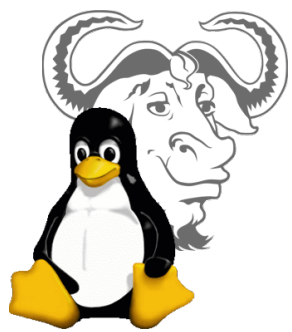
Administration d'un système GNU / Linux

02 — Entrées, sorties et *pipes*

Anthony Labarre

上海师范大学

1 novembre 2023



Plan d'aujourd'hui

1. Entrées et sorties
2. Les *pipes*
3. `grep`
4. `sed`
5. Expressions régulières

Entrées et sorties

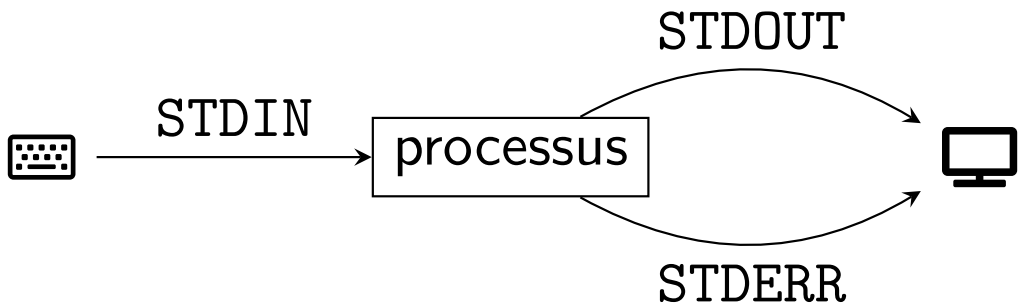
Processus

- Rappel: un **processus** est une instance d'un programme en cours d'exécution;
- En général, ils reçoivent et produisent des données;
- On va voir comment faire communiquer les processus;
- Cela revient à manipuler leurs **entrées** et **sorties**;

Entrées et sorties

- Un processus manipule deux types de flux:
  1. des **entrées**: les données qu'il reçoit;
  2. des **sorties**: les données qu'il produit;
- Nous allons voir aujourd'hui comment exploiter ces entrées et sorties;

Entrées et sorties standard



- En plus des entrées et sorties d'un processus, il existe trois types d'entrées / sorties "généraux", qui sont numérotés:
  0. l'**entrée standard** (ou `STDIN`) est le clavier (`/dev/stdin`);
  1. la **sortie standard** (ou `STDOUT`) est l'écran (`/dev/stdout`);
  2. l'**erreur standard** (ou `STDERR`) est aussi l'écran (`/dev/stderr`);

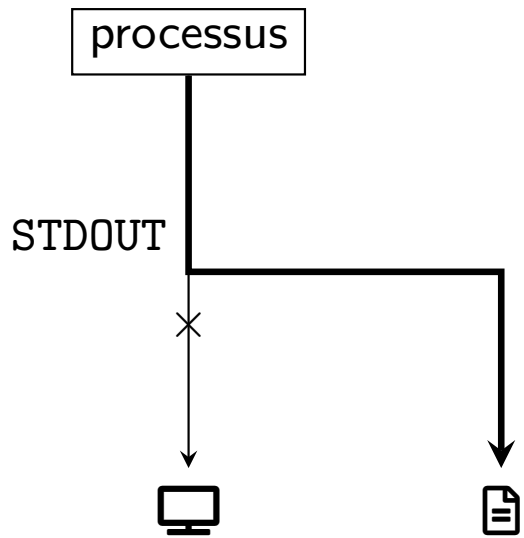
Redirection des sorties: >

- La plupart des commandes affichent leur résultat sur la sortie standard;
- Comment faire pour enregistrer ce résultat dans un fichier?
- Il suffit de **rediriger** sa sortie avec l'opérateur >;

Exemple

```
$ ls -l > contenu_répertoire.txt
```

- `processus > sortie` redirige ce que `processus` devrait écrire sur `STDOUT` vers le fichier `sortie`;
- Si `sortie` existe, son contenu est remplacé; sinon, `sortie` est créé par la commande;



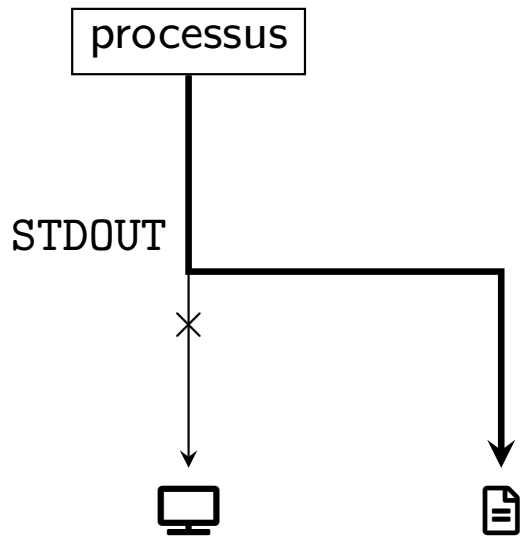
Redirection des sorties: >>

- Comment faire si l'on veut **ajouter** un résultat à une sortie?
- On utilise >> au lieu de > pour la redirection;

Exemple

```
$ ls -l > contenu_répertoire.txt
$ ls -l .. >> contenu_répertoire.txt
```

- La sortie de la seconde commande est rajoutée à la fin de `contenu_répertoire.txt`;



Redirection de l'erreur standard

On peut rediriger la sortie standard et l'erreur standard vers **deux fichiers distincts**: il suffit de préciser le numéro de la sortie à rediriger.

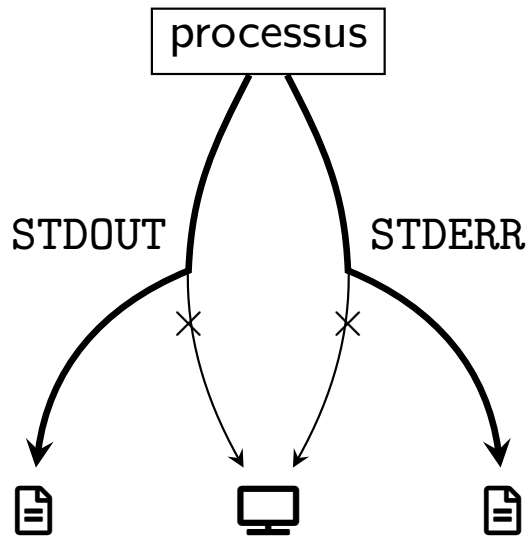
Exemple

(du = **d**isk **u**sage: affiche l'espace consommé par un répertoire.)

```
$ du -sh /
# erreurs
# espace utilisé sur le disque
```

```
$ du -sh / > consommation.txt
# erreurs
# espace -> consommation.txt
# espace utilisé sur le disque
```

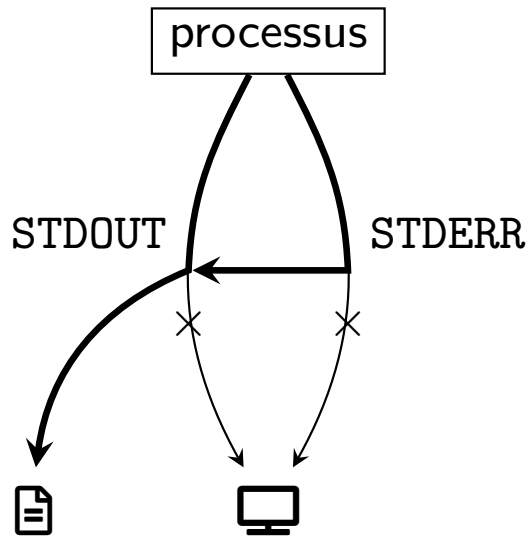
```
$ du -sh / > consommation.txt 2> log.txt
# aucun message
# espace -> consommation.txt
# erreurs -> log.txt
```



Grouper les redirections

- Plus compliqué: on peut aussi rediriger l'erreur et la sortie standard vers le même fichier;
- On utilise pour ça la syntaxe

  `processus > sortie 2>&1;`



Exemple

```
$ du -sh / > consommation.txt 2>&1
# erreurs ET résultat dans
↪  consommation.txt
```

- L'ordre est important!

  ✓ `processus > sortie 2>&1`

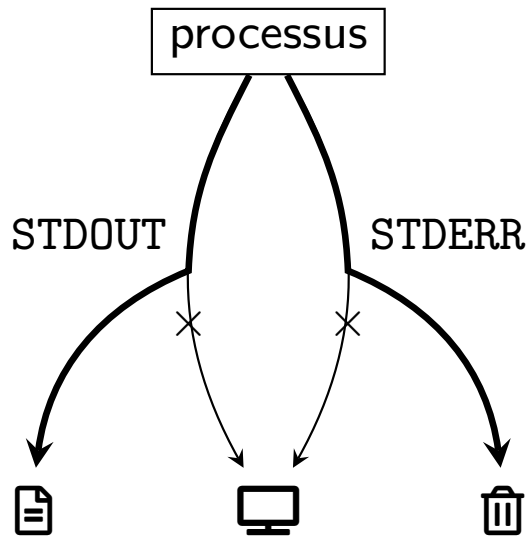
  × `processus 2>&1 > sortie`

/dev/null: la poubelle

- Et si on ne veut ni voir les erreurs, ni les placer dans un fichier?
- On peut rediriger l'erreur standard vers `/dev/null`.

Exemple

```
$ du -sh / > consommation.txt 2>
↪  /dev/null
# espace dans consommation.txt;
# erreurs à la poubelle
```



Redirection des entrées: <

- Si l'on veut utiliser une autre entrée que le clavier, on utilise l'opérateur <;
- On est obligé de le faire avec les programmes qui ne travaillent qu'avec l'entrée standard;

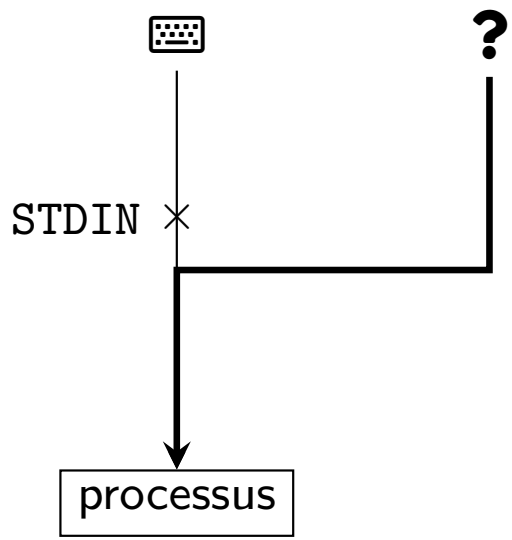
Exemple

Pour installer un nouveau système identique à l'ancien, on peut enregistrer les noms des paquets installés:

```
$ dpkg --get-selections > paquets
```

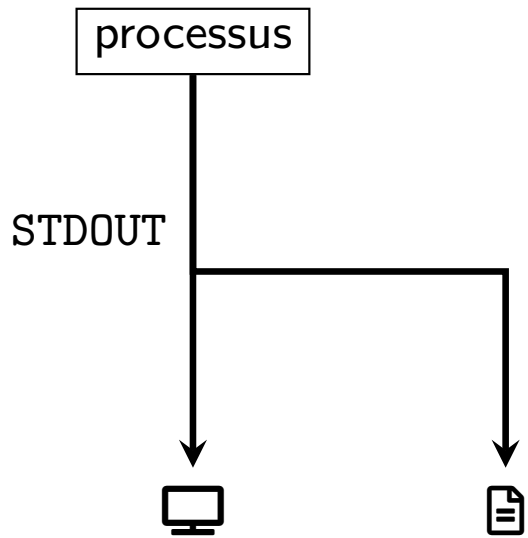
et puis les réinstaller sur le nouveau système:

```
$ sudo dpkg --set-selections < paquets
$ sudo apt-get dselect-upgrade
```



tee

- Quand on redirige la sortie standard, on ne voit plus ce que produit le processus;
- Le programme `tee` permet de régler ce problème: on voit la sortie, et on l'enregistre en même temps dans un fichier;



Exemple

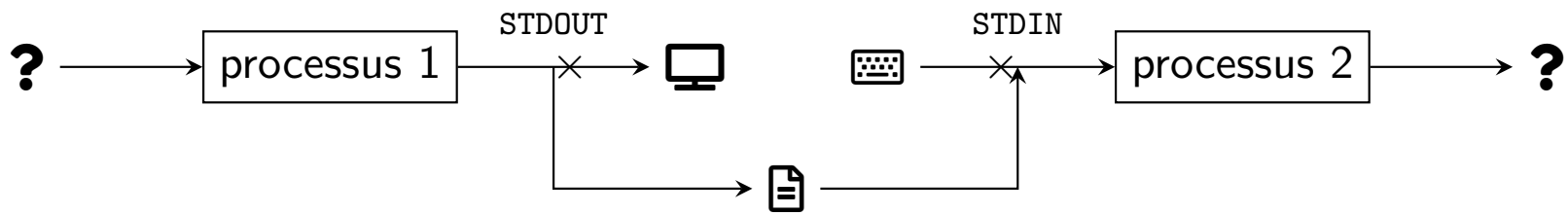
```
$ ls -l | tee sortie.txt
# équivalent de >
$ ls -l | tee -a sortie.txt
# équivalent de >>
```

- Le symbole | correspond à un *pipe* (= tuyau), qu'on va maintenant expliquer;

Les *pipes*

Les *pipes*: motivations

- Les **pipes** sont un mécanisme permettant d'utiliser la sortie d'un processus comme entrée d'un autre processus;
- On peut y arriver avec des redirections:



- . . . mais cela nous oblige à créer un fichier intermédiaire, avec plusieurs inconvénients:
  - × taille: le fichier peut être gros, et ne sert qu'une fois;
  - × lenteur: on doit d'abord tout écrire dans le fichier, et ensuite le lire en entier;

Les *pipes*: utilisation

- Au lieu de cela, on peut utiliser un *pipe* pour que la sortie d'un processus devienne l'entrée du processus suivant:

- On obtient ce comportement avec la syntaxe `processus_1 | processus_2`

Exemple

Extrayons des champs de `/etc/passwd` et trions-les:

```
$ cut -f1 -d: /etc/passwd | sort    # usernames
$ cut -f3 -d: /etc/passwd | sort -h # UIDs
```

Contraintes des pipes

- La seule contrainte des *pipes* est le fonctionnement des programmes que l'on combine;
- En effet, quand on écrit `processus_1 | processus_2`:
  1. on redirige `STDOUT` vers l'entrée du *pipe* pour `processus_1`;
  2. on remplace `STDIN` par la sortie du *pipe* pour `processus_2`;
- Si `processus_1` n'écrit pas sur `STDOUT` (ou si `processus_2` ne lit pas sur `STDIN`), il y aura des redirections à faire;

Avantages des pipes

Si vous n'avez pas besoin des données intermédiaires, utilisez les *pipes*!

- ✓ pas de données grosses et inutiles à stocker;
- ✓ plus rapide: on peut traiter chaque ligne directement au lieu d'attendre le résultat complet;

Pipes multiples

- On peut combiner autant de *pipes* qu'on veut!
- Dans ce cas, la sortie du processus avant le *pipe* `i` devient l'entrée du processus suivant ce même *pipe*;

Exemple

La commande suivante permet d'obtenir la liste triée des noms des processus qui travaillent sur un fichier ouvert:

```
$ lsof | cut -f1 -d' ' | sort | uniq
```

grep

grep

- grep est l'un des programmes les plus utiles de GNU;
- Il permet de récupérer toutes les lignes d'un fichier contenant ou évitant un morceau de texte donné;
- Il permet aussi d'utiliser des **expressions régulières** (plus tard);

grep: premier exemple

Utilisation la plus simple: chercher toutes les occurrences exactes d'un mot (en respectant la casse).

Exemple

```
$ grep Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
# affiche toutes les lignes contenant Quasimodo
```

```
$ grep -c Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
241 # affiche le nombre de lignes contenant Quasimodo
```

```
$ grep -o Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
# affiche chaque occurrence sur une ligne séparée
Quasimodo
Quasimodo
...
```

```
$ grep -o Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt | wc -l
244 # le nombre d'occurrences de Quasimodo dans le livre
```

grep: deuxième exemple

On peut demander à ignorer la casse:

Exemple

```
$ grep -c Quatre Hugo\ -\ Notre-Dame\ de\ Paris.txt
9 # le nombre d'occurrences de "Quatre"
```

```
$ grep -c -i Quatre Hugo\ -\ Notre-Dame\ de\ Paris.txt
120 # le nombre d'occurrences de "Quatre" et "quatre" et ...
```

Attention: grep ne considère pas “Quatre” comme un mot isolé!

Parmi les occurrences, on retrouvera donc aussi “Quatrelivres”, “Quatre-Couronnes”, “Quatre-Nations”, . . .

Variantes

Il existe quelques variantes utiles de `grep`:

- `pgrep`: recherche un motif parmi les noms des processus;
- `agrep`: recherche de motifs approximatifs (avec des “erreurs”);
- `ngrep`: recherche de motifs dans le trafic réseau;
- `zipgrep`: `grep` pour les fichiers zip . . . sans devoir les décompresser!
- . . .

sed

sed

- grep est très utile pour trouver des motifs dans un texte . . .
- . . . mais comment faire si l'on veut *remplacer* des motifs par d'autres motifs?
- On doit utiliser un autre programme: par exemple, sed;

sed

- sed est un éditeur de flux (**s**tream **ed**itor);
- Il permet de remplacer des motifs par d'autres dans n'importe quel flux — donc des fichiers, mais aussi des sorties de *pipes*;
- Par défaut, sed écrit son résultat sur STDOUT;
- L'option -i permet de modifier les fichiers directement;

Si vous débutez avec sed, vérifiez son résultat avant d'utiliser le mode -i!

Remplacement de texte avec `sed`

- La commande `sed s/avant/après/g fichier` remplace toutes les occurrences de “avant” par “après” dans `fichier`;
- “avant” peut être une chaîne ou une expression régulière (voir plus loin);
- Attention: la casse est respectée;
- On peut faire beaucoup d’autres choses avec `sed` (voir séances d’exercices);

Expressions régulières

Expressions régulières

- Une **expression régulière** est une chaîne qui encode un *modèle* de texte.
- Exemples:
  - les mots en minuscules ou en majuscules;
  - les mots qui commencent par une majuscule;
  - les noms;
  - les numéros de téléphone;
  - les adresses e-mail;

  ...
- Utilité: au lieu de chercher un mot précis dans un texte, on peut chercher tous les morceaux de texte respectant une certaine structure;

Applications

Recherchons toutes les dates “avant Jésus-Christ” dans le texte suivant (par exemple: 1300 av. J.-C. = 1300 B.C. = 前1300年):

Ressources

Les expressions régulières peuvent parfois être difficiles à interpréter.

Le site `https://regexr.com/` vous permet de tester en direct vos expressions régulières, avec le texte de votre choix.

N'hésitez pas à utiliser cette ressource pour vous entraîner et vérifier vos expressions.

Premier exemple: les dates

- Construisons une expression permettant de décrire les dates au format international (`YYYY-MM-DD`, par exemple: 2023-11-06);
  - l'année `YYYY` est un nombre entre 0000 et 9999, donc quatre entiers dans `[0, 9]`;
  - le mois `MM` est un nombre entre 01 et 12, donc un entier dans `[0, 1]` suivi d'un entier dans `[0, 9]`;
  - le jour `DD` est un nombre entre 01 et 31, donc un entier dans `[0, 3]` suivi d'un entier dans `[0, 9]`;
- Le résultat est l'expression régulière suivante:

`[0-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]`

(année) (mois) (jour)

- Si on préfère les dates chinoises:

`[0-9][0-9][0-9][0-9]年[0-1][0-9]月[0-3][0-9]日`

Remarques sur les dates

- Notre expression ne distingue pas les dates justes des dates fausses:
  - elle accepte des mois faux: `00`, ou des nombres > 12;
  - elle accepte des jours faux: `00`, ou des nombres > 31;
- On n'essaiera pas de la corriger, car une expression correcte deviendrait très compliquée:
  - certains mois ont 30 ou 31 jours;
  - le mois de février a parfois 28 jours, parfois 29;

- On préférera récupérer toutes les dates en supposant qu'elles sont correctes, puis éventuellement les filtrer autrement;

Deuxième exemple: les prénoms

- Un prénom commence par une majuscule, suivie de n'importe quel nombre de lettres minuscules;
  - la première lettre est donc n'importe quelle majuscule dans `[A, Z]`;
  - les lettres suivantes sont n'importe quelle minuscule dans `[a, z]` (oublions les caractères spéciaux pour simplifier);
  - l'opérateur `*` dans l'expression `x*` précise que `x` apparaît n'importe quel nombre de fois (éventuellement 0);
- Le résultat est l'expression régulière suivante:

```
[A-Z][a-z]*
```

...qui trouvera Marie, Yves, Louis, ... mais pas Jean-Paul (pourquoi?)

Les bases

Les **éléments** d'une expression régulière peuvent être:

- des caractères "normaux": a, b, c, 1, 2, ...;
- des caractères "spéciaux": `\t`, `\n`, ...;
- des ensembles de caractères: `[az]` = "a ou z";
- des intervalles: `[a-z]`, `[A-Z]`, `[0-9]`, ...;

Symboles fréquents		Négations	
`[ab]`	a ou b	`[^ab]`	tout sauf a et b
`[0-9]` ou `\d`	les chiffres	`\D`	tout sauf un chiffre
`[a-z]`	les lettres minuscules		
`[A-Z]`	les lettres majuscules		
`\w`	les lettres et les chiffres	`\W`	tout sauf une lettre ou un chiffre
`\s`	les espaces	`\S`	tout sauf un espace
`.`	tout sauf `\n`		

Les répétitions

On indique les répétitions de caractères avec les opérateurs suivants (format: symbole suivi de l'opérateur, sans espace):

- `?`: 0 ou 1 fois;
- `+`: au moins une fois;
- `*`: autant de fois qu'on veut;
- `{n}`: exactement `n` fois;
- `{n,}`: au moins `n` fois;
- `{,m}`: au plus `m` fois;
- `{n,m}`: entre `n` et `m` fois;

Remarques

- On peut combiner les intervalles: `[a-zA-Z]` = n'importe quel caractère non accentué;
- Si vous avez besoin d'un caractère spécial, "échappez"-le; par exemple:
  - `a*` = "n'importe quel nombre de a";
  - `a\*` = "a suivi d'une étoile";

Expression plus simple pour les dates

On peut maintenant simplifier l'expression qu'on a obtenue pour les dates:

```
[0-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]
                    ↓
\d\d\d\d-[0-1]\d-[0-3]\d
                    ↓
\d{4}-[0-1]\d-[0-3]\d
```

Exemple plus avancé: les nombres naturels

- L'expression régulière pour les dates autorise la présence de 0 comme "remplissage" (par exemple: 0635 pour l'année 635);
- Comment exprimer les nombres naturels sans remplissage?
- `[1-9][0-9]*` exprime les naturels qui ne commencent pas par 0 ... et on ne peut donc pas représenter 0!
- On utilise l'opérateur logique `or` pour exprimer qu'un naturel est "soit 0, soit un nombre ne commençant pas par 0";
- Cet opérateur s'écrit `|` (attention: ceci n'est pas un *pipe*);
- Cela donne:

```
0|[1-9][0-9]*
```

Utilisation des expressions régulières avec grep

- Pour expliquer à grep que le motif est une expression régulière et pas une chaîne “normale”, on utilise l’option -E ou -P;
- grep nous affiche chaque ligne contenant ce qu’on cherche en entier;
- Pour n’afficher que ce qui nous intéresse, on utilise l’option -o: elle affiche:
  - uniquement la partie correspondant à l’expression donnée;
  - chaque résultat sur une ligne différente;

Place des expressions régulières

- On peut aussi préciser **où** les résultats doivent apparaître, avec les opérateurs suivants:
  - `^expression`: l'expression doit apparaître au début de la ligne;
  - `expression$`: l'expression doit apparaître à la fin de la ligne;

Examinons quelques exemples . . .

Différents formats

Il existe différentes conventions (ou formats) pour les expressions régulières!

- Testez donc vos expressions, et consultez les manuels des programmes!
- Par exemple, `[0-9]+` et `\d+` sont équivalentes . . . mais:
  - ✓ `grep -E "[0-9]+" fichier` fonctionne;
  - × `grep -E "\d+" fichier` ne fonctionne pas;
  - ✓ `grep -P "\d+" fichier` fonctionne;

Groupes

- `grep` sur une expression régulière nous donne toutes les lignes qui contiennent un motif correspondant à l'expression;
- Bien souvent, on veut récupérer le motif, pas la ligne entière;
- Pour cela, on utilise les **groupes (de capture)**, et un autre programme nommé `pcregrep`;

Groupes: définition

- Un **groupe** dans une expression régulière est une partie de l'expression que l'on veut isoler;
- On le définit simplement avec des parenthèses;

Exemple

Pour isoler les années, mois et jours d'une date, on définit les trois groupes suivants:

(année) (mois) (jour)

```
(\d{4})-([0-1]\d)-([0-3]\d)
```

Groupes: extraction

Une fois les groupes définis, `pcregrep` les extrait avec l'option `-oN`, où `N` est le numéro du groupe.

Exemple

Extrayons divers morceaux des dates du fichier `test.txt`:

```
$ pcregrep -o1 "(\d{4})-([0-1]\d)-([0-3]\d)" test.txt
# toutes les années
```

```
$ pcregrep -o2 "(\d{4})-([0-1]\d)-([0-3]\d)" test.txt
# tous les mois
```

```
$ pcregrep -o3 "(\d{4})-([0-1]\d)-([0-3]\d)" test.txt
# tous les jours
```

```
$ pcregrep -o3 -o2 -o1 "(\d{4})-([0-1]\d)-([0-3]\d)" test.txt
# tous les dates au format DDMMYYYY
```

Quelques mots sur les expressions régulières chinoises

- `grep` accepte les caractères chinois, y compris dans les expressions régulières;
- `pcregrep` aussi mais il faut rajouter l'option `-u`;
- Les intervalles ne fonctionnent pas: au lieu de `[一-十]`, on est obligé d'écrire `[一二三四五六七八九十]`
- `"\p{Han}"` = n'importe quel caractère chinois (avec `-u` pour `pcregrep` et `-P` pour `grep`);
- Attention: rappelez-vous que “ 。 ” ≠ “.” (pareil pour le reste de la ponctuation);