (visited on May. 02, 2021).

Downey, A. B. (2012). *Think Python*. O'Reilly Media, Inc. 299 pp. ISBN: 978-1-4493-3202-0.

Fitzgerald, M. (2012). *Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step*. "O'Reilly Media, Inc.". 153 pp. ISBN: 978-1-4493-3890-9.

Friedl, J. (2006). *Mastering Regular Expressions*. "O'Reilly Media, Inc.". 542 pp. ISBN: 978-0-596-52812-6.

Mueller, J. P. (2018). *Beginning Programming with Python For Dummies*. John Wiley & Sons. 412 pp. ISBN: 978-1-119-45790-9.

Stubblebine, T. (2007). *Regular Expression Pocket Reference: Regular Expressions for Perl, Ruby, PHP, Python, C, Java and .NET*. "O'Reilly Media, Inc.". 129 pp. ISBN: 978-0-596-51427-3.