

| STAT243 Lecture 3 Bash Shell

| 1 Bash Shell-Overview

| 1.1 访问 Shell

| 1.1.1 查看默认 Shell

```
>- Shell
1 $ echo $SHELL
2 /bin/bash
```

⚠ 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
1 echo "Hello, world!"
输出:
1 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
1 name="Alice"
2 echo "My name is $name."
输出:
1 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
1 echo "This is the first line." > file.txt
2 echo "This is the second line." >> file.txt
运行后, file.txt 文件内容为:
1 This is the first line.
2 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
1 echo -n "Starting..."
```

```
2 echo " Done."
```

输出:

```
1 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

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

输出:

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

1.1.2 切换到 Bash (one-time basis)



Shell

```
1 $ bash
```

1.1.3 设置为默认



Shell

```
1 $ chsh /bin/bash
```

其中的路径应该为 bash shell 的路径, 可以通过以下方法确定:



Shell

```
1 $ type bash
2 bash is /usr/bin/bash
```

1.2 使用变量

1.2.1 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

```
1 $ echo $USER
2 paciorek
```

1.2.2 声明变量

声明变量时无需使用 **\$**



Shell

```
1 $ counter=1
```

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
1  $ base=/home/jarrodd/
2  $ echo ${base}src
3  /home/jarrodd/src
4  $ echo $basesrc
5  (empty line)
```

1.2.3 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

1.2.3.1 查看环境变量

```
>- Shell
1  $ printenv
```

1.2.3.2 创建环境变量

```
>- Shell
1  $ export base=/home/jarrodd/
```

- 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

1.2.3.3 控制 Bash prompt 的外观

```
>- Shell
1  $ echo $PS1
2  \s-\v\$\n
```

可以通过对 `PS1` 进行以下修改来显示 username, hostname, 和 current working directory

```
>- Shell
1  $ export PS1='\u@\h \W\$\n '
2  [user1@local1 ~]\$
```

1.3 Introduction to Commands

1.3.1 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
1 $ 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

1.3.1.1 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
1 $ echo $PATH
2 /home/jarroddusr/bin:/usr/local/bin:/bin:/usr/bin:
```

1.3.1.2 Example: Command Lookup

```
Shell
1 $ grep pdf file.txt
```

执行步骤：

1. 检查 `grep` 是否是函数或内建命令。
2. 若不是，则 `bash` 会在 `$PATH` 路径列表中依次查找：
 - `/home/jarroddusr/bin`
 - `/usr/local/bin`
 - `/bin`
 - `/usr/bin`
3. 一旦找到匹配的可执行文件，就运行它。

使用 `type` 可查看命令位置：

```
Shell
1 $ type grep
2 grep is hashed (/usr/bin/grep)
```

1.3.1.3 Variable Substitution

`bash` 会在执行命令前进行变量替换：

```
Shell
1 $ myfile=file.txt
2 $ grep pdf $myfile
```

执行时会被解析为：

>_

Shell

```
1  grep pdf file.txt
```

🔗 Logic ▾

bash 在执行命令前会先替换变量（如 `$myfile`）、展开通配符、处理重定向等，再执行命令。

1.3.2 Getting Help with Commands

- 查看命令手册（man pages）

>_

Shell

```
1  $ man ls
```

- `man`（manual）命令显示系统自带帮助文档。
- 使用 `q` 退出手册界面。

- 使用帮助选项（--help）

>_

Shell

```
1  $ ls --help
```

- 直接在终端输出简要帮助与可用选项。

1.4 Command Line 中的高效操作

1.4.1 Tab 补全

在 Shell 中输入命令或文件名时，可以按下 `Tab` 键，Shell 会尝试补全我们正在输入的命令或文件名：

- 如果搜索路径中只有一个匹配的命令，那么 shell 会显示其值，光标会停在补全名称之后一个空格处
- 如果有多个命令与部分名称匹配，shell 会尽可能多地补全；此时连续按两次 `Tab` 键会显示一个选项列表，并重新显示部分命令行以供进一步编辑

1.4.2 键盘快捷键

Key Strokes	Descriptions
<code>Ctrl-a</code>	Beginning of line 行首
<code>Ctrl-e</code>	End of line 行尾
<code>Ctrl-k</code>	Delete line from cursor forward 从光标处删除到行尾
<code>Ctrl-w</code>	Delete word before cursor 删除光标前的单词
<code>Ctrl-y</code>	pastes in whatever was deleted previously with <code>Ctrl-k</code> or <code>Ctrl-w</code> 粘贴之前用 <code>Ctrl-k</code> 或 <code>Ctrl-w</code> 删除的内容
<code>ESC-F</code>	Forward one word 向前移动一个单词
<code>ESC-B</code>	Backwards one word 向后移动一个单词
<code>Ctrl-d</code>	EOF; exit 退出
<code>Ctrl-c</code>	Interrupt current command 中断当前命令
<code>Ctrl-z</code>	Suspend current command 挂起当前命令

Key Strokes	Descriptions
<code>Ctrl-l</code>	Clear screen 清除屏幕
<code>Ctrl-r</code>	Enables an interactive search history . 启用交互式搜索历史

1.4.3 Command History and Editing

1.4.3.1 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
1 $ history
2 1 echo $PS1
3 2 PS1=$
4 3 bash
5 4 export PS1=$
6 ...
7 11 ls -al manual.xml
```

- 历史记录行为由以下环境变量控制：

```
>_ Shell
1 $ echo $HISTFILE ## 历史记录文件
2 $ echo $HISTSIZE ## 记录的命令条数
```

1.4.3.2 Recalling Commands Quickly

- 通过命令编号或字符串回调：

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

示例：

```
>_ Shell
1 $ !-2 ## 执行倒数第二条命令
2 $ !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

```
1 $ !gi:p
```

这样可以查看命令内容，再按 **↑** 键编辑或执行。

1.4.3.3 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。

1.5 Accessing Remote Machines

1.5.1 SSH: Secure Shell

- **SSH** 提供加密的远程登录方式（Linux / Mac 常用）。
 - 基本用法：



Shell

```
1 $ ssh arwen.berkeley.edu
2 Password:
```

- 若远程用户名 (如 jarrod) 与本地机器上的用户名不同：



Shell

```
1 $ ssh jarrod@arwen.berkeley.edu
```

- 若需运行图形界面程序：



Shell

```
1 $ ssh -X jarrod@arwen.berkeley.edu
```

Logic ▾

-X 参数启用 **X11 forwarding**，
允许在远程服务器运行的 GUI 程序显示在本地屏幕上。

1.5.2 SCP: Secure Copy

- **从本地复制文件到远程主机**



Shell

```
1 $ scp file1.txt jarrod@arwen.berkeley.edu:.
```

将本地文件复制到远程用户主目录。

- **.** 表示保持原文件名。

- **从远程复制到本地**



Shell

```
1 $ scp jarrod@arwen.berkeley.edu:file2.txt .
```

- 从一个远程主机复制到另一个远程主机

```
>_ Shell
1 $ scp jarrod@arwen.berkeley.edu:file3.txt jmillman@scf-ug02.berkeley.edu:.
```

- 复制整个目录

```
>_ Shell
1 $ scp -r src jmillman@arwen.berkeley.edu:.
```

`-r` 表示递归复制整个目录结构。

⚠ Remark ▾

`scp` 在底层仍通过 SSH 传输数据，因此具有同样的加密与安全性。

1.5.3 Using `sudo` for Administrative Access

- 某些操作（安装软件、修改系统设置）需要 **root 权限**。
- 在 Ubuntu 或 WSL 中，可使用 `sudo` 临时以管理员身份执行命令。

示例：

```
>_ Shell
1 $ sudo apt-get upgrade    ## 升级全部软件
2 $ sudo apt-get install vim ## 安装 vim 编辑器
```

🔗 Logic ▾

`sudo` 代表 **superuser do**，
它让普通用户在授权情况下执行管理员操作。

Windows 用户可通过 **Ubuntu Subsystem (WSL)** 体验完整 Linux 环境并拥有 root 权限。

2 Bash Shell-File Management

🔗 Logic ▾

文件通常具有以下属性：

- Name 姓名
- Type 类型
- Location 位置
- Size 大小
- Protection (i.e., permissions on what can be done with the file)
保护（即对文件可执行的操作权限）
- Time, date, and user identification
时间、日期和用户标识

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
1 $ find . -name '*.txt' ## find files named *.txt
2 $ find . -mtime -2     ## find files modified less than 2 days ago
3 $ 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

⚠ Remark ▾

可以使用以下命令获取关于 `find` 命令的更多信息:

```
Shell
1 $ man find
2 $ find --help
```

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
1 $ ls -al
2 total 1489
3 drwxr-sr-x 7 paciorek scfstaff 31 Apr 21 16:39 ./
4 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
<code>*</code>	Match zero or more characters. 匹配零个或多个字符。
<code>?</code>	Match exactly one character. 匹配恰好一个字符。
<code>[characters]</code>	Match any single character from among <i>characters</i> listed between brackets. 匹配括号中列出的任意单个字符。
<code>[!characters]</code>	Match any single character other than <i>characters</i> listed between brackets. 匹配括号中未列出的任意单个字符。
<code>[a-z]</code>	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。
<code>[!a-z]</code>	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
<code>{frag1,frag2,...}</code>	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
1 $ ls *[0-9]
```

Example ▾

将 `filename` 复制为 `filename.old`：

```
>_ Shell
1 $ cp filename{,.old}
```

Example ▾

删除所有以 a 或 z 开头的文件：

```
>_ Shell
1 $ rm [az]*
```

Example ▾

列出所有具有各种后缀的 R 代码文件：

```
>_ Shell
1 $ 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
1 $ echo cp filename{,.old}
2 cp filename filename.old
```

Remark ▾

1. 可以使用反斜杠前缀（`\`）抑制通配符转义

```
>_ Shell
1 $ touch \*test    ## create a file called *test
2 $ ls \**
3 *test
```

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

```
>- Shell
1 $ 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

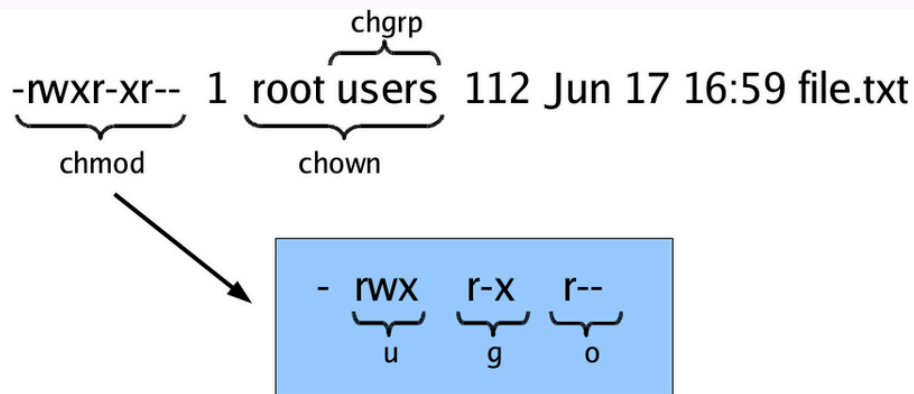
```
>- Shell
1 $ cd ~/stat243-fall-2020
2 $ ls -l
```

```
1 total 152
2 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 data
3 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 howtos
4 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 project
5 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 ps
6 -rw-rw-r-- 1 scflocal scflocal 11825 Dec 28 13:15 README.md
7 drwxrwxr-x 13 scflocal scflocal 4096 Dec 28 13:15 sections
8 -rw-rw-r-- 1 scflocal scflocal 37923 Dec 28 13:15 syllabus.lyx
9 -rw-rw-r-- 1 scflocal scflocal 77105 Dec 28 13:15 syllabus.pdf
10 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

```
1 $ ls -l
2 total 156
3 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 data
4 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 howtos
5 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 project
6 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:15 ps
7 -rw-rw-r-- 1 scflocal scflocal 11825 Dec 28 13:15 README.md
8 drwxrwxr-x 13 scflocal scflocal 4096 Dec 28 13:15 sections
9 -rw-rw-r-- 1 scflocal scflocal 37923 Dec 28 13:15 syllabus.lyx
10 -rw-rw-r-- 1 scflocal scflocal 77105 Dec 28 13:15 syllabus.pdf
11 drwxrwxr-x 2 scflocal scflocal 4096 Dec 28 13:37 units
```

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:

```
1 $ echo "first line" > tmp.txt # create a test text file that contains "first line"
2 $ ls -l tmp.txt
3 -rw-rw-r-- 1 scflocal scflocal 11 Dec 28 13:39 tmp.txt
```

移除读取权限:

```
1 $ chmod u-r tmp.txt # prevent owner from reading
2 $ chmod g-r tmp.txt # prevent users in the file's group from reading
3 $ chmod o-r tmp.txt # prevent others from reading
4 $ ls -l tmp.txt
5 --w--w---- 1 scflocal scflocal 11 Dec 28 13:39 tmp.txt
```

或者使用单行指令:

```
1 $ chmod ugo-r tmp.txt # prevent all three
2 $ ls -l tmp.txt
3 --w--w---- 1 scflocal scflocal 11 Dec 28 13:39 tmp.txt
```

此时我们无法读取文件:

```
1 $ cat tmp.txt
2 cat: tmp.txt: Permission denied
```

如果想同时移除读取和写入权限, 可以:

```
1 $ chmod ugo-rw tmp.txt # prevent all three
```

此时如果使用 **>>** 重定向运算符向文件添加一行, 会被拒绝:

```
1 $ echo "added line" >> tmp.txt
2 -bash: tmp.txt: Permission denied
```

接下来我们把权限恢复给所有者:

```
1 $ chmod u+rw tmp.txt
2 $ echo "added line" >> tmp.txt
3 $ cat tmp.txt
4 first line
5 added line
```

2.3.5 令其他用户访问文件

在使文件对其他用户可访问时, 还有许多重要的细节, 包括:

- [how to make files in a particular directory available to other users on the system](#), 如何使特定目录中的文件对系统上的其他用户可用
- [how to set up a directory for use by a UNIX group](#) 如何为 UNIX group 设置 directory, 即使用所谓的“sticky bit”, 以便将来在目录中创建的文件属于该组, 使组成员可以默认轻松访问它们

- [how to use access control lists](#)
如何使用访问控制列表来获得更多控制权

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

2.5 文档格式和转换

Pandoc 是一个广泛使用的文档转换工具

要将一个 Markdown (`report.md`) 格式的文件转换为 PDF (`report.pdf`), 可以这样做:



Shell

```
1 $ pandoc -o report.pdf report.md
```

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

```
1 ## 明确变量边界
2 echo "${variable}_suffix"
3
4 ## 默认值设置
5 echo "${VAR:-default_value}"
6
7 ## 字符串操作
8 echo "${string:0:5}"      ## 子字符串
9 echo "${string##prefix}"  ## 删除前缀
```

- **序列生成:**



Shell

```
1 ## 数字序列
2 echo {1..5}      ## 输出: 1 2 3 4 5
3 echo {5..1}      ## 输出: 5 4 3 2 1
4 echo {01..10}    ## 输出: 01 02 03 04 05 06 07 08 09 10
5
6 ## 字符序列
7 echo {a..e}      ## 输出: a b c d e
```

- 文件名拓展 (通配符):

```
>- Shell
1  ## 创建多个文件
2  touch file{1..3}.txt    ## 创建 file1.txt, file2.txt, file3.txt
3
4  ## 复制多个文件
5  cp file.{txt,bak}      ## 复制 file.txt 到 file.bak
```

- 命令分组:

```
>- Shell
1  ## 命令组合并重定向
2  { command1; command2; } > output.txt
3
4  ## 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
1  ## 在子 Shell 中执行命令
2  ( cd /tmp; ls )          ## 目录改变不会影响当前 Shell
3
4  ## 后台进程组
5  ( sleep 10; echo "Done" ) &
```

- 命令替换

```
>- Shell
1  ## 捕获命令输出
2  current_date=$(date)     ## 等同于 `date`
3
4  ## 嵌套命令
5  files=$(ls $(pwd))
```

- 数组定义

```
>- Shell
1  ## 创建数组
2  fruits=("apple" "banana" "cherry")
3
4  ## 访问数组元素
5  echo "${fruits[0]}"      ## 输出: apple
```

- 算术运算

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

- 进程替换

```
Shell
1  ## 比较两个命令的输出
2  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
1  $ command -options argument1 argument2 ...
```

Example ▾

```
Shell
1  $ 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

```
1 $ 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

```
1 $ 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

```
1 $ wget https://raw.githubusercontent.com/berkeley-scf/tutorial-using-bash/master/cpds.csv
2 $ tail -n 10 cpds.csv    ## 输出文件末尾10行
3 $ 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

```
1 $ grep ^2001 cpds.csv    ## 以 '2001' 开头的行
2 $ grep 0$ cpds.csv      ## 以 '0' 结尾的行
3 $ grep 19.0 cpds.csv    ## 匹配 19任意字符0
4 $ grep 19.*0 cpds.csv   ## 匹配 19 与 0 之间任意字符
5 $ 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

```
1 $ grep "\"Canada\"" cpds.csv  ## 匹配含引号的 "Canada"
2 $ 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

```
1 $ 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

```
1 $ grep "Canada" bigdata.csv > subset.csv
2 $ 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
<code>cmd > file</code>	将标准输出 (stdout) 写入文件 (覆盖原内容)
<code>cmd 1> file</code>	同上 (1 代表 stdout)
<code>cmd 2> file</code>	将标准错误 (stderr) 写入文件
<code>cmd > file 2>&1</code>	将 stdout 与 stderr 同时写入文件
<code>cmd < file</code>	从文件中读取输入 (stdin)
<code>cmd >> file</code>	将标准输出追加到文件末尾
<code>cmd 2>> file</code>	将标准错误追加到文件末尾
<code>cmd >> file 2>&1</code>	同时追加 stdout 与 stderr
<code>cmd1 cmd2</code>	将 cmd1 的标准输出输入到 cmd2
<code>cmd1 2>&1 cmd2</code>	将 cmd1 的标准输出和标准错误输入到 cmd2
<code>cmd1 tee file1 cmd2</code>	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
1 $ echo "hey there" | wc -w
2 2
```

3.2.3.2 示例 2：大小写转换

```
>- Shell
1 $ echo "user1" | tr 'a-z' 'A-Z'
2 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
1 $ cut -d',' -f2 cpds.csv | sort | uniq | wc
```

或保存结果到文件：

```
>- Shell
1 $ 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
1 $ cut -b29,37,45,53,61,69,77,85,93,101,109,117,125,133,141,149, \
2   157,165,173,181,189,197,205,213,221,229,237,245,253, \
3   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
1 $ cut -d',' -f2 cpds.csv | sort | uniq
2 $ cut -d',' -f2 cpds.csv | sort | uniq > countries.txt
```

更高效方式（使用 `tee`）：

```
>- Shell
1 $ 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
1 $(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
1 $ 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
1 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）	<code>ls</code> ， <code>rm</code> ， <code>cp</code> ， <code>echo</code>	<code>ls /home</code>

命令行为	数据来源	典型命令	举例
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

```
1 $ ls -t *. {R,r} | head -4 | xargs grep pdf
```

执行逻辑与上节等价于：

Shell

```
1 $ 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

```
1 $ ls -l tr ## 若当前目录下没有 tr，则会报错
2 $ type -p tr ## /usr/bin/tr
3 $ ls -l type -p tr ## 报错
4 $ 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

```
1 $ mv my_long_filename.{txt,csv}
```

```
2 $ ls my_long_filename*
3 my_long_filename.csv
```

相当于：

```
>_ Shell
1 mv my_long_filename.txt my_long_filename.csv
```

再例如：

```
>_ Shell
1 $ mv my_long_filename.csv{,-old}
2 $ ls my_long_filename*
3 my_long_filename.csv-old
```

等价于：

```
>_ Shell
1 mv my_long_filename.csv my_long_filename.csv-old
```

Logic ▾

花括号展开在命令执行前由 shell 完成，
生成的结果会被直接传递给命令。

3.4.1.2 Example 2: 使用序列展开

```
>_ Shell
1 $ echo {1..15}
2 $ echo c{c..e}
3 $ echo {d..a}
4 $ echo {1..5..2}
5 $ 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
1 $ 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
1 $ echo "My home directory is $HOME"
2 My home directory is /home/jarrold
3
4 $ echo 'My home directory is $HOME'
5 My home directory is $HOME
```

Logic ▾

使用双引号 " " 可让变量在字符串中被解释，
使用单引号 ' ' 则输出字面量。

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
1 $ ls $HOME/with\ space
2 file1.txt
```

或更简洁的写法：

```
>_ Shell
1 $ ls "$HOME/with space"
2 file1.txt
```

若使用硬引号（单引号）：

```
>_ Shell
1 $ ls '$HOME/with space'
2 ls: cannot access $HOME/with space: No such file or directory
```

因为 \$HOME 未被解析。

3.5.3 Escaping Double Quotes

若目录或文件名本身包含双引号 ("with"quote)：

```
>_ Shell
1 $ ls "$HOME/\"with\"quote"
```

通过 \" 转义内部双引号。

Remark ▾

- 尽量避免文件或目录名中出现空格与引号。
- 弯引号（curly quotes，如 “ ” 或 ‘ ’）在代码中无效，仅在字符串文本中可用。

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

3.6.1 Overview

在文本编辑器出现之前，Unix 用户依靠 行编辑器（line editor）来修改文件。早期的编辑器 `ed` 仅在需要时显示特定行，而非整个文件。许多现代工具（如 `grep`，`sed`，`awk`，`vim`）都源自 `ed`：

工具	起源与特性
<code>grep</code>	源自 <code>ed</code> 的命令 <code>g/<re>/p</code> ，用于全局匹配正则表达式
<code>sed</code>	“stream editor”——基于流的 <code>ed</code> 版本，可批量处理文本
<code>awk</code>	更通用的文本处理语言，语法上受 <code>ed</code> 启发
<code>vim</code>	<code>vi</code> 的改进版，同样继承了 <code>ed</code> 的操作逻辑

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`：

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

1. 查找包含某模式的行

Shell

```
1 $ grep is testfile.txt
2 This is the first line.
3 Followed by this line.
```

2. 查找不包含该模式的行

Shell

```
1 $ grep -v is testfile.txt
2 And then ...
```

`-v` 表示反选（打印不匹配的行）。

3. 只打印匹配内容

```
Shell
1 $ grep -o is testfile.txt
2 is
3 is
4 is
```

4. 彩色高亮匹配结果

```
Shell
1 $ 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` (**s**tream **e**ditor) 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
1 $ sed -n '1,9p' file.txt      ## 打印第 1-9 行
2 $ 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
1 $ sed -e '1,9d' file.txt
2 $ 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
1 $ sed 's/old_pattern/new_pattern/' file.txt > new_file.txt
2 $ sed 's/old_pattern/new_pattern/g' file.txt > new_file.txt
3 $ sed -i 's/old_pattern/new_pattern/g' file.txt
```

选项	说明
无 <code>g</code>	每行只替换第一个匹配项
<code>g</code>	全局替换行内所有匹配项
<code>-i</code>	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：

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

3.6.4.1 Example 1: 选择列

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

结果：

```
1 /root
```

3.6.4.5 Example 5: 列求和

```
Shell
1 $ 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
<code>grep</code>	搜索匹配行	支持正则表达式，快速定位文本
<code>sed</code>	编辑文本流	支持替换、删除、打印、批量修改
<code>awk</code>	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。

3.7 Aliases (Command Shortcuts) and `.bashrc`

3.7.1 What Are Aliases?

Aliases 是 Bash 提供的命令快捷方式，用于：

- 简化常用命令（缩写）；
- 为现有命令设置默认选项；
- 自定义命令行为。

基本语法：

```
>- Shell
1 $ alias name='command'
```

示例 1：创建退出命令的快捷方式

```
>- Shell
1 $ alias q=exit
```

现在输入 `q` 就等同于输入 `exit`。

示例 2：让 `ls` 命令始终显示文件类型标记

```
>- Shell
1 $ alias ls="ls -F"
```

此时：

- 目录会显示 `/`
- 可执行文件显示 `*`
- 链接显示 `@`

如果需要临时使用未被 `alias` 修改的原始命令，可使用反斜杠：

```
>- Shell
1 $ \ls
```

Logic ▾

Alias 的本质是 **命令替换 (text substitution)**，当你输入命令时，Bash 会先将别名替换成定义的完整命令。

3.7.2 Making Aliases Permanent

命令行直接设置的别名在关闭终端后会失效。

若要 **自动加载 alias 设置**，需将定义写入：

```
>- Shell
1 ~/.bashrc
```

该文件会在每次打开新的 Bash 会话时自动执行。

3.7.3 Example: A Typical `.bashrc` Configuration

>-

Shell

```
1  ## .bashrc
2
3  ## 载入全局设置
4  if [ -f /etc/bashrc ]; then
5      . /etc/bashrc
6  fi
7
8  ## 用户自定义函数
9  pushdp () {
10     pushd "$(python -c "import os.path as _, ${1}; \
11         print _.dirname(_.realpath(${1}).__file__[:-1])")"
12 }
13
14 ## 默认编辑器
15 export EDITOR=vim
16
17 ## Git 命令提示增强
18 source /usr/share/git-core/contrib/completion/git-prompt.sh
19 export PS1='[\u@\h \W$__git_ps1 " (%s)"]\$ '
20
21 ## 历史记录设置
22 export HISTCONTROL=ignoredups    ## 不记录重复命令
23 shopt -s histappend             ## 追加记录而非覆盖
24
25 ## R 环境设置
26 export R_LIBS=$HOME/usr/lib64/R/library
27 alias R="/usr/bin/R --quiet --no-save"
28
29 ## 路径设置
30 mybin=$HOME/usr/bin
31 export PATH=$mybin:$HOME/.local/bin:$HOME/usr/local/bin:$PATH
32 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/usr/local/lib
33
34 ## 常用别名
35 alias grep='grep --color=auto'
36 alias hgrep='history | grep'
37 alias l.='ls -d .* --color=auto'
38 alias ll='ls -l --color=auto'
39 alias ls='ls --color=auto'
40 alias more=less
41 alias vi=vim
```

3.7.4 Explanation of Key Sections

区块	功能说明
Global Settings	检查并载入系统级配置 <code>/etc/bashrc</code>
Custom Functions	定义用户自用函数（如 <code>pushdp</code> ）
Environment Variables	设置编辑器、历史记录行为、R 库路径等
PATH & LD_LIBRARY_PATH	添加自定义执行与库文件路径
Aliases	统一格式输出、颜色显示、常用命令缩写等

⚠ Remark ▾

修改 `.bashrc` 后需执行以下命令以立即生效：



Shell

```
1 $ source ~/.bashrc
```

或重新打开一个新的终端窗口。

🔗 Logic ▾

`.bashrc` 不仅可以存放 alias，还可定义函数、变量、命令提示符样式、路径设置等。
熟练运用 `.bashrc` 是打造个性化 Shell 环境的关键。

| 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

```
1 $ source ./file.sh
2 ## 或
3 $ . ./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。

| 4.1.1.1.2 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

```
1 #!/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

```
1 $ chmod +x file.sh
```

3. 执行脚本：

```
>_ Shell
1 $ ./file.sh
```

Logic ▾

建议始终在脚本首行加上 `#!/bin/bash`，
确保脚本在不同系统中使用正确的解释器。

4.2 Functions

4.2.1 Overview

Bash 函数是将多条命令封装在一起的可复用单元，
相比 `alias` 更强大，支持参数传递与逻辑控制。

定义语法：

```
>_ Shell
1 function name() {
2     commands
3 }
```

调用方式：

```
>_ Shell
1 $ 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
<code>\$1</code> , <code>\$2</code> , <code>\$3</code> , ...	依次表示第 1、2、3 个参数
<code>###</code>	参数个数
<code>@</code>	所有参数（以空格分隔）

4.2.2.1 Example: 自定义上传函数

```
>_ Shell
1 function putscf() {
2     scp $1 jarrod@arwen.berkeley.edu:$2
3 }
```

执行：

```
>_ Shell
1 $ putscf unit1.pdf teaching/243/.
```

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

🔗 Logic ▾

函数可放入 `.bashrc` 文件中，实现登录后自动加载。

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
1  if [ condition ]; then
2      commands
3  elif [ other_condition ]; then
4      commands
5  else
6      commands
7  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
1  ## niceR: 提交优先级较低的 R 作业
2  ## 用法: niceR inputRfile outputRfile
3  ## Author: Brian Caffo
4
5  function niceR() {
6      if [ $# != "2" ]; then
7          echo "usage: niceR inputRfile outputfile"
8      elif [ -e "$2" ]; then
9          echo "$2 exists, I won't overwrite"
10     elif [ ! -e "$1" ]; then
11         echo "inputRfile $1 does not exist"
12     else
13         echo "running R on $1"
14         nice -n 19 R --no-save < $1 &> $2
15     fi
16 }
```

- `-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`。

⚠ Remark ▾

将 `then` 放在单独一行可省略分号；
即：



Shell

```
1  if [ condition ]
2  then
3      ...
4  fi
```

4.3.3 Example: 字符串比较



Shell

```
1  var="some text"
2
3  if [ "${var}" == "some text" ]; then
4      echo "found equal"
5  fi
6
7  if [ "${var}" != "some text" ]; then
8      echo "found not equal"
9  fi
```

- 建议始终为变量加双引号，避免空格或通配符误解析。

⚠ Remark ▾

在 Bash 中，字符串比较时若省略引号，
空格与 `*` 等符号可能被误认为分隔符或通配符，导致逻辑错误。

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。
它是 shell 脚本中最常用的控制结构之一，常用于批量重命名、下载文件、或启动任务。

4.4.2 Example 1: 文件批量重命名



Shell

```
1  $ for FILE in $(ls *.txt); do
2  >     mv $FILE ${FILE/.txt/.R}
3  >     ## 将文件扩展名 .txt 替换为 .R
4  > 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

```
1  ## forloopDownload.sh
2  ## 使用 wget 下载多个文件
3  ## Author: Chris Paciorek
```

```

4  ## Date: July 28, 2011
5
6  url='ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/grid/years'
7  types="tmin tmax"
8
9  for ((yr=1950; yr<=2017; yr++))
10 do
11     for type in ${types}
12     do
13         wget ${url}/${yr}.${type}
14     done
15 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。

Logic ▾

若 `do` 独占一行，可省略分号。

即：

```

>_ Shell
1  for item in list
2  do
3      ...
4  done

```

4.4.4 Example 3: 批量启动任务

```

>_ Shell
1  ## forloopJobs.sh
2  ## 启动一系列 R 模拟任务
3  ## Author: Chris Paciorek
4  ## Date: July 28, 2011
5
6  n=100
7  for (( it=1; it<=100; it++ ))
8  do
9      echo "n=$n; it=$it; source('base.R')" > tmp-$n-$it.R  ## 创建定制 R 文件
10     R CMD BATCH --no-save tmp-$n-$it.R sim-n$n-it$it.Rout  ## 执行任务
11 done
12 ## 注意 base.R 不应在脚本中定义 n 或 it

```

Remark ▾

若任务间仅参数不同，更好的做法是通过 `commandArgs()` 或环境变量在 R 脚本中读取参数，而非生成多个临时文件。

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
1 $ IFS=:
2 $ types=tmin:tmax:pmin:pmax
3 $ for type in $types
4 > do
5 >     echo $type
6 > done
```

输出：

```
1 tmin
2 tmax
3 pmin
4 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-Managing Processes

5.1 What Is a Process

- **定义：**
进程（process）是正在被执行的程序实例。
- **进程属性**包括：
 - 生命周期（lifetime）
 - 进程 ID（PID）
 - 用户 ID（UID）
 - 组 ID（GID）
 - 父进程 ID（PPID）
 - 环境变量（environment）
 - 当前工作目录（current working directory）
- 每当执行一个命令或程序时，系统都会启动一个或多个进程来执行任务。

5.2 Monitoring

5.2.1 Monitoring Processes

5.2.1.1 Using `ps`

- **基本命令：**查看当前 shell 的子进程：

```
Shell
1 $ ps
2 PID TTY          TIME CMD
3 19370 pts/3        00:00:00 bash
4 22846 pts/3        00:00:00 ps
```

- 详细显示当前 **shell** 的进程层级：

	>_	Shell							
1	\$	ps -f							
2	UID	PID	PPID	C	STIME	TTY	TIME	CMD	
3	jarrod	19370	19368	0	10:51	pts/3	00:00:00	bash	
4	jarrod	22850	19370	0	14:57	pts/3	00:00:00	ps -f	

- 显示系统中所有进程：

	>_	Shell							
1	\$	ps -ef							
2	UID	PID	PPID	C	STIME	TTY	TIME	CMD	
3	root	1	0	0	Aug21	?	00:00:05	/usr/lib/systemd	
4	root	2	0	0	Aug21	?	00:00:00	[kthreadd]	
5	root	3	2	0	Aug21	?	00:00:07	[ksoftirqd/0]	
6	...								
7	jarrod	16219	16210	0	07:19	tty1	00:00:00	-bash	
8	jarrod	16361	16219	0	07:19	tty1	00:00:00	/bin/sh /bin/startx	

- 常用选项：

- u：显示每个进程的 CPU 和内存占用百分比。
- o：自定义输出字段，例如：

	>_	Shell							
1	\$	ps -o pid,ni,pcpu,pmem,user,comm							
2	PID	NI	%CPU	%MEM	USER	COMMAND			
3	18124	0	0.0	0.0	jarrod	bash			
4	22963	0	0.0	0.0	jarrod	ps			

- 查看进程树结构：

	>_	Shell							
1	\$	ps tree							

显示父进程与子进程之间的层级关系。

5.2.1.2 Using **top**

- 动态查看系统进程状态：

	>_	Shell							
1	\$	top							

输出示例：

1	top - 13:49:07 up 1:49, 3 users, load average: 0.10, 0.15, 0.18
2	Tasks: 160 total, 1 running, 158 sleeping, 1 stopped, 0 zombie
3	%Cpu(s): 2.5 us, 0.5 sy, 0.0 ni, 96.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
4	KiB Mem : 7893644 total, 5951552 free, 1085584 used, 856508 buff/cache
5	KiB Swap: 7897084 total, 7897084 free, 0 used. 6561548 avail Mem
6	
7	PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8	1607 jarrod 20 0 2333568 974888 212944 S 12.5 12.4 11:10.67 firefox
9	3366 jarrod 20 0 159828 4312 3624 R 6.2 0.1 0:00.01 top

- RES**：进程实际使用的物理内存（Resident memory）。
- %MEM**：占物理内存比例。

- **%CPU**: 占用 CPU 核心比例（多线程进程可 >100%）。
- **TIME+**: 进程累计运行时间。
- 按 **q** 退出。

Logic ▾

在 **top** 中可以动态管理进程：

- 按 **r** 可调整进程优先级（renice）；
- 按 **k** 可终止进程（kill）。

5.2.2 Monitoring Memory Use

- **内存使用异常的典型表现**：
 - 进程使用接近 **100% 内存**，但 **CPU 使用率较低**。
 - 表明程序正在使用磁盘作为虚拟内存（swap/paging），性能极差。
 - 此时程序大部分时间在磁盘读写，几乎无法完成任务。

⚠ Remark ▾

如果程序陷入“paging”状态，可能长时间无响应。
通常应减少内存占用或使用分布式处理框架。

- **查看整机可用内存**：

	Shell
1	\$ free -h
2	
3	total used free shared buff/cache available
4	Mem: 251G 998M 221G 2.6G 29G 247G
4	Swap: 7.6G 210M 7.4G

- 关注 **Mem** 行；
- 最关键的列是：
 - **total**：总内存
 - **used**：被进程实际占用的内存
 - **available**：可立即使用的内存

Logic ▾

free 列往往具有误导性，它不代表真实“可用”内存。
应始终参考 **available** 列来判断系统是否内存不足。

6 Bash Shell-Regular Expression

6.1 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。

6.1.1 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

6.1.2 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

6.1.3 特殊字符（元字符）：

```
>_ Shell
1 . ^ $ + ? ( ) [ ] { } | \
```

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 `\\`）。

6.2 字符集与字符类

6.2.1 运算符说明：

表达式	说明
<code>[abc]</code>	匹配任意一个列出的字符
<code>[a-z]</code>	匹配任意一个范围内的字符
<code>[^abc]</code>	匹配任意一个不在列出的字符
<code>[^a-z]</code>	匹配任意一个不在范围内的字符
<code>.</code>	匹配除换行符外的任意字符
<code>\</code>	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

6.2.2 示例：

```
>_ Shell
1 ## 匹配包含数字的行
2 grep -E [0-9] test.txt
3
4 ## 只输出匹配的数字
5 grep -E -o [0-9] test.txt
```

6.2.3 命名字符类（Named Character Classes）：

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

```
>_ Shell
```

```
1 ## 匹配包含标点符号的行
2 grep -E [[:punct:]] test.txt
3
4 ## 匹配数字、点或逗号
5 grep -E [[:digit:].,] test.txt
```

6.3 位置匹配

6.3.1 运算符说明：

表达式	说明
<code>^</code>	匹配行首
<code>\$</code>	匹配行尾

6.3.2 示例：

Shell

```
1 ## 匹配以数字开头的行
2 grep -E ^[0-9] test.txt
3
4 ## 匹配以数字结尾的行
5 grep -E [0-9]$ test.txt
```

6.4 重复、分组与引用

6.4.1 修饰符说明：

表达式	说明
<code>*</code>	匹配 0 次或多次
<code>?</code>	匹配 0 次或 1 次
<code>+</code>	匹配 1 次或多次
<code>{n,m}</code>	匹配 n 到 m 次
<code>\ </code>	匹配左边或右边的表达式

6.4.2 示例：

Shell

```
1 ## 匹配 http 或 ftp
2 grep -E -o "(http|ftp)" test.txt
3
4 ## 匹配电话号码格式
5 egrep '(1[-.])?[[:digit:]]{3}[-.][[:digit:]]{3}[-.][[:digit:]]{4}' file2.txt
```

6.5 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。

6.5.1 示例：

```
Shell
1  ## 贪婪匹配
2  grep -o "<.*>" file1.txt
3  ## 输出: <b> in place </b> of <b> one </b>
4
5  ## 非贪婪匹配 (Perl 语法)
6  grep -P -o "<.*?>" file1.txt
7  ## 输出: <b> </b> <b> </b>
```

6.5.2 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

7 Examples and Challenges

7.1 Bash Shell Examples

7.1.1 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
1  cd fall-2025/data
2  gzip -cd coop.txt.gz | less      ## 查看压缩文件内容
3  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
1  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 Python/R 加载数据。

7.1.2 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
1 tail -n 1 cpds.csv | grep -o ',' | wc -l
2 nfields=$(tail -n 1 cpds.csv | grep -o ',' | wc -l)
3 nfields=$((nfields+1))
4 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
1 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。

7.1.3 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
1 cd ../units
2 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
1 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

7.1.4 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
1 function mvlast() {

```

```

2     for ((i=1; i<=${1}; i++)); do
3         mv "/accounts/vis/paciorek/Downloads/${ls -rt \
4             /accounts/vis/paciorek/Downloads | tail -n 1}" ${2}
5     done
6 }

```

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
1  mvlst 3 ~/Desktop

```

7.1.5 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
1  grep --no-filename "^import " *.qmd | cut -d'##' -f1 | \
2      sed "s/as .///" | sed "s/import //" > tmp.txt
3  sed "s/,/\n/g" tmp.txt | sed "s/ //g" | sort | uniq | tee requirements.txt
4  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。

7.1.6 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
1  nJobs=30
2  for (( i=1; i<=${nJobs}; i++ )); do
3      python job.py > job-${i}.out &
4  done
5
6  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。

7.2 Bash Shell Challenges

7.2.1 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
1  ## 基本版 (以 Belgium 为例)
2  count=$(grep -o "Belgium" cpds.csv | wc -l)
3  echo "There are ${count} occurrences of the word 'Belgium' in this file."
```

7.2.2 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
1  cut -d',' -f4 RTADataSub.csv | sort | uniq > uniq_field4.txt
2  grep -E '^[0-9]' uniq_field4.txt
```

或只使用一行代码：

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

7.2.3 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
1  ## Belgium
2  grep "Belgium" cpds.csv | cut -d',' -f6 | sort -n | head -1
3  ## 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
1 countries=$(cut -d',' -f1 cpds.csv | tr -d '"' | sort | uniq)
2 for c in $countries; do
3     minval=$(grep "$c" cpds.csv | cut -d',' -f6 | sort -n | head -1)
4     echo "$c $minval"
5 done
```

7.2.4 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
1 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
1 function clean_missing() {
2     local missing=$1
3     local infile=$2
4     local removed=$(grep "${missing}" "${infile}" | wc -l)
5     echo "${removed} rows removed."
6     grep -v "${missing}" "${infile}"
7 }
8
9 ## 用法示例 (输出可用于管道):
10 clean_missing x RTADataSub.csv > clean.csv
```

7.2.5 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
1 grep -b "CA US" coop.txt | head -1
```

输出示例:

```
1 59:CA US
```

→ 州字段从第 60 字节开始。

自动提取位置并用于 `cut`

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

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

7.2.6 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 1,"America, United States of",45,96.1,"continental, coastal"
2 2,"France",33,807.1,"continental, coastal"
```

解决方案（简单替换）

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

输出示例：

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

7.3 Assignment 1 题目

7.3.1 Question 1 — 获取 Python 路径

```
>- Shell
1 mypython=$(which python)
```

7.3.2 Question 2 — 用户名 + 主机名

```
>- Shell
1 username_machinename=$USER@$(hostname)
```

7.3.3 Question 3 — 一行创建复杂目录结构

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

7.3.4 Question 4 — 统计文件行数

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

7.3.5 Question 5 — 打印前 3 行与第 3 行

```
>_ Shell
1 head -3 FILENAME
2 head -3 FILENAME | tail -1
```

7.3.6 Question 6 — 将第 3 行写入新文件

```
>_ Shell
1 head -3 FILENAME | tail -1 > NEWFILENAME
```

7.3.7 Question 7 — 追加第 5 行到同一文件

```
>_ Shell
1 head -5 FILENAME | tail -1 >> NEWFILENAME
```

7.3.8 Question 8 — 提取 Australia 数据

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

7.3.9 Question 9 — 查找不含逗号的行

```
>_ Shell
1 grep -v ',' FILENAME
2 ## 或:
3 grep -E '^[^,]*$' FILENAME
```

7.3.10 Question 10 — 批量创建文件

```
>_ Shell
1 for i in {1..N}; do
2     echo "blah" > "file${i}.txt"
3 done
```

7.4 Assignment 2 题目

7.4.1 Question 1 — 匹配任意大小写形式的 “dog”

	Shell
1	<code>grep -E "[Dd][Oo][Gg]" FILENAME</code> <code>## 正则匹配</code>
2	<code>grep -i "dog" FILENAME</code> <code>## 忽略大小写</code>

7.4.2 Question 2 — 匹配 “cat”、“caat”、“caaat” 等

	Shell
1	<code>grep -E "ca+t" FILENAME</code>
2	<code>grep -E "caa*t" FILENAME</code>
3	<code>grep -E "ca{1,}t" FILENAME</code>

7.4.3 Question 3 — 匹配 “cat”、“at”、“t”

	Shell
1	<code>grep -E "c?a?t" FILENAME</code> <code>## 允许可选的 c 和 a</code>
2	<code>grep -E "cat at t" FILENAME</code> <code>## 明确枚举匹配</code>

7.4.4 Question 4 — 匹配由任意空白分隔的两个单词

	Shell
1	<code>grep -E "[[:alnum:]]+[[:space:]]+[[:alnum:]]+" FILENAME</code>
2	<code>grep -E "[A-Za-z]+[[:space:]]+[A-Za-z]+" FILENAME</code> <code>## 仅限英文字母</code>