STAT243 Lecture 3 Bash Shell (Reduced)

1 Bash Shell-Overview

> ⚠ **Remark: 关于 echo 命令**
>
> `echo` 是一个 Bash 命令, 用于在终端显示文本或变量的值

`echo` 的主要用途

- **显示简单的文本:** 可以用 `echo` 来打印任何字符串。

  Shell
  ```
  echo "Hello, world!"
  ```

  输出:
  ```
  Hello, world!
  ```

- **显示变量的值:** 在脚本中, `echo` 通常用来检查变量是否被正确设置

  Shell
  ```
  name="Alice"
  echo "My name is $name."
  ```

  输出:
  ```
  My name is Alice.
  ```

- **输出到文件:** 你可以使用重定向操作符 (`>` 或 `>>`) 将 `echo` 的输出写入文件，而不是显示在终端。
  - `>` 会覆盖文件原有内容。
  - `>>` 会将新内容追加到文件末尾。

  Shell
  ```
  echo "This is the first line." > file.txt
  echo "This is the second line." >> file.txt
  ```

  运行后， `file.txt` 文件内容为：
  ```
  This is the first line.
  This is the second line.
  ```

`echo` 的常见选项

- **`-n`:** 不在输出的末尾添加换行符, 可以在同一行上连续输出内容

  Shell
  ```
  echo -n "Starting..."
  echo " Done."
  ```

  输出:
  ```
  Starting... Done.
  ```

- **`-e`:** 启用反斜杠转义, 可以使用一些特殊的转义序列, 如:
  - `\n` : 新行 (换行)
  - `\t` : 制表符 (Tab)

- `\c`: 停止输出, 不输出后面的内容

```shell
echo -e "Line 1\nLine 2\twith a tab."
```

输出:

```
Line 1
Line 2  with a tab.
```

1.1 使用变量

1.1.1 访问和打印变量

- 通过在变量前添加 `$` 访问变量
- 通过 `echo` 命令来打印变量

```shell
$ echo $USER
paciorek
```

Remark

1. 由于 bash 使用空格来解析你输入的表达式, 因此需要确保等号周围没有空格 (否则 bash 会将 counter 视作一个命令)
2. 可以将变量名用花括号括起来, 以确保 shell 知道变量名在哪里结束

```shell
$ base=/home/jarrod/
$ echo ${base}src
/home/jarrod/src
$ echo $basesrc
(empty line)
```

1.1.2 环境变量

环境变量是一类特殊的 Shell 变量, 有助于控制 Shell 的 behavior, 通常以全大写命名

1.1.2.1 查看环境变量

```shell
$ printenv
```

1.1.2.2 创建环境变量

```shell
$ export base=/home/jarrod/
```

- 不使用 `export`: 设置的变量只会在当前 Shell 内生效
- 使用 `export`: 设置的变量会在当前 Shell 及其派生 shell (如运行程序) 中生效
- 在 `.bashrc` 文件中使用 `export`: 设置的变量始终生效

1.2 Introduction to Commands

1.2.1 Elements of a Command

- 一般而言，命令行由 **四个部分** 组成：
  1. **命令（command）**
  2. **选项（options）**
  3. **参数（arguments）**
  4. **执行确认（line acceptance）** —— 按 Enter 键执行

示例：

Shell

```
$ ls -l file.txt
```

- `ls` → 命令
- `-l` → 选项（long format，长格式显示）
- `file.txt` → 参数（指定文件）
- 按下 **Enter** 执行命令

1.2.1.1 Command Execution Process

当用户在 bash 提示符下按 Enter 后，系统会：

1. **解析命令（parse）**
   - bash 会识别命令及其参数。
2. **查找命令来源**
   - 先检查该命令是否为 **shell function（自定义函数）**；
   - 若不是，再查找是否为 **shell builtin（内建命令）**；
   - 若两者都不是，则在 **PATH 变量** 指定的目录中依次查找可执行文件。

Shell

```
$ echo $PATH
/home/jarrod/usr/bin:/usr/local/bin:/bin:/usr/bin:
```

1.2.1.2 Variable Substitution

bash 会在执行命令前进行变量替换：

Shell

```
$ myfile=file.txt
$ grep pdf $myfile
```

执行时会被解析为：

Shell

```
grep pdf file.txt
```

1.3 Command Line 中的高效操作

1.3.1 Command History and Editing

1.3.1.1 Navigating and Reusing Commands

- 使用 ↑ / ↓ 键浏览先前输入的命令。

- 可直接按 Enter 重新运行；
- 或修改后再执行。

- 使用 `history` 查看完整命令记录：

```shell
$ history
1    echo $PS1
2    PS1=$
3    bash
4    export PS1=$
...
11   ls -al manual.xml
```

  - 历史记录行为由以下环境变量控制：

```shell
$ echo $HISTFILE   ## 历史记录文件
$ echo $HISTSIZE   ## 记录的命令条数
```

1.3.1.2 Recalling Commands Quickly

- 通过命令编号或字符串回调：

命令	说明
`!!`	上一条命令
`!n`	第 n 条命令
`!-n`	倒数第 n 条命令
`!string`	最后以 string 开头的命令
`!?string`	最后包含 string 的命令
`^old^new`	替换上一条命令中字符串 old → new

示例：

```shell
$ !-2      ## 执行倒数第二条命令
$ !gi      ## 执行最后以 "gi" 开头的命令
```

Logic

可在结尾加上 `:p` 只打印、不执行：

```shell
$ !gi:p
```

这样可以查看命令内容，再按 ↑ 键编辑或执行。

1.3.1.3 Searching Command History

- **Ctrl + r**：在命令历史中反向搜索匹配字符串。
  - 按 Enter 执行匹配命令。
  - 按 **Ctrl + c** 退出搜索模式。
  - 按 **Esc** 将搜索结果放回命令行以便编辑。

2 Bash Shell-File Management

2.1 查找文件和浏览文件系统

2.1.1 查找文件

可以通过文件名, 修改时间和类型查找文件:

Shell

```
$ find . -name '*.txt'  ## find files named *.txt
$ find . -mtime -2      ## find files modified less than 2 days ago
$ find . -type l        ## find links
```

这里的 `.` 参数表示在当前工作目录及其子目录中查找文件

2.1.2 保留多个先前工作目录的堆栈

除了使用 `cd -` 返回到前一个使用的工作目录外，如果你希望保留多个先前工作目录的堆栈而不是仅保留最后一个，还可以使用 `pushd` (push directory), `popd` (pop directory) 和 `dirs` (directories) 命令

2.1.3 特殊目录

在每个目录中，有两个特殊目录， `.` 和 `..` ，它们分别指向当前目录和当前目录的父目录。只有当我们使用 `-a` 标志来显示隐藏文件时，才能看到这些目录

Shell

```
$ ls -al
total 1489
drwxr-sr-x  7 paciorek scfstaff     31 Apr 21 16:39  ./
drwxr-sr-x 19 paciorek scfstaff     30 Feb 28 15:07  ../
```

2.2 文件名匹配 (globbing 通配符)

Wildcard　通配符	Function　功能
`*`	Match zero or more characters. 匹配零个或多个字符。
`?`	Match exactly one character. 匹配恰好一个字符。
`[characters]`	Match any single character from among *characters* listed between brackets. 匹配括号中列出的任意单个字符。
`[!characters]`	Match any single character other than *characters* listed between brackets. 匹配括号中未列出的任意单个字符。
`[a-z]`	Match any single character from among the range of characters listed between brackets. 匹配括号中列出的字符范围内的任意单个字符。
`[!a-z]`	Match any single character from among the characters not in the range listed between brackets 匹配括号中未列出的字符范围内的任意单个字符
`{frag1,frag2,…}`	Brace expansion: create strings frag1, frag2, etc. 括号扩展：创建字符串 frag1、frag2 等

Example

列出所有以数字结尾的文件:

Shell

```
$ ls *[0-9]
```

Example

将 `filename` 复制为 `filename.old` :

Shell

```
$ cp filename{,.old}
```

Example

删除所有以 a 或 z 开头的文件:

Shell

```
$ rm [az]*
```

Example

列出所有具有各种后缀的 R 代码文件:

Shell

```
$ ls *.{r,R}
```

Example

`echo` 命令可用于验证通配符扩展是否按预期工作:

Shell

```
$ echo cp filename{,.old}
cp filename filename.old
```

Remark

1. 可以使用反斜杠前缀（ `\` ）抑制通配符转义

   Shell

   ```
   $ touch \*test    ## create a file called *test
   $ ls \**
   *test
   ```

2. 要了解更多关于 standard globbing patterns 的信息，可以使用:

Shell

```
$ man 7 glob
```

> ⚠ Remark
> 该指令在 mac 上无法运行, 可以考虑使用 `man 3 glob` 替代

2.3 文件权限

2.3.1 查看文件权限

可以在 `ls` 后添加 `-l` flag 来查看文件权限

Shell

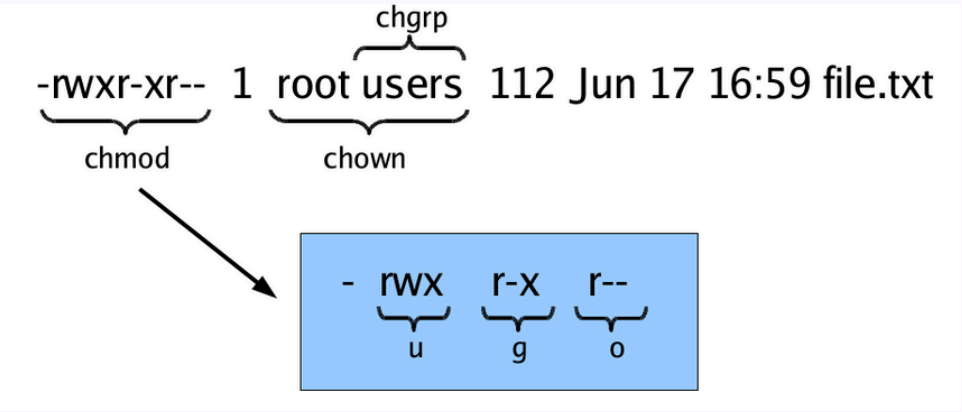
```
$ cd ~/stat243-fall-2020
$ ls -l
```

```
total 152
drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 data
drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 howtos
drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 project
drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 ps
-rw-rw-r-- 1 scflocal scflocal 11825 Dec 28 13:15 README.md
drwxrwxr-x 13 scflocal scflocal 4096 Dec 28 13:15 sections
-rw-rw-r-- 1 scflocal scflocal 37923 Dec 28 13:15 syllabus.lyx
-rw-rw-r-- 1 scflocal scflocal 77105 Dec 28 13:15 syllabus.pdf
drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:37 units
```

其中所包含的重要信息分别是:

- (column 1) file permissions (more later)
  文件权限（稍后详述）
- (column 3) the owner of the file ('scflocal' here)
  文件的所有者（此处为'scflocal'）
- (column 4) the group of users that the file belongs too (also 'scflocal' here)
  文件所属的用户组（此处也为'scflocal'）
- (column 5) the size of the file in bytes
  文件大小（字节）
- (column 6-8) the last time the file was modified
  文件最后修改时间
- (column 9) name of the file
  文件名

> Example



这是名为“file. txt”的文件的一个 graphical summary，其所有者是“root”，组是“users”。（图中还表明可以使用命令 `chmod` 、 `chown` 和 `chgrp` 来更改文件权限和所有者。）

2.3.2 文件权限详解

第一列中包含 10 个 individual single-character columns, 分别表示

- 第 1 个字母: 表示是否是 directory (`d` 表示是, `-` 表示不是)
- 第 2-4 个字母: 表示**文件所有者**是否可以读取 (`r`), 写入 (`w`) 和执行 (`x`),
- 第 5-7 个字母: 表示**文件所属 group 的任何成员**是否可以读取 (`r`), 写入 (`w`) 和执行 (`x`)
- 第 8-10 个字母: 表示**其他任何用户**是否可以读取 (`r`), 写入 (`w`) 和执行 (`x`)
- `-` 表示没有对应权限

Example

例如，对于 syllabus. pdf 文件，文件的所有者可以读取它，并且可以通过写入来修改文件（第一个 triplet 是 `'rw-'` ），文件所属组的用户也可以这样做。但对于其他用户，他们只能读取它（第三个 triplet 是 `'r--'` ）

2.3.3 更改文件权限

可以使用 `chmod` 指令来更改文件权限, 使用时需要指定:

- 用户类型:
  - `u` : 文件所有者
  - `g` : 文件所属组的用户
  - `o` : 其他任何用户
- 进行添加/移除操作:
  - `+` : 添加权限
  - `-` : 移除权限
- 权限类型:
  - `r` : 读取
  - `w` : 写入
  - `x` : 执行

2.3.4 例子: 阻止任何人读取 `tmp.txt` 文件

首先创建文件:

```
$ echo "first line" > tmp.txt  # create a test text file that contains "first line"
$ ls -l tmp.txt
-rw-rw-r--  1 scflocal scflocal    11 Dec 28 13:39 tmp.txt
```

移除读取权限:

```
$ chmod u-r tmp.txt # prevent owner from reading
$ chmod g-r tmp.txt # prevent users in the file's group from reading
$ chmod o-r tmp.txt # prevent others from reading
$ ls -l tmp.txt
--w--w---- 1 scflocal scflocal 11 Dec 28 13:39 tmp.txt
```

或者使用单行指令:

```
$ chmod ugo-r tmp.txt # prevent all three
$ ls -l tmp.txt
--w--w---- 1 scflocal scflocal 11 Dec 28 13:39 tmp.txt
```

此时我们无法读取文件:

```
$ cat tmp.txt
cat: tmp.txt: Permission denied
```

如果想同时移除读取和写入权限, 可以:

```
$ chmod ugo-rw tmp.txt # prevent all three
```

接下来我们把权限恢复给所有者:

```
$ chmod u+rw tmp.txt
$ echo "added line" >> tmp.txt
$ cat tmp.txt
first line
added line
```

2.4 尽可能使用简单的 text files

- UNIX 命令是强大的 text file 操作工具, 因此尽可能地将信息存储在 text files 中是有帮助的
- UNIX 操作文件的基本命令是**逐行操作**的（例如， `grep` 、 `sed` 、 `cut` 等）。因此，使用每行包含不同信息集的格式（如 CSV）比其他存储相关信息在多行中的文本格式（如 XML 和 JSON）更有优势

> ⚠ **Remark**
>
> 对于大型数据集, 仍然建议存储在二进制文件中，特别是考虑
>
> - 数据访问速度是否快
> - 存储格式是否高效

3 Bash Shell-Using Commands

3.1 基本命令

3.1.1 命令中的括号

3.1.1.1 花括号 `{}` 的使用场景

- **变量拓展:**

Shell

```
## 明确变量边界
```

```shell
echo "${variable}_suffix"

## 默认值设置
echo "${VAR:-default_value}"

## 字符串操作
echo "${string:0:5}"    ## 子字符串
echo "${string##prefix}" ## 删除前缀
```

- **序列生成:**

Shell

```shell
## 数字序列
echo {1..5}      ## 输出: 1 2 3 4 5
echo {5..1}      ## 输出: 5 4 3 2 1
echo {01..10}    ## 输出: 01 02 03 04 05 06 07 08 09 10

## 字符序列
echo {a..e}      ## 输出: a b c d e
```

- **文件名拓展 (通配符):**

Shell

```shell
## 创建多个文件
touch file{1..3}.txt   ## 创建 file1.txt, file2.txt, file3.txt

## 复制多个文件
cp file.{txt,bak}      ## 复制 file.txt 到 file.bak
```

- **命令分组:**

Shell

```shell
## 命令组合并重定向
{ command1; command2; } > output.txt

## 注意：花括号内命令必须以分号结尾，且花括号与命令间必须有空格
```

3.1.1.2 圆括号 `()` 的使用场景

- **创建子 Shell（在新进程中执行）**

Shell

```shell
## 在子 Shell 中执行命令
( cd /tmp; ls )        ## 目录改变不会影响当前 Shell

## 后台进程组
( sleep 10; echo "Done" ) &
```

- **命令替换**

Shell

```shell
## 捕获命令输出
current_date=$(date)   ## 等同于 `date`

```

```shell
## 嵌套命令
files=$(ls $(pwd))
```

- **数组定义**

Shell

```shell
## 创建数组
fruits=("apple" "banana" "cherry")

## 访问数组元素
echo "${fruits[0]}"    ## 输出: apple
```

- **算术运算**

Shell

```shell
## 使用 $(( )) 进行算术运算
result=$(( 5 + 3 * 2 ))  ## 输出: 11
```

- **进程替换**

Shell

```shell
## 比较两个命令的输出
diff <(ls dir1) <(ls dir2)
```

3.1.2 重要命令

有许多有用的命令和工具来查看和操作文件:

- `cat` – concatenate 文件与打印到 standard output 详见 cat
- `cp` – 复制文件和目录, 详见 cp
- `cut` – 从文件的每一行中提取部分内容, 详见 cut
- `diff` – 查找两个文件之间的差异, 详见 diff
- `grep` – 打印匹配特定模式的行, 详见 grep
- `head` – 输出文件的开头部分, 详见 head
- `find` – 在 directory hierarchy 中搜索文件, 详见 find
- `less` – 是 `more` 的增强版, 详见 less
- `more` – 用于分页显示文件内容, 详见 more
- `mv` – 移动（重命名）文件, 详见 mv
- `nl` – 为文件中的行编号, 详见 nl
- `paste` – 合并文件中的行, 详见
- `rm` – 删除文件或目录, 详见
- `rmdir` – 删除空目录, 详见
- `sort` – 对文本文件中的行排序, 详见
- `split` – 将文件分割成多个部分, 详见
- `tac` – 反向连接并打印文件, 详见
- `tail` – 输出文件的最后一部分, 详见
- `touch` – 更改文件时间戳, 详见
- `tr` – translate 或 delete 字符, 详见
- `uniq` – 删除 sorted file 中的重复行 (若相邻行中有重复, 则仅保留一个), 详见
- `wc` – 显示文件中的字节数、单词数和行数, 详见
- `wget` and `curl` – non-interactive 式网络下载, 详见

3.1.3 UNIX 命令的一般语法

3.1.3.1 一般语法结构

UNIX 命令的一般语法结构如下:

Shell

```
$ command -options argument1 argument2 ...
```

Example

Shell

```
$ grep -i graphics file.txt
```

会在 `file.txt` （参数 2）中查找字面字符串 `graphics` （参数 1），并且选项 `-i` 表示忽略字母的大小写

Example

Shell

```
$ less file.txt
```

允许我们简单地翻阅文本文件（可以用空格键和上下箭头键上下导航），以便了解文件内容。可以通过输入 `q` 来退出 `less`

3.1.3.2 Options (Flags)

- 单破折号选项（`-`）：短格式（如 `-n 10`）。
- 双破折号选项（`--`）：长格式（如 `--help`）。

  Shell

  ```
  $ tail --help
  ```

- 常见规则：
  - 选项可连写，如 `-al` 等价于 `-a -l`；
  - 参数与选项之间的空格可省略或保留。

3.1.3.3 Example: `tail` Command

Shell

```
$ wget https://raw.githubusercontent.com/berkeley-scf/tutorial-using-bash/master/cpds.csv
$ tail -n 10 cpds.csv    ## 输出文件末尾10行
$ tail -f cpds.csv       ## 实时跟踪文件变化
```

Logic

`wget` 与 `curl` 都可用于下载网络文件。

- Linux 通常自带 `wget`；
- macOS 默认仅提供 `curl`。

3.1.3.4 Example: `grep` Command

`grep` 用于按 **模式（pattern）** 搜索文本。

Shell

```
$ grep ^2001 cpds.csv   ## 以 '2001' 开头的行
$ grep 0$ cpds.csv      ## 以 '0' 结尾的行
$ grep 19.0 cpds.csv    ## 匹配 19任意字符0
$ grep 19.*0 cpds.csv   ## 匹配 19 与 0 之间任意字符
$ grep -o 19.0 cpds.csv ## 仅输出匹配部分
```

Remark

`grep` 使用的 **正则表达式语法** 与文件通配符不同。
在需要匹配特殊字符（如 `.`、`"`）时必须使用反斜杠转义：

Shell

```
$ grep "\"Canada\"" cpds.csv   ## 匹配含引号的 "Canada"
$ grep "19\.0" cpds.csv        ## 匹配文本 19.0
```

3.1.3.5 Quoting Patterns

在包含空格或特殊字符的模式时，使用引号可避免 shell 错误解析：

Shell

```
$ grep "George .* Bush" cpds.csv
```

Logic

用双引号包裹字符串能确保它被视为单一参数。
在 shell 中 `"pattern with space"` 是安全做法。

3.1.3.6 Example: Working with Large Data Files

- 使用 `grep` 按行筛选数据，或使用 `cut` 提取字段：

  Shell

  ```
  $ grep "Canada" bigdata.csv > subset.csv
  $ cut -d',' -f1,3 subset.csv
  ```

- 相比在 R/Python/SAS 中读入大文件，这种方式速度更快且占用更少内存。

Logic

Unix 工具如 `grep`, `cut`, `awk`, `sort`, `uniq` 能在命令行快速处理 GB 级文件, 常被用于数据预处理和日志筛查。

3.2 Streams, Pipes, and Redirects

3.2.1 Streams (stdin / stdout / stderr)

在 Unix 系统中，程序通过 **数据流 (streams)** 与外部交互：

名称	缩写	默认方向	默认设备	说明
标准输入	`stdin`	输入	键盘	程序从此处读取数据
标准输出	`stdout`	输出	屏幕	程序的正常输出结果
标准错误	`stderr`	输出	屏幕	程序的错误信息与警告

在交互式 Shell 会话中：

- **输入** 默认来自键盘；
- **输出** 与 **错误信息** 默认显示在屏幕。

Logic

通过重定向（redirection），可改变这三个流的输入输出位置，
例如将输出保存到文件、将输入读取自文件，或隐藏错误信息。

3.2.2 Overview of Redirection

Shell 提供了通用的 **重定向操作符**，可改变程序的输入输出位置。
下表总结了常见语法与功能：

重定向语法	功能说明
`cmd > file`	将标准输出（stdout）写入文件（覆盖原内容）
`cmd 1> file`	同上（1 代表 stdout）
`cmd 2> file`	将标准错误（stderr）写入文件
`cmd > file 2>&1`	将 stdout 与 stderr 同时写入文件
`cmd < file`	从文件中读取输入（stdin）
`cmd >> file`	将标准输出追加到文件末尾
`cmd 2>> file`	将标准错误追加到文件末尾
`cmd >> file 2>&1`	同时追加 stdout 与 stderr
cmd1 | cmd2	将 cmd1 的标准输出输入到 cmd2
cmd1 2>&1 | cmd2	将 cmd1 的标准输出和标准错误输入到 cmd2
cmd1 | tee file1 | cmd2	将 cmd1 的标准输出写入 file1 的同时输入到 cmd2

Remark

重定向由 **Shell 提供**，并非具体命令的特性。
因此这些语法适用于所有标准 Unix 程序。

3.2.3 Standard Redirection (Pipes 管道)

管道 (pipe) 操作符 `|`
用于将一个命令的输出（stdout）作为下一个命令的输入（stdin）。

3.2.3.1 示例 1：统计字符串单词数

Shell

```
$ echo "hey there" | wc -w
2
```

3.2.3.2 示例 2：大小写转换

Shell

```
$ echo "user1" | tr 'a-z' 'A-Z'
USER1
```

3.2.3.3 示例 3：提取数据文件第二列中的唯一条目数

Shell

```
$ cut -d',' -f2 cpds.csv | sort | uniq | wc
```

或保存结果到文件：

Shell

```
$ cut -d',' -f2 cpds.csv | sort | uniq > countries.txt
```

⚠ Remark: 执行逻辑分解

1. `cut -d',' -f2 cpds.csv`
   - 提取以逗号分隔的第二列；
2. `sort`
   - 对输出进行排序；
3. `uniq`
   - 删除重复值（仅保留唯一条目）；
4. `wc`
   - 统计输出行数、单词数与字节数；
5. `>`
   - 将最终输出保存为文件。

Logic

许多 Unix 命令（如 `sort`, `grep`, `cut`, `wc`）
若未指定文件名，会自动从 **stdin** 读取输入，
这使得管道机制能自由组合命令链。

3.2.3.4 示例 4：大规模文件检测

查找 22,000 个文件（5GB 数据）中是否有字段值为 “S”：

Shell

```
$ cut -b29,37,45,53,61,69,77,85,93,101,109,117,125,133,141,149, \
      157,165,173,181,189,197,205,213,221,229,237,245,253, \
      261,269 USC*.dly | grep S | less
```

⚠ Remark

此命令仅用约 5 分钟即可完成，
若用 R 或 Python 读入同样体量的数据则可能耗时数小时。

3.2.4 The `tee` Command

`tee` 命令可 **将一个流复制成两份**：

- 一份传递到下一个命令；
- 另一份保存到文件。

3.2.4.1 示例

传统方式（重复执行两次命令）：

Shell

```
$ cut -d',' -f2 cpds.csv | sort | uniq
$ cut -d',' -f2 cpds.csv | sort | uniq > countries.txt
```

更高效方式（使用 `tee`）：

Shell

```
$ cut -d',' -f2 cpds.csv | sort | uniq | tee countries.txt
```

输出效果：

- 在终端显示结果；
- 同时写入文件 `countries.txt`。

Logic

`tee` 是数据分析中常用的辅助工具，
特别适用于监控长时间运行的管道命令结果。

3.3 Command Substitution and the `xargs` Command

3.3.1 Command Substitution

Command substitution 允许我们把 **一个 command 的 output** 作为另一个 command 的 **argument**

其语法为：

Shell

```
$(command)
```

当 shell 遇到 `$()` 包裹的命令时，它会：

1. **执行括号内的命令**；
2. **将该命令的输出结果替换到当前位置**。

这与 **管道 (|)** 相似，但适用于 **命令需要从命令行参数读取数据** 的情况，而不是从标准输入 (`stdin`) 读取。

3.3.1.1 Example

假设我们想在当前目录下的 **最新修改的四个 R 文件** 中查找文本 `pdf`：

Shell

```
$ grep pdf $(ls -t *.{R,r} | head -4)
```

执行逻辑：

1. `ls -t *.{R,r}` → 列出以 `.R` 或 `.r` 结尾的文件，按修改时间排序；
2. `head -4` → 取前四个文件名；
3. `$( ... )` → 将上一步输出作为 `grep` 的命令行参数；
4. Shell 最终执行：

Shell

```
grep pdf test.R run.R analysis.R process.R
```

Remark

如果改用管道 `ls -t *.{R,r} | head -4 | grep pdf`，将 不会 达到同样效果，因为 `grep` 读取的文件名来自命令行参数，而非标准输入。

3.3.1.2 总结规律表

命令行为	数据来源	典型命令	举例
只从参数读	命令行参数（文件名、字符串等）	`ls`，`rm`，`cp`，`echo`	`ls /home`
默认从 stdin 读（可选参数）	标准输入或文件参数	`cat`，`grep`，`sort`，`wc`，`awk`，`cut`	`grep "a" file.txt` 或 `echo hi | grep "h"`
只从 stdin 读	标准输入流	一些专用工具或交互式程序	`cat`，`tr`，`head`
可从两者读	根据是否提供参数而定	`grep`，`awk`，`wc`，`sort`	都可用

3.3.2 The `xargs` Command

管道无法直接将输出作为“命令行参数”传递，
但可以使用 `xargs` 工具实现这一功能。

示例：

Shell

```
$ ls -t *.{R,r} | head -4 | xargs grep pdf
```

执行逻辑与上节等价于：

Shell

```
$ grep pdf $(ls -t *.{R,r} | head -4)
```

Logic

- `xargs` 将标准输入（stdin）的内容拼接成命令行参数；
- 常用于解决“一个命令输出 → 另一个命令参数”类型的问题。

3.3.2.1 Exercise

请尝试以下命令，理解命令替换与 `xargs` 的区别：

Shell

```
$ ls -l tr  ## 若当前目录下没有 tr, 则会报错
```

```
$ type -p tr    ## /usr/bin/tr
$ ls -l type -p tr  ## 报错
$ ls -l $(type -p tr)   ## -rwxr-xr-x 1 root root 43840 Jan 19  2024 /usr/bin/tr
```

提示：

- `type -p tr` 输出命令路径；
- `$(type -p tr)` 将路径替换进 `ls -l` 命令中。

3.4 Brace Expansion

3.4.1 Overview

Brace expansion（花括号扩展） 是 shell 的一种语法功能，用于自动生成多个字符串或文件名。
Shell 在执行命令前，会先展开花括号的内容。

3.4.1.1 Example 1: 文件重命名

Shell

```
$ mv my_long_filename.{txt,csv}
$ ls my_long_filename*
my_long_filename.csv
```

相当于：

Shell

```
mv my_long_filename.txt my_long_filename.csv
```

再例如：

Shell

```
$ mv my_long_filename.csv{,-old}
$ ls my_long_filename*
my_long_filename.csv-old
```

等价于：

Shell

```
mv my_long_filename.csv my_long_filename.csv-old
```

3.4.1.2 Example 2: 使用序列展开

Shell

```
$ echo {1..15}
$ echo c{c..e}
$ echo {d..a}
$ echo {1..5..2}
$ echo {z..a..-2}
```

说明：

- `{1..15}` → 从 1 到 15；
- `{c..e}` → c, d, e；
- `{1..5..2}` → 从 1 到 5，步长为 2；

- `{z..a..-2}` → 从 z 向 a 逆序，步长为 2。

可用于批量命令，例如结束多个连续进程：

```shell
$ kill 1397{62..81}
```

3.5 Quoting

3.5.1 Single vs. Double Quotes

引号类型	名称	特性
`' '`	硬引用 (hard quote)	禁止变量替换
`" "`	软引用 (soft quote)	允许变量替换

3.5.1.1 Example

```shell
$ echo "My home directory is $HOME"
My home directory is /home/jarrod

$ echo 'My home directory is $HOME'
My home directory is $HOME
```

3.5.2 Handling Spaces in Filenames

当路径或文件名中包含空格时，
可用转义符或引号防止 shell 将空格误认为参数分隔符。

```shell
$ ls $HOME/with\ space
file1.txt
```

或更简洁的写法：

```shell
$ ls "$HOME/with space"
file1.txt
```

3.5.3 Escaping Double Quotes

若目录或文件名本身包含双引号 (`"with"quote`)：

```shell
$ ls "$HOME/\"with\"quote"
```

通过 `\"` 转义内部双引号。

3.6 Powerful Tools for Text Manipulation: `grep`, `sed`, and `awk`

3.6.1 Overview

优势：

- 逐行处理文件（line by line），内存占用低；
- 能高效处理大型文本文件（如日志、CSV、配置文件）；
- 可结合管道与正则表达式实现复杂操作。

3.6.2 `grep`

`grep` 是最常用的文本搜索命令。
它用于打印文件中符合指定 **模式（pattern）** 或 **正则表达式** 的行。

3.6.2.1 基本用法

假设我们有文件 `testfile.txt`：

```
This is the first line.
Followed by this line.
And then ...
```

1. 查找包含某模式的行

Shell

```
$ grep is testfile.txt
This is the first line.
Followed by this line.
```

2. 查找不包含该模式的行

Shell

```
$ grep -v is testfile.txt
And then ...
```

`-v` 表示反选（打印不匹配的行）。

3. 只打印匹配内容

Shell

```
$ grep -o is testfile.txt
is
is
is
```

4. 彩色高亮匹配结果

Shell

```
$ grep --color is testfile.txt
```

Logic

`grep` 名称来自 `ed` 命令 **g/re/p**，即 “globally search for a regular expression and print”。

3.6.3 sed

`sed`（**stream editor**）是一个强大的流式文本编辑器。
它逐行读取输入，对匹配的行执行替换、删除或打印操作。
默认输出结果到 **stdout**，除非使用 `-i` 参数修改文件本身。

3.6.3.1 打印特定行

Shell

```shell
$ sed -n '1,9p' file.txt       ## 打印第 1-9 行
$ sed -n '/^##/p' file.txt      ## 打印以 '##' 开头的行
```

- `-n`：抑制默认输出，仅输出匹配结果；
- `/^##/`：正则表达式，匹配行首为 `##` 的行。

3.6.3.2 删除特定行

Shell

```shell
$ sed -e '1,9d' file.txt
$ sed -e '/^;/d' -e '/^$/d' file.txt
```

- 第一行：删除第 1–9 行；
- 第二行：
  - `/^;/d` 删除以分号开头的行；
  - `/^$/d` 删除空行。

Logic

`-e` 用于同时执行多个表达式，
若只需一个命令可省略。

3.6.3.3 文本替换

Shell

```shell
$ sed 's/old_pattern/new_pattern/' file.txt > new_file.txt
$ sed 's/old_pattern/new_pattern/g' file.txt > new_file.txt
$ sed -i 's/old_pattern/new_pattern/g' file.txt
```

选项	说明
无 `g`	每行只替换第一个匹配项
`g`	全局替换行内所有匹配项
`-i`	原地修改文件（谨慎使用）

Remark

使用 `-i` 时不会自动备份，
建议先输出到新文件以防止数据丢失。

3.6.4 awk

`awk` 是一种专为文本与表格数据处理设计的 **轻量编程语言**。
它按行处理文件，并根据条件执行操作。
其语法结构为：

```
awk 'pattern { action }' file
```

3.6.4.1 Example 1: 选择列

Shell

```
$ ps -f | awk '{ print $2 }'
```

输出进程列表中的第 2 列（PID）。

> ⚠ Remark
>
> `$1`, `$2`, … → 对应第 1、2、… 列；
> `$0` → 表示整行。

3.6.4.2 Example 2: 文件双倍行距

Shell

```
$ awk '{ print } { print "" }' file.txt
```

- 第一个 `{ print }` 输出原行；
- 第二个 `{ print "" }` 输出空行。

3.6.4.3 Example 3: 筛选长行

Shell

```
$ awk 'length($0) > 80' file.txt
```

输出长度大于 80 字符的行。

3.6.4.4 Example 4: 提取用户主目录

Shell

```
$ awk -F: '{ print $6 }' /etc/passwd
```

解析说明：

- `-F:` → 设置分隔符为冒号 `:`；
- `$6` → 第 6 个字段，对应用户主目录。

查看文件格式：

Shell

```
$ head -n 1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

结果：

```
/root
```

3.6.4.5 Example 5: 列求和

Shell

```
$ awk '{ print $1 + $2 }' file.txt
```

输出文件中第 1 列与第 2 列的和。

Logic

`awk` 既可处理文本，也可执行算术操作，
是轻量数据分析和日志提取的理想工具。

3.6.5 Summary

工具	功能	特点
`grep`	搜索匹配行	支持正则表达式，快速定位文本
`sed`	编辑文本流	支持替换、删除、打印、批量修改
`awk`	结构化文本处理	支持条件判断、字段操作与计算

Logic

这三者常搭配使用：

- `grep` 用于过滤；
- `sed` 用于编辑；
- `awk` 用于分析。

4 Bash Shell-Shell Programming

4.1 Shell Scripts

4.1.1 Overview

Shell scripts 是包含一系列 shell 命令的文本文件（通常以 `.sh` 结尾）。
通过编写脚本，你可以自动化一组命令的执行，而不必在终端中手动输入每一行。

4.1.1.1 运行 Shell 脚本的方式

4.1.1.1.1 方法 1：使用 `source` 或 `.`

Shell

```
$ source ./file.sh
## 或
$ . ./file.sh
```

- `source`（或 `.`）**在当前 shell 环境中**执行脚本内容；
- 脚本中定义的变量和函数会保留在当前会话中。

4.1.1.1.1.2 方法 2：直接运行脚本

如果直接输入脚本名（如 `file.sh`），可能遇到以下问题：

1. **找不到文件**（脚本未在 `$PATH` 路径中）；
2. **缺乏执行权限**（未设置可执行标志 `-x`）。

正确方式：

1. 在脚本开头声明解释器（shebang）：

   ```shell
   ##!/bin/bash
   ```

   告诉系统使用 Bash 来解释执行脚本。

2. 赋予可执行权限：

   ```shell
   $ chmod +x file.sh
   ```

3. 执行脚本：

   ```shell
   $ ./file.sh
   ```

4.2 Functions

4.2.1 Overview

定义语法：

```shell
function name() {
   commands
}
```

调用方式：

```shell
$ name arg1 arg2 ...
```

4.2.2 参数传递

在函数中，Bash 会自动创建以下特殊变量：

变量	含义
`$1`, `$2`, `$3`, ...	依次表示第 1、2、3 个参数
`$##`	参数个数
`$@`	所有参数（以空格分隔）

4.2.2.1 Example: 自定义上传函数

Shell

```shell
function putscf() {
   scp $1 jarrod@arwen.berkeley.edu:$2
}
```

执行：

Shell

```shell
$ putscf unit1.pdf teaching/243/.
```

该命令将 `unit1.pdf` 上传到远程服务器目录 `~/teaching/243/`。

4.3 If / Then / Else

4.3.1 控制流结构

Shell 支持 **条件分支**（if-then-else）语法：

Shell

```shell
if [ condition ]; then
   commands
elif [ other_condition ]; then
   commands
else
   commands
fi
```

- 条件判断语句**必须用方括号包裹**（`[ ]`），
- 每个部分之间**需要空格**，
- 结尾**必须用 `fi` 结束**。

4.3.2 Example: `niceR` 函数

Shell

```shell
## niceR: 提交优先级较低的 R 作业
## 用法: niceR inputRfile outputRfile
## Author: Brian Caffo

function niceR() {
    if [ $#\# != "2" ]; then
         echo "usage: niceR inputRfile outputfile"
    elif [ -e "$2" ]; then
         echo "$2 exists, I won't overwrite"
    elif [ ! -e "$1" ]; then
         echo "inputRfile $1 does not exist"
    else
         echo "running R on $1"
         nice -n 19 R --no-save < $1 &> $2
    fi
}
```

- `-e` 检查文件是否存在；

- `! -e` 检查文件是否不存在；
- `nice -n 19` 表示降低进程优先级，以免占用过多 CPU；
- `&>` 同时重定向 stdout 与 stderr。

4.3.3 Example: 字符串比较

Shell

```shell
var="some text"

if [ "${var}" == "some text" ]; then
   echo "found equal"
fi

if [ "${var}" != "some text" ]; then
   echo "found not equal"
fi
```

> ⚠ **Remark**
>
> 建议始终为变量加双引号，避免空格或通配符误解析。

4.4 For Loops

4.4.1 Overview

在 Bash 中，**for 循环** 用于遍历一组文件、目录或变量值。

4.4.2 Example 1: 文件批量重命名

Shell

```shell
$ for FILE in $(ls *.txt); do
>     mv $FILE ${FILE/.txt/.R}
>     ## 将文件扩展名 .txt 替换为 .R
> done
```

> **Logic**
>
> - `${FILE/.txt/.R}` 是 **参数替换语法**，将变量内容中的 `.txt` 替换为 `.R`；
> - `>` 提示符表示 Shell 等待多行输入（循环尚未结束）。

4.4.3 Example 2: 自动化文件下载

Shell

```shell
url='ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/grid/years'
types="tmin tmax"

for ((yr=1950; yr<=2017; yr++))
do
    for type in ${types}
    do
        wget ${url}/${yr}.${type}
    done
done
```

说明：

- 双层循环：外层遍历年份，内层遍历文件类型；
- `for (( ... ))` → 数值循环语法（C 风格）；
- `wget` 从 FTP 服务器批量下载文件。

4.4.4 Example 3: 批量启动任务

Shell

```
n=100
for (( it=1; it<=100; it++ ))
do
    echo "n=$n; it=$it; source('base.R')" > tmp-$n-$it.R   ## 创建定制 R 文件
    R CMD BATCH --no-save tmp-$n-$it.R sim-n$n-it$it.Rout  ## 执行任务
done
## 注意 base.R 不应在脚本中定义 n 或 it
```

4.4.5 Example 4: 自定义分隔符循环

默认情况下，`for` 循环以 空格 为分隔符。
若要更改分隔符，可修改变量 `IFS`（Internal Field Separator）。

Shell

```
$ IFS=:
$ types=tmin:tmax:pmin:pmax
$ for type in $types
> do
>     echo $type
> done
```

输出：

```
tmin
tmax
pmin
pmax
```

Logic

`IFS` 控制 Bash 如何拆分字符串。
该技巧常用于解析 CSV、路径或自定义格式字符串。

5 Bash Shell-Regular Expression

5.1 概述与核心语法

正则表达式（regex）是一种用于匹配文本模式的领域特定语言（DSL），广泛用于 Python、R、UNIX 工具（如 `sed`、`awk`、`grep`）等环境中。

5.1.1 主要用途：

- 提取文本（如电话号码）
- 从文本中创建变量
- 清洗和格式化文本

- 文本挖掘
- 网页数据抓取

5.1.2 正则表达式由三部分组成：

1. **Literal characters**：字面匹配
2. **Character classes**：匹配某一类字符中的任意一个
3. **Modifiers**：修饰符，用于控制重复、位置等

5.1.3 特殊字符（元字符）：

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

若要匹配这些字符本身，需使用反斜杠 `\` 进行转义（在 R 中需使用两个反斜杠 `\\`）。

5.2 字符集与字符类

5.2.1 运算符说明：

表达式	说明
`[abc]`	匹配任意一个列出的字符
`[a-z]`	匹配任意一个范围内的字符
`[^abc]`	匹配任意一个不在列出的字符
`[^a-z]`	匹配任意一个不在范围内的字符
`.`	匹配除换行符外的任意字符
`\`	转义元字符的特殊含义

> **Remark**
>
> - 若在 `[]` 中加入 modifier (如 `.`, `?` 等), 则这些 modifier 不会生效
> - 但是需要注意不要和 shell 的一些特殊运算符冲突, 例如 `[.?!]` 会报错, 因为 `!` 在 Shell 中用于历史扩展 (History Expansion), 此时需要使用转义字符

5.2.2 示例：

```shell
## 匹配包含数字的行
grep -E [0-9] test.txt

## 只输出匹配的数字
grep -E -o [0-9] test.txt
```

5.2.3 命名字符类（Named Character Classes）：

使用 `[:CLASS:]` 格式，如 `[:digit:]`、`[:punct:]` 等。

```shell
## 匹配包含标点符号的行
```

```
grep -E [[:punct:]] test.txt

## 匹配数字、点或逗号
grep -E [[:digit:].,] test.txt
```

5.3 位置匹配

5.3.1 运算符说明：

表达式	说明
`^`	匹配行首
`$`	匹配行尾

5.3.2 示例：

Shell

```
## 匹配以数字开头的行
grep -E ^[0-9] test.txt

## 匹配以数字结尾的行
grep -E [0-9]$ test.txt
```

5.4 重复、分组与引用

5.4.1 修饰符说明：

表达式	说明
`*`	匹配 0 次或多次
`?`	匹配 0 次或 1 次
`+`	匹配 1 次或多次
`{n,m}`	匹配 n 到 m 次
`|`	匹配左边或右边的表达式

5.4.2 示例：

Shell

```
## 匹配 http 或 ftp
grep -E -o "(http|ftp)" test.txt

## 匹配电话号码格式
egrep '(1[-.])?[[:digit:]]{3}[-.][[:digit:]]{3}[-.][[:digit:]]{4}' file2.txt
```

5.5 贪婪匹配

默认情况下，正则表达式是“贪婪”的，即匹配尽可能长的字符串。

5.5.1 示例：

```shell
## 贪婪匹配
grep -o "<.*>" file1.txt
## 输出: <b> in place </b> of <b> one </b>

## 非贪婪匹配 (Perl 语法)
grep -P -o "<.*?>" file1.txt
## 输出: <b> </b> <b> </b>
```

5.5.2 避免贪婪匹配的技巧：

使用更精确的字符集，避免使用 `.*`，例如：`<[^>]*>`

> ⚠ **Remark: 注意 Globbing 与 Regex 的区别**
>
> - **Globbing**：用于文件名匹配，`*` 表示任意字符序列
> - **Regex**：用于文本模式匹配，`*` 表示前一个字符的重复

6 Examples and Challenges

6.1 Bash Shell Examples

6.1.1 示例一：统计不同州的气象站数量

目标：从压缩文件 `coop.txt.gz` 中分析每个州的气象站数。

核心命令：

```shell
cd fall-2025/data
gzip -cd coop.txt.gz | less        ## 查看压缩文件内容
cut -b60-61 coop.txt | sort | uniq -c  ## 提取第60-61列（州代码）并统计频数
```

说明：

- `gzip -cd`：解压并输出到标准输出。
- `cut -b60-61`：按字节位置提取州代码字段。
- `sort | uniq -c`：统计各州出现次数。
- 也可以直接在一行中完成：

```shell
gzip -cd coop.txt.gz | cut -b60-61 | sort | uniq -c
```

要点：这是快速查看文本数据结构和分布的命令行替代方案，无需用 Python/R 加载数据。

6.1.2 示例二：程序化计算 CSV 文件字段数

目标：自动计算 CSV 文件中字段的数量。

核心命令：

```shell
tail -n 1 cpds.csv | grep -o ',' | wc -l
nfields=$(tail -n 1 cpds.csv | grep -o ',' | wc -l)
nfields=$((${nfields}+1))
echo $nfields
```

说明：

- `tail -n 1`：取文件最后一行。
- `grep -o ','`：提取所有逗号。
- `wc -l`：统计逗号数。
- 由于字段数 = 逗号数 + 1，因此再加 1。
- 也可以使用 `bc` 计算：

```shell
nfields=$(echo "${nfields}+1" | bc)
```

延伸：可以写成函数，检查所有行字段数是否一致。

6.1.3 示例三：查找最近修改的 Quarto 文件中是否使用 `requests` 包

目标：判断 `requests` 是否出现在最近 5 个 `.qmd` 文件中。

核心命令：

```shell
cd ../units
ls -tr *.qmd | tail -n 5 | xargs grep -l 'import requests'
```

说明：

- `ls -tr`：按修改时间排序。
- `tail -n 5`：取最近的 5 个文件。
- `xargs grep -l`：在这些文件中搜索包含 `import requests` 的文件。
- `xargs` 将标准输入（stdin）转换为命令参数（arguments）。

替代写法（命令替换）：

```shell
grep -l 'import requests' $(ls -tr *.qmd | tail -n 5)
```

要点：展示了命令间信息传递的三种方式：

- `|` 管道（stdout → stdin）
- `$()` 命令替换
- `>` 文件重定向

6.1.4 示例四：移动最近下载的 n 个文件

目标：编写函数，把最近下载的 n 个文件移动到指定目录。

核心命令：

```shell
function mvlast() {
```

```
    for ((i=1; i<=${1}; i++)); do
        mv "/accounts/vis/paciorek/Downloads/$(ls -rt \
           /accounts/vis/paciorek/Downloads | tail -n 1)" ${2}
    done
}
```

说明：

- `${1}`：第一个参数，文件数。
- `${2}`：第二个参数，目标目录。
- `ls -rt`：按时间排序（最旧→最新）。
- `tail -n 1`：取最近一个文件。
- 引号确保文件名中有空格时不出错。

示例用法：

Shell

```
mvlast 3 ~/Desktop
```

6.1.5 示例五：自动提取所有 qmd 文件中的 Python 包并安装

目标：自动找出所有 `.qmd` 文件中导入的 Python 包并生成 `requirements.txt`。

核心命令：

Shell

```
grep --no-filename "^import " *.qmd | cut -d'##' -f1 | \
    sed "s/as .*//" | sed "s/import //" > tmp.txt
sed "s/,/\n/g" tmp.txt | sed "s/ //g" | sort | uniq | tee requirements.txt
pip install -r requirements.txt
```

说明：

- `grep "^import "`：提取所有以 `import` 开头的行。
- `cut -d'##' -f1`：删除注释。
- `sed`：去掉 `as` 别名并只保留包名。
- `tee`：同时输出到文件和终端。
- 最终生成 `requirements.txt` 并安装依赖。

要点：展示如何用 shell 快速自动化文本分析与环境管理任务。

6.1.6 示例六：批量终止后台 Python 任务

目标：若误启动多个 Python 进程，批量终止。

核心命令：

Shell

```
nJobs=30
for (( i=1; i<=${nJobs}; i++ )); do
   python job.py > job-${i}.out &
done

ps -o pid --sort=start_time -C python | tail -n ${nJobs} | xargs kill
```

说明：

- `&`：后台运行。
- `ps -o pid -C python`：列出 Python 进程 ID。
- `tail -n`：取最近的几个。
- `xargs kill`：逐个杀死进程。
- Mac 上略有不同，需用 `grep python` 获取 PID。

要点：展示如何通过管道与 `xargs` 结合批量操作进程。

6.2 Bash Shell Challenges

6.2.1 First Challenge — 统计单词出现次数

目标：统计文件中某个单词出现的次数，并打印为完整句子。

Shell

```shell
## 基本版（以 Belgium 为例）
count=$(grep -o "Belgium" cpds.csv | wc -l)
echo "There are ${count} occurrences of the word 'Belgium' in this file."
```

6.2.2 Second Challenge — 检查字段是否为数字

目标：

1. 找出第 4 列的唯一值。
2. 检查这些值中是否有非数字。

Shell

```shell
cut -d',' -f4 RTADataSub.csv | sort | uniq > uniq_field4.txt
grep -E '[^0-9]' uniq_field4.txt
```

或只使用一行代码:

Shell

```shell
cut -d',' -f4 RTADataSub.csv | sort | uniq | grep -E '[^0-9]'
```

6.2.3 Third Challenge — 各国最低失业率

目标：

1. 找出 Belgium 的最小失业率（第 6 列）。
2. 自动计算所有国家的最小值并输出。

Shell

```shell
## Belgium
grep "Belgium" cpds.csv | cut -d',' -f6 | sort -n | head -1
## 输出示例：6.2
```

Remark

`sort` 后必须加上 `-n`，表示按照数值大小排序，否则默认会按照字符串顺序排序

完整自动化版本

Shell

```
countries=$(cut -d',' -f1 cpds.csv | tr -d '"' | sort | uniq)
for c in $countries; do
    minval=$(grep "$c" cpds.csv | cut -d',' -f6 | sort -n | head -1)
    echo "$c $minval"
done
```

6.2.4 Fourth Challenge — 删除含缺失值的行

目标：创建一个不含缺失符号 “x” 的新文件。

Shell

```
grep -v "x" RTADataSub.csv > RTADataSub_clean.csv
```

扩展：写成函数，可统计删除行数并可指定缺失符号

Shell

```
function clean_missing() {
    local missing=$1
    local infile=$2
    local removed=$(grep "${missing}" "${infile}" | wc -l)
    echo "${removed} rows removed."
    grep -v "${missing}" "${infile}"
}

## 用法示例（输出可用于管道）：
clean_missing x RTADataSub.csv > clean.csv
```

6.2.5 Fifth Challenge — 自动提取州字段位置

目标：利用 `grep` 找到州字段在 coop.txt 中的起始字节位置。

Shell

```
grep -b "CA US" coop.txt | head -1
```

输出示例：

```
59:CA US
```

→ 州字段从第 60 字节开始。

自动提取位置并用于 cut

Shell

```
offset=$(grep -b "CA US" coop.txt | head -1 | cut -d':' -f1)
```

```
start=$((offset + 1))
end=$((start + 1))
cut -b${start}-${end} coop.txt | sort | uniq -c
```

6.2.6 Sixth Challenge — 替换分隔符以避免嵌套逗号问题

目标：将含双引号内逗号的 CSV 改成用其他分隔符（如 `|`）的文件。

示例输入：

```
1,"America, United States of",45,96.1,"continental, coastal"
2,"France",33,807.1,"continental, coastal"
```

解决方案（简单替换）

Shell

```
sed 's/","/"|"/g' input.csv | sed 's/,/|/g' | sed 's/|"/","/g' > output.csv
```

输出示例：

```
1,"America - United States of",45,96.1,"continental - coastal"
2,"France",33,807.1,"continental - coastal"
```

6.3 Assignment 1 题目

6.3.1 Question 1 — 获取 Python 路径

Shell

```
mypython=$(which python)
```

6.3.2 Question 2 — 用户名 + 主机名

Shell

```
username_machinename=$USER@$(hostname)
```

6.3.3 Question 3 — 一行创建复杂目录结构

Shell

```
mkdir -p temp/proj{1,2,3}/{code,data}
```

6.3.4 Question 4 — 统计文件行数

Shell

```
wc -l < data.txt          ## 常规情况
grep -c "" data.txt       ## 若最后一行缺少换行符
```

6.3.5 Question 5 — 打印前 3 行与第 3 行

Shell

```shell
head -3 FILENAME
head -3 FILENAME | tail -1
```

6.3.6 Question 6 — 将第 3 行写入新文件

Shell

```shell
head -3 FILENAME | tail -1 > NEWFILENAME
```

6.3.7 Question 7 — 追加第 5 行到同一文件

Shell

```shell
head -5 FILENAME | tail -1 >> NEWFILENAME
```

6.3.8 Question 8 — 提取 Australia 数据

Shell

```shell
grep "Australia" cpds.csv > cpds_australia.csv
## 或更严格匹配：
grep -E '^[^,]*,"Australia",' cpds.csv > cpds_australia.csv
```

6.3.9 Question 9 — 查找不含逗号的行

Shell

```shell
grep -v ',' FILENAME
## 或：
grep -E '^[^,]*$' FILENAME
```

6.3.10 Question 10 — 批量创建文件

Shell

```shell
for i in {1..N}; do
    echo "blah" > "file${i}.txt"
done
```

6.4 Assignment 2 题目

6.4.1 Question 1 — 匹配任意大小写形式的 “dog”

Shell

```shell
grep -E "[Dd][Oo][Gg]" FILENAME      ## 正则匹配
grep -i "dog" FILENAME               ## 忽略大小写
```

6.4.2 Question 2 — 匹配 “cat”、"caat"、"caaat" 等

Shell

```shell
grep -E "ca+t" FILENAME
grep -E "caa*t" FILENAME
grep -E "ca{1,}t" FILENAME
```

6.4.3 Question 3 — 匹配 “cat”、"at"、"t"

Shell

```shell
grep -E "c?a?t" FILENAME     ## 允许可选的 c 和 a
grep -E "cat|at|t" FILENAME  ## 明确枚举匹配
```

6.4.4 Question 4 — 匹配由任意空白分隔的两个单词

Shell

```shell
grep -E "[[:alnum:]]+[[:space:]]+[[:alnum:]]+" FILENAME
grep -E "[A-Za-z]+[[:space:]]+[A-Za-z]+" FILENAME   ## 仅限英文字母
```