Python Programming for Data Science

Week 38, Friday

- Unix:
  - Input/output redirection
  - Pipelines
  - File permissions

Unix: Input/output redirection

Input/Output

Per default, a program takes input from `stdin` (the keyboard) and writes output to `stdout` (the screen).

In Unix, input and output can however be redirected in different ways:

- Input from a file
- Output to a file
- Output from one program used as input to another

Redirection from a file (input)

Specifying a file as input to a program:

```
command < filename
```

Example:

```
~$ less < .bash_history     # Sending .bash_history to the less program
```

Redirection to a file (output)

Specifying a file as output to a program:

```
command > filename
```

Rather than printing the output to screen (stdout), it will send it to *filename*.

Note: This will erase any existing data in *filename*!

Example:

```
~$ sort .bash_history > .bash_history_sorted    # Saving output to a file
```

Redirection to a file (appending)

Appending to an existing file:

```
command >> filename
```

If *filename* does not exist, it will be created for you

Example:

```
~$ echo "Hello!" >> diary.txt      # Add Hello! to the end of diary.txt
```

Redirection input and output

Input and output redirection can (of course) be combined

Example:

```
~$ sort < .bash_history > .bash_history_sorted
```

Redirection — Exercise

In our last exercise, we struggled with saving the results of our commands. Let's try again with our new tools:

1. Use the `sed` command to replace **`black`** with **`white`** in the british-english dictionary, but now save the result to a new file using output redirection.
2. Verify that the word **`black`** no longer occurs in this new dictionary file that you created.
3. Repeat step 1., but now append to the new file instead of overwriting it.
4. Use sort to sort the output file, and verify that all lines now occur twice.

Recall: Unix commands read from stdin

If you type

```
~$ sort
```

You will notice that the command does not return.

```
~$ sort
3      # entered from keyboard
2      # entered from keyboard
1      # entered from keyboard - followed by ctrl-d (means end-of-file)
1      # Output produced by the sort command
2      # Output produced by the sort command
3      # Output produced by the sort command
```

Recall: Unix commands read from stdin

If you type

```
~$ sort
```

You will notice that the command does not return.

This is because it is waiting for input from the keyboard

```
~$ sort
3      # entered from keyboard
2      # entered from keyboard
1      # entered from keyboard - followed by ctrl-d (means end-of-file)
1      # Output produced by the sort command
2      # Output produced by the sort command
3      # Output produced by the sort command
```

Recall: Unix commands read from stdin

If you type

```
~$ sort
```

You will notice that the command does not return.

This is because it is waiting for input from the keyboard

Try typing: `3 ↲ 2 ↲ 1`, and then press `ctrl-d`

```
~$ sort
3       # entered from keyboard
2       # entered from keyboard
1       # entered from keyboard - followed by ctrl-d (means end-of-file)
1       # Output produced by the sort command
2       # Output produced by the sort command
3       # Output produced by the sort command
```

Recall: Unix commands read from stdin

If you type

```
~$ sort
```

You will notice that the command does not return.

This is because it is waiting for input from the keyboard

Try typing: `3 ↵ 2 ↵ 1`, and then press `ctrl-d`

```
~$ sort
3      # entered from keyboard
2      # entered from keyboard
1      # entered from keyboard - followed by ctrl-d (means end-of-file)
1      # Output produced by the sort command
2      # Output produced by the sort command
3      # Output produced by the sort command
```

This is what makes redirection work

Wait a minute...

If we can send the output from a command to a file...

...and we can send the output from a file to a command

Wait a minute...

If we can send the output from a command to a file...

...and we can send the output from a file to a command

...can't we then just skip the file and send the output from one command directly to another command?

Wait a minute...

If we can send the output from a command to a file...

...and we can send the output from a file to a command

...can't we then just skip the file and send the output from one command directly to another command?

Yes, we can!

Pipelines

Use the | character to use the output of one command as input to another command

```
command | command
```

Pipelines

Use the | character to use the output of one command as input to another command

```
command | command
```

You can establish a whole chain of commands like this, where each command works as a filter on the output produced by the previous command.

Pipelines

Use the | character to use the output of one command as input to another command

```
command | command
```

You can establish a whole chain of commands like this, where each command works as a filter on the output produced by the previous command.

Example:

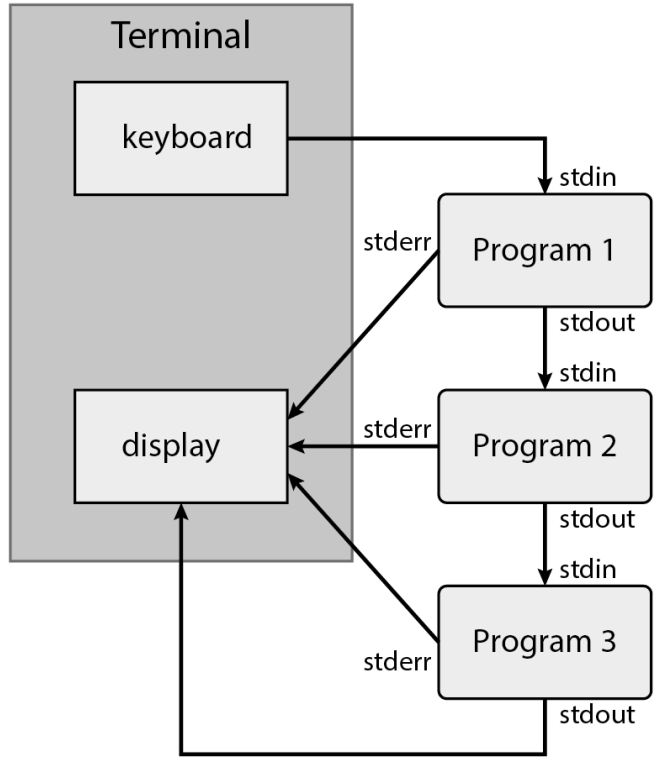
```
~$ sort /usr/share/dict/british-english | less  # view one page at a time
~$ grep apple /usr/share/dict/british-english | wc -l  # Count matches
~$ cat .bash_history | wc -l  # Same as wc -l < .bash_history
```

Standard-error: stderr

In addition to `stdin` and `stdout`, there is a third stream called `stderr`.

Like `stdout`, `stderr` is normally outputted to screen.

`stderr` is used for outputting errors.



Standard-error: stderr (2)

Since errors are outputted on a separate stream, you can see error messages (stderr) even if you redirect the output (stdout) from a command:

```
~$ grep -r hello / > results.txt
grep: /boot/System.map-3.19.0-25-generic: Permission denied
```

Standard-error: stderr (2)

Since errors are outputted on a separate stream, you can see error messages (stderr) even if you redirect the output (stdout) from a command:

```
~$ grep -r hello / > results.txt
grep: /boot/System.map-3.19.0-25-generic: Permission denied
```

It *is* possible to redirect `stderr` as well, but we won't go into this in this course.

More Unix commands

Unix commands: tee

If you want to redirect output, but also want to save a copy to a file, use `tee`

```
command | tee filename | command
```

Example:

```
~$ grep apple /usr/share/dict/british-english | tee dict.txt | wc -l
```

Unix commands: cut

Extract columns from a file

```
cut -f column-number -d delimiter filename
```

Example

```
~$ cat /usr/share/matplotlib/sample_data/demodata.csv
clientid,date,weekdays,gains,prices,up
0,2008-04-30,Wed,-0.52458192906686452,7791404.0091921333,False
1,2008-05-01,Thu,0.07619153620173826 9,3167180.7366340165,True
...
```

```
~$ cut -d "," -f 2-3 /usr/share/matplotlib/sample_data/demodata.csv
date,weekdays
2008-04-30,Wed
2008-05-01,Thu
...
```

Unix commands: number lines (nl)

The `nl` command adds line numbers

```
nl filename
```

Example:

```
~$ nl /usr/share/dict/british-english
     1  A
     2  A's
     3  AA's
```

Unix commands: substitute characters

The `tr` command allows you to replace one set of characters with another

```
tr characters-to-replace characters-to-replace-with
```

Using `-s`, you can also squeeze multiple repeated occurrences of a character into a single occurrence.

Using `-d`, you can delete a character alltogether.

```
~$ echo "hello" | tr "l" "p"
heppo
~$ echo "hello" | tr -s "l"        # Squeezing
helo
~$ echo "hello" | tr -d "l"        # Deleting
heo
```

Unix commands: Remove duplicate lines

The `uniq` command filters out repeated lines

```
uniq filename
```

Use `-d` to show only the duplicate lines

Use `-c` to count how many times each line is repeated

Example:

```
~$ uniq -d /usr/share/dict/british-english  # test for duplicates
```

Unix commands: Difference between files

Compare files line by line, and check for differences

```
diff filename1 filename2
```

Example:

```
~$ diff /usr/share/dict/british-english /usr/share/dict/american-english
...
18494,18495c18495,18496
< aluminium
< aluminium's
---
> aluminum
> aluminum's
...
```

Unix commands: Difference between files

Compare files line by line, and check for differences

```
diff filename1 filename2
```

Example:

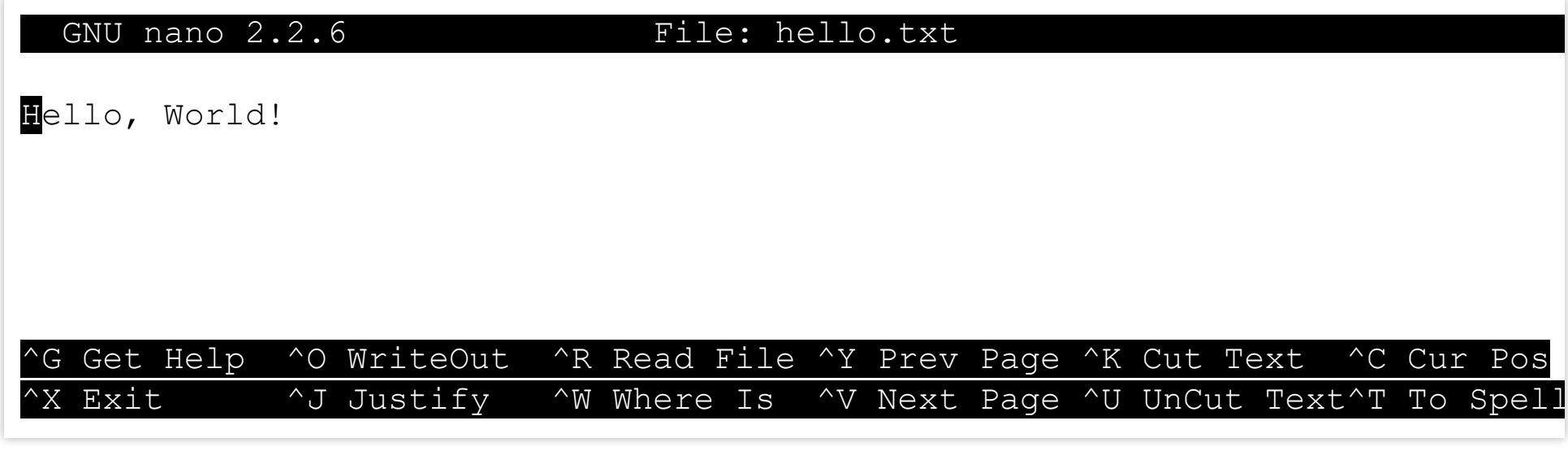
```
~$ diff /usr/share/dict/british-english /usr/share/dict/american-english
...
18494,18495c18495,18496
< aluminium
< aluminium's
---
> aluminum
> aluminum's
...
```

The numbers provide information about where in the files the changes were found (see the man page for details)

Editor in the terminal: nano

For quick editing tasks in the terminal:

```
~$ nano hello.txt
```



Unix pipelines - Exercise

1. Use `head` and `tail` to output lines 1000 to 1500 of the british-english dictionary.
2. Extend the command from 1. to show only the lines in which the letter "a" occurs.
3. Extend the command from 2. to count the total number of characters in these lines.
4. Imagine we did not have the `diff` command. Try to use the `sort` and `uniq` commands to find the differences between the English and American dictionaries.
5. Try to explain what these two commands do:

```
~$ wget http://www.gutenberg.org/cache/epub/10/pg10.txt
~$ cat pg10.txt | tr 'A-Z' 'a-z' | tr -cs 'a-z' '\n' | sort | uniq -c | sort -n
```

Permissions

Permissions

Each file and directory on a Unix system has an associated set of permissions

Permissions to the *User (u)*
The *user* is the owner of the file.

Permissions to the *Group (g)*
Each file is associated with a group of users.

Permissions to *Other (o)*
Everybody else.

Permissions to do what?

The system differentiates between permission to

- Read
- Write
- Execute

Permissions to do what?

The system differentiates between permission to

- Read
- Write
- Execute

For files, *execute* means to run a file as a program.

Permissions to do what?

The system differentiates between permission to

- Read
- Write
- Execute

For files, *execute* means to run a file as a program.

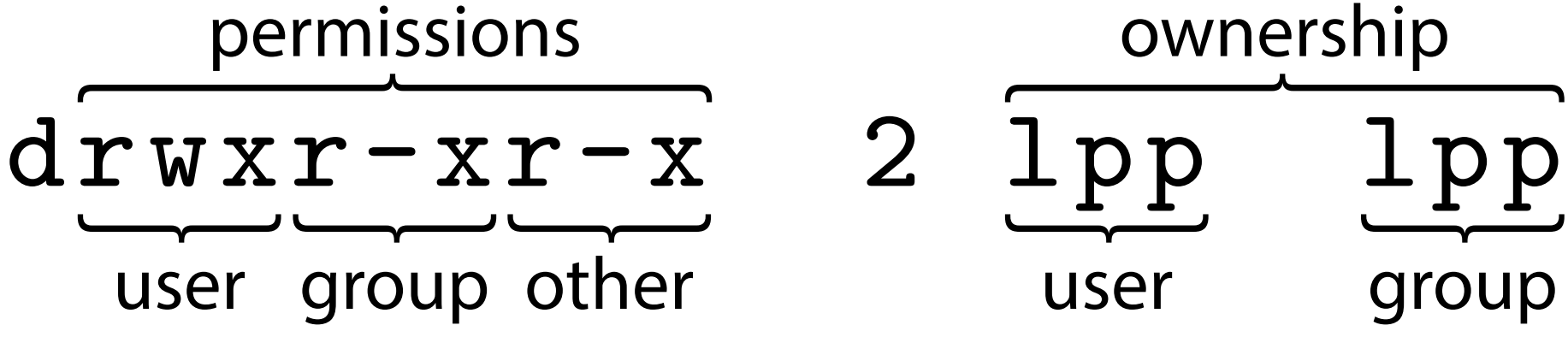
For directories, *execute* means the permission to enter the directory (e.g. `cd` to it or use `ls` on it).

How to see the permissions for a file

Using the long-format options to `ls`:

```
ls -lF
```

the permissions are listed to the left.



For this directory, the user can do everything, while anyone else can read and enter, but is not allowed to write.

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

where *permissions* consists of three parts:

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

where *permissions* consists of three parts:

1. Whose permission are we changing: user(u), group(g), other(o), all(a)

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

where *permissions* consists of three parts:

1. Whose permission are we changing: user(u), group(g), other(o), all(a)
2. Action: add(+), remove(-), set to (=)

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

where *permissions* consists of three parts:

1. Whose permission are we changing: user(u), group(g), other(o), all(a)
2. Action: add(+), remove(-), set to (=)
3. Type of permission: read(r), write(w), execute(x)

Changing permissions

You can change permissions of a file using `chmod`:

```
chmod permissions file-or-directory
```

where *permissions* consists of three parts:

1. Whose permission are we changing: user(u), group(g), other(o), all(a)
2. Action: add(+), remove(-), set to (=)
3. Type of permission: read(r), write(w), execute(x)

Example:

```
~$ chmod u-w .bashrc    # Making .bashrc read-only
```

Changing group and ownership

You can change the group of a file using `chgrp`:

```
chgrp new-group file-or-directory
```

Changing group and ownership

You can change the group of a file using `chgrp`:

```
chgrp new-group file-or-directory
```

...and ownership can be changed using `chown`:

```
chown new-user file-or-directory
```

Changing group and ownership

You can change the group of a file using `chgrp`:

```
chgrp new-group file-or-directory
```

...and ownership can be changed using `chown`:

```
chown new-user file-or-directory
```

Both commands take a `-R` option to allow you to apply them recursively to a directory

Changing group and ownership

You can change the group of a file using `chgrp`:

```
chgrp new-group file-or-directory
```

...and ownership can be changed using `chown`:

```
chown new-user file-or-directory
```

Both commands take a `-R` option to allow you to apply them recursively to a directory

```
~$ touch testfile.txt                    # Create empty file
~$ chown root testfile.txt               # "Donate" testfile.txt to superuse
chown: changing ownership of ‘testfile.txt’: Operation not permitted
```

Changing group and ownership

You can change the group of a file using `chgrp`:

```
chgrp new-group file-or-directory
```

...and ownership can be changed using `chown`:

```
chown new-user file-or-directory
```

Both commands take a `-R` option to allow you to apply them recursively to a directory

```
~$ touch testfile.txt                          # Create empty file
~$ chown root testfile.txt                     # "Donate" testfile.txt to superuse
chown: changing ownership of ‘testfile.txt’: Operation not permitted
```

...hmm...it seems we need some more privileges

The sudo command

`root` is a super-user that has full privileges to modify any file.

The sudo command

`root` is a super-user that has full privileges to modify any file. But even when you are the system administrator of a system, you don't want do be logged in as root all the time (to avoid accidental deletions etc.).

The sudo command

`root` is a super-user that has full privileges to modify any file. But even when you are the system administrator of a system, you don't want do be logged in as root all the time (to avoid accidental deletions etc.).
The `sudo` command allows you to assume superuser privileges just for a single command:

```
sudo command
```

The sudo command

`root` is a super-user that has full privileges to modify any file. But even when you are the system administrator of a system, you don't want do be logged in as root all the time (to avoid accidental deletions etc.).
The `sudo` command allows you to assume superuser privileges just for a single command:

```
sudo command
```

Example:

```
~$ sudo chown root testfile.txt       # "Donate" testfile.txt to superuser
[sudo] password for wb:               # Asked to enter password
~$                                    # Operation completed
```

The sudo command

`root` is a super-user that has full privileges to modify any file. But even when you are the system administrator of a system, you don't want do be logged in as root all the time (to avoid accidental deletions etc.).
The `sudo` command allows you to assume superuser privileges just for a single command:

```
sudo command
```

Example:

```
~$ sudo chown root testfile.txt      # "Donate" testfile.txt to superuser
[sudo] password for wb:              # Asked to enter password
~$                                   # Operation completed
```

Note that this only works on systems where you are an administrator (not on our server)

Permissions — Exercise

Using permissions, we can make a script *executable*:

1. Use `nano` to create a file called `hello.py`, containing:

   ```
   #!/usr/bin/env python
   print("Hello, World!")
   ```

   The first line is called a she-bang that tells Unix which interpreter should be used for this file.
2. Verify that the script works by running it as usual:
   `$ python hello.py`
3. Now try to run the file directly as an executable:
   `$ ./hello.py`. What happens?
4. Try to add executable permissions to the file, and then try again
5. What happens if you remove the `./` in front of hello.py. Why?

Unix process control

Processes

When a program is running, it is called a *process* or a *job*.

Processes

When a program is running, it is called a *process* or a *job*.

Unix supports multitasking, and can therefore run many programs at once.

Processes

When a program is running, it is called a *process* or a *job*.

Unix supports multitasking, and can therefore run many programs at once.

To get an overview of the processes running right now, use the `ps` or `top` commands

The ps command

When ps is started without any options, it displays only the processes started from the current shell.

```
ps       # Shown only jobs started from current shell
```

To display all processes and who owns them:

```
ps aux
```

To display only your own:

```
ps ux
```

The `top` command

For an interactive overview, use the command `top`.

```
Tasks: 139 total,   1 running, 138 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0
KiB Mem:    507988 total,   402052 used,   105936 free,    59212 buffers
KiB Swap:   522236 total,        0 used,   522236 free,   203676 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 1416 lpp       20   0  9896 1440  976 S   0.3  0.3   0:00.04 VBoxClient
 1812 lpp       20   0  103m  15m  11m S   0.3  3.2   0:00.18 gnome-termin
 1867 lpp       20   0  5204 1320  972 R   0.3  0.3   0:00.01 top
    1 root      20   0  3908 2240 1380 S   0.0  0.4   0:01.07 init
```

You can quit using the `q` key.

Running programs in the background

We can run programs in the background using the & character.

```
~$ sleep 10 &
[1] 2162
~$ ps
  PID TTY          TIME CMD
 1818 pts/0    00:00:00 bash
 2162 pts/0    00:00:00 sleep
 2163 pts/0    00:00:00 ps
```

Running programs in the background

We can run programs in the background using the & character.

```
~$ sleep 10 &
[1] 2162
~$ ps
  PID TTY          TIME CMD
 1818 pts/0    00:00:00 bash
 2162 pts/0    00:00:00 sleep
 2163 pts/0    00:00:00 ps
```

Note that the ID of the process is written to screen

Backgrounding running programs

A locked terminal:

What can we do?

```
~$ sleep 300
```

`ctrl-c`: kill process

`ctrl-z`: suspend process

If we suspend it, we can thereafter either:

- send it to the background using `bg`.
- send it to the foreground using `fg`.

```
~$ sleep 300
^Z
[1]+  Stopped                 sleep 300
~$ bg
[1]+ sleep 300 &
~$
```

Killing processes: `kill`

To stop a job running in the background, use the `kill` command.

```
kill PID
```

You can find the process ID (PID) using `ps` or `top`.

If a program won't die, try using

```
kill -9 PID
```

Killing processes: `killall`

Kill processes by name

```
~$ sleep 300 &
[1] 2216
~$ killall sleep
[1]+  Terminated              sleep 300
```

Process control — Exercise

Let's create a Python program that does nothing for 5 minute:

```
import time       # We'll get back to what this means next week
time.sleep(300)   # Sleeps for 300 seconds
print("Done waiting!")
```

1. Use `nano` to type in the above script on the server - in a file called `wait.py`
2. Start the program. This will lock your terminal.
3. Now suspend the job, and run it in the background instead.
4. Run the `wait.py` script again, but now start it so that it runs in the background immediately.
5. Run `ps` to get an overview over the processes.
6. Now kill one of the wait processes using the `kill` command. You can let the other one run (it will print something to screen when it's done).