Research Computing

Xinyu Zhong
Queens' College

January 12, 2024

Contents

Abstract

Abstract of this course

1 Terminal Command lines

1.1 Linux

Linux is a free open source Unix-like operating system kernel. A kernel is the part of the operating system that is resposible for the lowest level task such as memory management, process management, task management and disk management.

1.2 BASH

Bash stands for Bourne Again Shell, it is a type of shell, like ksh csh tcsh and zsh. It is the default shell for most Linux distributions.
It is accessable through the terminal.

1.2.1 Navigation

- pwd: tells you where you are
- ls: lists the files in the current directory -a: all files including hidden files -ltra: long list, by time, reverse order, all files
- cd: change directory
- mkdir: make directory
- rmdir: remove a directory
- touch: create a file
- rm: remove a file
- mv: move a file
- cp: copy a file

  ```
          $ cp file1.txt file2.txt
  ```

  the above command copies file1.txt to file2.txt
- rename: rename a file

  ```
          $ rename file1.txt file2.txt
  ```

  the above command renames file1.txt to file2.txt

1.2.2 Permission

The permissions for each file is described by the first 10-characters in long format (**drwxrwxrwx** would mean directory read write execute at **user**, then **group**, then **other level**. If a letter is replaced by **-** then the corresponding permission is denied for that set of users)

- d: directory

- r: read
- w: write
- x: execute

Permissions can be changed using the following commands:

- chmod: ("change mode") change the permission of a file chmod [options] mode file: where mode is a 3-digit octal number
  - 1st digit: owner
  - 2nd digit: group
  - 3rd digit: other

  Example:

  ```
              chmod 755 file.txt
  ```

  - 7 = 4 + 2 + 1 = rwx
  - 5 = 4 + 1 = rx
  - 5 = 4 + 1 = rx
- chown: ("change owner") change the ownership of a file
- umask: set the default permission of a file e.g. umask 644

The default permission is 666 for files and 777 for directories.

1.2.3 Search

There are two commands for searching: find and grep.

- find: find files
- grep: search for a pattern in a file

Find is a very powerful command, it can be used to find files by name, size, type, date modified, etc.

```
    $ find . -name "*.txt"
```

This command finds all the files with the extension .txt in the current directory and all subdirectories. Here, the double quotes are important, otherwise the shell will expand the wildcard before passing it to find.
Grep is used to search for a pattern in a file.
The difference between double quotes and single quotes is that double quotes allow the shell to expand the wildcard, while single quotes do not.
Backtickts indicates that it must be expanded before the command is run. It is the same as $(command).

```
    $ grep hello greeting.txt     ->  hello
    $ grep hello *.txt            ->  hello
    $ grep hello "*.txt"          ->  grep: *.txt: No such file or
       directory
    $ grep hello '*.txt'          ->  grep: *.txt: No such file or
       directory
```

1.2.4 Wildcards

wildcards are used to match patterns.
For *find* command:

- *: matches any number of characters
- ?: matches any single character
- []: matches any character in the brackets
- [!]: matches any character not in the brackets

1.3 Read

We use **cat**, **more**, **less** to read files.

- cat: prints the whole file
- more: prints the file one page at a time
- less: prints the file one page at a time, but allows you to scroll up and down

1.3.1 Help

man is the manual for the command

1.3.2 Convenience

- up arrow: previous command
- tab: autocomplete

tab, up error, down arrow to go through commands

1.4 File manipulation

file, tar, zip, unzip, diff, cut

- **sort** -n (n: numeric)
- **sort** -r (r: reverse)
- **sort** -u (u: unique)
- **unique**
- **tar** -czf (c:create, z:zip f:file name specified)
- **tar** -xf (x: extract)

```
$ tar -czf file.tar.gz "list of files"
$ tar -xf file.tar.gz
$ zip file.zip "list of files"
$ unzip file.zip
```

- **tar** vs **zip**, tar creates smaller zipped files, can only zip and unzip all the files together
- **diff** tells you what has changed
- **diff** Git uses a bunch of diff files to keep track of the file changes for version control

```
$ diff file1.txt file2.txt
```

It will tell you what has changed between the two files:

```
"file1 line start","file1 line end"{a,c,d}"file2 line start","file2 line end"
> lines in
> file1
---
< lines in
< file2
```

here a: add, c: change, d: delete In specific example:

```
$ diff diff_file1.txt diff_file2.txt
3c3
< fish
---
> bird
7d6
< scarf
10a10
> three
```

It tells you that line 3 has changed from fish to bird, line 7 has been deleted, line 10 has been added

- **cut** -f (f: field)

```
$ cut -f 1,3,5 file.txt
```

It will extract the content from column 1, 3, 5

- **cut** -d (d: delimiter)

1.5 Redirection

Redirection allows us to pass the output of one command as the input to another command.
We have 3 standard streams:

1. STDIN(0)
2. STDOUT(1)
3. STDERR(2)

1.5.1 STDOUT

STDOUT is the standard output stream. It is used to print output to the terminal.

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

It redirects the output of ls -l to output.txt

```
$ grep 'func' code.py >> output
```

It appends the output of grep 'func' code.py to output.txt To capture the error, use 2>:

```
$ ls test.txt 2> errors.txt
```

To redirect both STDOUT and STDERR, use &> or combines outputs:

```
    $ ls test.txt &> output.txt
    $ ls *.txt >output.txt 2>&1
```

2 Advanced Linux Terminal

2.1 Managing Processes

To run a python script, Use

```
    $ python3 script.py
```

To run a python script in the background, Use

```
    $ python3 script.py &
```

List all the processes running in the background, Use

```
    $ ps
```

List all processes on the system with all info

```
    $ ps -elf
```

When you start a process in the background, it will be assigned a number, called the PID (process ID). You can use this number to kill the process.
To kill a process, Use

```
    $ kill PID
```

Use -SIGTERM to kill a process gracefully, -SIGKILL to kill a process immediately and -SIGSTOP to pause a process.

```
    $ kill -SIGTERM PID
    $ kill -SIGKILL PID
    $ kill -SIGSTOP PID
```

2.2 Remote Computing

SSH SSH stands for Secure Shell, it is a cryptographic network protocol for operating network services securely over an unsecured network.
To connect to a remote server, Use

```
    $ ssh username@server
```

To set up a SSH key, Use

```
    $ ssh-keygen
```

To copy the SSH key to the remote server, Use

```
    $ ssh-copy-id username@server
```

SSH-agents are used to store the private key so that you don't have to type the password every time you connect to the server.
To start the SSH-agent, Use

```
$ eval $(ssh-agent)
```

To add the private key to the SSH-agent, Use

```
$ ssh-add
```

2.3 Scripting

All commands can be executed in a script file. It is often given the extension .sh.
First, need to start the file with the following line:

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

This makes sure that the script is executed by bash.
Once you have this line you can write commands as if you were typing them in the terminal.e.g.

```
#!/bin/bash
cd ~code/output
rm -f *.log *.out
```

Note that you need to make the script executable before you can run it.
To make the script executable, Use

```
$ chmod 744 script.sh
```

Here 744 means that the owner has read, write and execute permission, while the group and others have only read and execute permission.

2.3.1 Variables

Variables are used to store values.
To assign a value to a variable, Use

```
$ var1=1
$ echo $var1
```

2.3.2 strings

```
echo "**** Strings ****"
test_string1='abcdefghijklm'
test_string2='nopqrstuvwxyz'
echo ${#test_string1}  #string length
echo "** Substrings"
echo ${test_string1:7} #substring from position 7
echo ${test_string1:7:4} #substring from position 7, for 4 characters
echo "** Substring Removal"
echo ${test_string1#'abc'} #shortest substring removed from front
echo ${test_string1##'abc'} #longest substring removed from front
echo ${test_string1%'klm'} #shortest substring removed from back
echo ${test_string1%%'klm'} #longest substring removed from back
echo "** Replacement"
echo ${test_string1/efg/567} #replacement first match
echo ${test_string1//efg/567} #replacement all matches
echo "** note: to make match at front or back add # or %"
echo "** if no replacement is supplied does deletion"
echo ${test_string1/efg} #deletion
echo "** Joining"
echo $test_string1$test_string2 #joining strings, += also works
```

2.3.3 Loops

```
for i in {1..10}; # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
do
    echo "List form:    The iteration number is $i"
done

for (( i = 0; i < 10; i++ )) #C style loop
do
    echo "C style form: The iteration number is $i"
done

i=0
while [ $i -lt 5 ] #Executes until false
do
    echo "while: i is currently $i"
    i=$[$i+1] #Not the lack of spaces around the brackets. This makes it a not a test expression
done

i=5
until [[ $i -eq 10 ]]; #Executes until true
do
    echo "until: i is currently $i"
    i=$((i+1))
done
```

2.3.4 Conditionals

```
if [ "$num" -eq 1 ]; then
    echo "the number is 1"
elif [ "$num" -gt 2 ]; then
    echo "the number is greater than 2"
else
    echo "The number was not 1 and is not more than 2."
fi
```

2.3.5 Functions

```
    function_greet () {
        echo "This script is called $0"
        echo "Hello $1"
    }

    function_greet "James"

    $ ./script.sh
    This script is called ./script.sh
    Hello James
```

Functions do not return anything, they just print to the screen.
You can use global variables to store the return value of a function.

2.3.6 Example of a script

```
#!/bin/bash
##################################################
# This script creates a blank Python repository: #
##################################################

# Set the repository name
# check if a name has been specified
# if not, exit with error
repo_name=$1
if [ -z $repo_name ]; then # The -z flag is used to check if the length of the string variable $repo_name is zero (i.e., if the string is empty). Also see: https://unix.stackexchange.com/questions/306111/what-is-the-difference-between-the-bash-operators-vs-vs-vs
    echo "Name is empty"
    echo "Usage is:"
    echo "./create_repo.sh <repo_name>"
    exit 1
fi
echo "Creating blank Python repository '$repo_name'..."
echo "**********************************************"

# Create the repository directory
echo "Creating directory structure..."
echo "**********************************************"
mkdir $repo_name
cd $repo_name

mkdir src
mkdir test
mkdir docs
mkdir output

# In case we would like to commit the empty folders to the repository (it's optional here)
touch src/.gitkeep
touch test/.gitkeep

# The .gitignore file is used to specify intentionally untracked files that Git should ignore. When you add entries to .gitignore, you are telling Git to ignore specific files or directories, preventing them from being tracked or included in the version control system.
echo "Creating gitignore file..."
echo "**********************************************"
# create the gitignore file
touch .gitignore
echo "# Cache and Testing">>.gitignore
echo "__pycache__/">>.gitignore # The __pycache__ directory is automatically generated by Python to store compiled bytecode files (.pyc) when a Python script or module is imported or executed for the first time
echo "*.py[cod]">>.gitignore # The pattern *.py[cod] in a .gitignore file is a wildcard pattern used to match and ignore compiled Python files.
echo "*.pytest">>.gitignore
echo "">>.gitignore
echo "# Folders">>.gitignore
echo "output/">>.gitignore
echo "docs/">>.gitignore
echo "!*Doxyfile">>.gitignore

# The pre-commit configuration file specifies which hooks to run and how they should be configured. It is usually named .pre-commit-config.yaml and is placed in the root directory of the Git repository.
echo "Creating pre-commit config file..."
echo "**********************************************"
# Create the pre-commit config file
# standard pre-commit hooks + Black + Black Jupyter + flake8
touch .pre-commit-config.yaml
echo "repos:">>.pre-commit-config.yaml
echo "  - repo: https://github.com/pre-commit/pre-commit-hooks">>.pre-commit-config.yaml
echo "    rev: v4.0.1">>.pre-commit-config.yaml
echo "    hooks:">>.pre-commit-config.yaml
echo "      - id: check-yaml">>.pre-commit-config.yaml
echo "      - id: end-of-file-fixer">>.pre-commit-config.yaml
echo "      - id: trailing-whitespace">>.pre-commit-config.yaml
echo "      - id: mixed-line-ending">>.pre-commit-config.yaml
echo "      - id: debug-statements">>.pre-commit-config.yaml
echo "  - repo: https://github.com/psf/black">>.pre-commit-config.yaml
echo "    rev: 23.11.0">>.pre-commit-config.yaml
echo "    hooks:">>.pre-commit-config.yaml
echo "      - id: black">>.pre-commit-config.yaml
echo "        language_version: python3.11.6">>.pre-commit-config.yaml
echo "      - id: black-jupyter">>.pre-commit-config.yaml
echo "        language_version: python3.11.6">>.pre-commit-config.yaml
echo "  - repo: https://github.com/pycqa/flake8">>.pre-commit-config.yaml
echo "    rev: 6.0.0">>.pre-commit-config.yaml
echo "    hooks:" >>.pre-commit-config.yaml
echo "      - id: flake8">>.pre-commit-config.yaml


# Create the conda enviroment and install the basics
if hash conda 2>/dev/null; then
    # Check if conda is installed
    read -p "Create new Conda Environment with standard packages? (y/n)" -n 1 -r
    echo "**********************************************"

    # Add the actions you want to perform when conda is installed
    if [[ $REPLY =~ ^[Yy]$ ]]; then
        # Create a new Conda environment with standard packages
        # Add your conda create command here
        echo "Creating a new Conda environment..."
        conda create -n $repo_name \
            pre-commit \
            pytest \
            black \
            flake8 \
            configparser \
            numpy \
            pandas \
            matplotlib \
            scipy
        conda env export -n $repo_name -f environment.yml --no-builds --from-history # export the environment that will work on any OS
    elif [[ $REPLY =~ ^[Nn]$ ]]; then
        read -p "Use existing Conda Environment? (y/n)" -n 1 -r
        echo "**********************************************"
        if [[ $REPLY =~ ^[Yy]$ ]]; then
            read -p "Specify environment to use:" conda_env
            conda env export -n $conda_env -f environment.yml --no-builds --from-history
        elif [[ $REPLY =~ ^[Nn]$ ]]; then
            echo "No conda environment specified, exiting..."
            exit 1
        else
            echo "Invalid input, exiting..."
            exit 1
        fi
    else
        # Add your actions when the user chooses not to create a new environment
        echo "Exiting without creating a new Conda environment."
    fi
```

```bash
else
    echo "Conda not installed, exiting..."
    exit 1
fi

echo "**********************************************"
echo "Installing pre-commit hooks..."
echo "**********************************************"
# set up pre-commit (this requires pre-commit to be installed in root environment)
# it is a massive pain to activate the conda environment in a bash script

if hash pre-commit 2>/dev/null; then # check if pre-commit is installed
# Note that 2>/dev/null is to make the check for the existence of conda more silent, and if it's not found, the script provides its own error message and exits with an error code.
    pre-commit install
else
    echo "Pre-commit not installed, skipping..."
fi

'''
 Notes about Doxygen:
-- JAVADOC_AUTOBRIEF is a configuration option that determines whether brief descriptions are automatically generated for functions, classes, and other entities in the generated documentation.
-- OPTIMIZE_OUTPUT_JAVA this option is used to control the optimization of the generated output for Java.
-- EXTRACT_ALL this option determines whether Doxygen should extract documentation for all entities, even if they are not explicitly documented.
-- EXTRACT_PRIVATE this option determines whether Doxygen should extract documentation for private members.
-- EXTRACT_STATIC this option determines whether Doxygen should extract documentation for static members.
-- INPUT this option is used to specify the input files or directories for Doxygen.
-- RECURSIVE this option determines whether Doxygen should process source code files recursively in the specified input directories.
'''
echo "Set up Doxyfile..."
echo "**********************************************"
if hash doxygen 2>/dev/null; then # check if doxygen is installed
    cd docs
    doxygen -g # generates a default Doxygen configuration file named Doxyfile.
    sed -i ".bak" "s/PROJECT_NAME           = \"My Project\"/PROJECT_NAME           = \"$repo_name\"/" Doxyfile
    sed -i ".bak" "s/JAVADOC_AUTOBRIEF      = NO/JAVADOC_AUTOBRIEF      = YES/" Doxyfile
    # The overall effect of this command is to modify the Doxyfile in place by changing the value of the JAVADOC_AUTOBRIEF configuration setting from "NO" to "YES". And them same for all others.
    sed -i ".bak" "s/PYTHON_DOCSTRING       = NO/PYTHON_DOCSTRING       = YES/"  Doxyfile
#    sed -i ".bak" "s/OPTIMIZE_OUTPUT_JAVA    = NO/OPTIMIZE_OUTPUT_JAVA    = YES/"  Doxyfile
    sed -i ".bak" "s/EXTRACT_ALL            = NO/EXTRACT_ALL            = YES/"  Doxyfile
    sed -i ".bak" "s/EXTRACT_PRIVATE        = NO/EXTRACT_PRIVATE        = YES/"  Doxyfile
    sed -i ".bak" "s/EXTRACT_STATIC         = NO/EXTRACT_STATIC         = YES/"  Doxyfile
    sed -i ".bak" "s/INPUT                  =/INPUT                  = ..\/src/" Doxyfile
    sed -i ".bak" "s/RECURSIVE              = NO/RECURSIVE              = YES/"  Doxyfile
    rm Doxyfile.bak
    cd ..
else
    echo "Doxygen not installed, skipping..."
fi

echo "Creating README file..."
echo "**********************************************"
# Create blank README file
touch README.md
echo "**********************************************">>README.md
echo "# $repo_name">>README.md
echo "**********************************************">>README.md
echo "">>README.md
echo "## Description">>README.md
echo "">>README.md
echo "## Installation">>README.md
echo "">>README.md
echo "## Usage">>README.md
echo "">>README.md
echo "## Contributing">>README.md
echo "">>README.md
echo "## License">>README.md
echo "">>README.md
echo "## Author">>README.md
echo "Your name">>README.md
echo $(date '+%Y-%m-%d')>>README.md


echo "Creating Containerisation Files..."
echo "**********************************************"

# Build basic Dockerfile
touch Dockerfile
# use basic miniconda image
```

```
echo "FROM continuumio/miniconda3" >> Dockerfile
echo "" >> Dockerfile
# create project directory
echo "RUN mkdir -p $repo_name" >> Dockerfile
echo "" >> Dockerfile
# copy the repository in
echo "COPY . /$repo_name" >> Dockerfile
echo "WORKDIR /$repo_name" >> Dockerfile
echo "" >> Dockerfile
# install the conda environment
echo "RUN conda env update --file environment.yml" >> Dockerfile
echo "" >> Dockerfile
# activate the conda environment
# can't do it with dockerfile
# instead we have to edit bashrc to load it on login
echo "RUN echo \"conda activate $repo_name\" >> ~/.bashrc" >> Dockerfile
echo "SHELL [\"/bin/bash\", \"--login\", \"-c\"]" >> Dockerfile
echo "" >> Dockerfile
# as we are in the conda enviroment we can install pre-commit hooks
echo "RUN pre-commit install" >> Dockerfile

echo "Creating GitHub repository..."
echo "**********************************************"
# Initialize the Git repository
if hash git 2>/dev/null; then # check if git is installed
    git init
    git status
    git add .
    git commit -m "Initial commit"
else
    echo "Git not installed, exiting..."
    exit 1
fi

# Create the repository on GitHub (assuming you have the GitHub CLI installed)
# could also use GitLab or BitBucket CLI
if hash gh 2>/dev/null; then # check if GitHub CLI is installed
    gh repo create $repo_name --private -s .
#     git branch -vv

    # Push the initial commit to the remote repository
    git push origin master
else
    echo "GitHub CLI not installed, skipping GitHub repository creation..."
fi

# Display success message

echo "**********************************************"
echo "Blank Python repository created successfully!"
echo "**********************************************"
```

2.4 Vim

Vim is a text editor.
Some useful commands include:

3 Python

3.1 Data Types

Common data types include:

- int: integer
- float: floating point number
- str: string
- bool: boolean
- list: list of objects
- tuple: immutable list of objects
- dict: dictionary of key-value pairs
- set: unordered collection of unique objects

3.2 Control Flow

Common control flow statements include:

- if, elif, else
- for
- while
- break
- continue
- pass

3.3 Functions

Functions are defined using the def keyword.

```
    def function_name(arg1, arg2, arg3):
        # do something
        return something
```

The *args and **kwargs arguments are used to pass a variable number of arguments to a function. *args is used to pass a variable number of arguments to a function, while **kwargs is used to pass a variable number of keyword arguments to a function.

Lambda functions are anonymous functions. They are defined using the lambda keyword.

3.4 Comprehensions and Generators

Comprehensions are a way of creating lists, dictionaries and sets in a single line.
For example:

```
# list comprehension
squares = [x**2 for x in range(10)]
# dictionary comprehension
squares_dict = {x: x**2 for x in range(10)}
# set comprehension
squares_set = {x**2 for x in range(10)}
```

Generators are a way of creating iterators. They look like comprehensions, but they use parentheses instead of square brackets.
For example:

```
# generator
squares_gen = (x**2 for x in range(10))
```

For some generators, it can be too complex for a single line. In this case, you can use the yield keyword to create a generator function.
For example:

```
def squares_gen():
    for x in range(10):
        yield x**2
```

3.5 Magic commands

Magic commands are special commands that are not part of the Python language, but are part of the IPython or jupyter kernel.
Common magic commands include:

```
%whos # list of all assigned variables
%history -n 1-4 # list commands from prompts 1-4
%run filename.py # runs the python script filename.py
%timeit # times one line of code
%%timeit # times multiple lines of code
%debug # opens a debugger where an exception was raised
%rerun # rerun previously entered commands (can specify range)
%reset # delete all variables and definitions
%save # save some lines to a specified file
```

4 Advanced Python

4.1 Classes

Classes are defined using the class keyword.

```
class MyClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2
    def my_method(self):
        # do something
```

The __init__ method is called when an instance of the class is created.
Inheritance is used to create a new class that inherits the methods and attributes of another class.

```
class MySubClass(MyClass):
    def __init__(self, arg1, arg2, arg3):
        super().__init__(arg1, arg2)
        self.arg3 = arg3
```

4.2 Decorators

Decorators are used to modify the behaviour of a function.

```
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@my_decorator
def my_function():
    print("Hello world")

my_function()
```

The above code will print:

```
Before function
Hello world
After function
```

You can also return within the wrapper function.

```
def change_sign(func):
    def wrapper(*args, **kwargs):
        return -func(*args, **kwargs)
    return wrapper

def times_two(x):
    return 2e0*x

def product(x,y):
    return x*y
```

Decorators are often used for controlling access to functions with decorators such as @login_required.

Another common use for decorator is to debug functions.

```
    def debug(func):
        def wrapper(*args, **kwargs):
            print(func.__name__)
            return func(*args, **kwargs)
        return wrapper
```

Or to caches functions output to save time. i.e. We create a dictionary to store the output of the function, and check if the input is already in the dictionary. If it is, we return the output from the dictionary, otherwise we run the function and store the output in the dictionary.

4.3 Global and Local Variables

Variables defined outside of a function are global variables.
Variables defined inside a function are local variables.
Sometimes we want to modify a global variable inside a function. To do this, we need to use the global keyword.

```
    x = 1
    def my_function():
        global x
        x = 2
```

Variables created in modules and packages are GLOBAL and accessible within the namespace so can't be made local to the module or package. If you want to create variables that should only be accessed inside the module then preface them with _ which doesn't stop anything but does indicate to the programmer that they shouldn't use them.

5 Git

Git can be used locally or remotely. It can be run on desktop or upload the code to a server.

5.1 Create Repository

```
    $ git init
```

This creates a hidden folder called .git which contains all the information about the repository.

5.2 Add Files

```
    $ git add file.txt
```

This adds the file to the staging area.

5.3 Commit Changes

```
    $ git commit -m "commit message"
```

This commits the changes to the repository. Use –amend to change the commit message.

5.4 View History

```
    $ git log
```

This shows the history of commits.

5.5 Configurations

```
    $ git config --global user.name "Your Name"
    $ git config --global user.email "
```

This sets the user name and email.

6 Maintenance

We are going to discuess documentation, modularity, prototyping

6.1 Documentation

A __init__.py file in the package
Enforce a certain style for such as i, j, k, underscores etc.

7 Robustness of confidence

7.1 I/O

7.1.1 Storing parameters

Store your constant in a file for development
Standard package include configparser:

```
        import configparser as cfg
        input_file = sys.argv[1]
        config = cfg.ConfigParser()
        config.read(input_file)
```

A neater way is to use the argparse package:

```
        [cosmology]
        omega_m = 0.3
        omega_l = 0.7

        [hyperparameters]
        error_tolerance = 0.01
        depth = 3

        [flags]
        model = local
        do_polarisation = True
```

```

[output]
output_path = output/
output_name = data
```

and use the following code:

```
omega_m = config.getfloat('cosmology', 'omega_m', fallback
   =0.3)
```

7.1.2 Error Trapping

Use **try, except, else** blocks

```
try:
    # do something
except:
    # do something else
else:
    # do something else
```

7.2 Debugging

A few ways to debug:

1. Use **%debug** magic command

2. %run -d script.py

3. python3 -m pdb script.py

7.3 Unit Testing/Testing led development

Write test files.
Have the habit of write test files before development.
Use assert to test the code.
Make your test folder a package by adding __init__.py file.

7.3.1 Continuous Integration

Continuous intigration is a development practise where developers integrate code into a shared repository frequently.
Each integration can then be verified by an automated build and automated tests. This stops people from introducing errors into the code base.
Ideally, we need to run

1. code formatter
2. code linter
3. unit tests

before commiting. This is done by:

```
black src test
flake8 src test
pytest
```

where black is a code formatter, flake8 is a code linter and pytest is a unit test.
This can be automated by using pre-commit hooks, put a *.pre-commit-config.yaml.*
Example of a *.pre-commit-config.yaml* file:

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.0.1
    hooks:
    - id: check-yaml
    - id: end-of-file-fixer
    - id: trailing-whitespace
    - id: mixed-line-ending
    - id: debug-statements
- repo: https://github.com/psf/black
    rev: 23.11.0
    hooks:
    - id: black
        language_version: python3.9
    - id: black-jupyter
        language_version: python3.9
- repo: https://github.com/pycqa/flake8
    rev: 6.0.0
    hooks:
    - id: flake8
- repo: local
    hooks:
    - id: testing
        name: testing
        entry: pytest
        language: system
        files: ^test/ # ^ means "start with test/"
        always_run: true # run on all files, not just those staged otherwise it will not run unless you update the test file
```

7.4 CI on remotes

Ideally the checks should be run on the remote server.
Common CI services include:

1. Travis CI
2. Circle CI
3. GitHub Actions

Github does this via *runners*, which are scripts that run on the remote server.

8 Performance

8.1 Profiling

8.1.1 Timing Operations

1. *%timeit*: time a single line of code
2. *%%timeit*: time multiple lines of code, i.e. entire cell

In general, CPU is faster in addition and multiplication than division.
In python multiplication is same as addition and division, as most of the time is used in finding the variable.
Multiply by a number is faster than multiply by a variable due to the time to look up the variable.

8.1.2 Python Profiler

Profilers analyse the performance of your code, and tells you what parts are taking the most time and memory to run. In Jupyter Notebook, you can use the %prun magic command to run the profiler.
Alternatively, you can use the %load_ext line_profiler and %lprun magic commands to run the line profiler.
Another way is to run the profiler from the command line:

```
        $ python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

8.2 Algorithmic Complexity

8.2.1 Scaling with the example of sorting algorithms

Common complexity:

1. O(1): constant
2. O(log n): logarithmic
3. O(n): linear
4. O(n log n): linearithmic
5. O(2^n): exponential

Sorting Algorithms:

1. Selection Sort: select the smallest element and put it in the first place, then select the second-smallest element and put it in the second place, etc. The complexity is O(n^2)
2. Merge Sort: divide the list into two halves, sort each half, then merge the two sorted halves. The complexity is O(n log n)

8.3 Optimisation

8.3.1 Overview of modern computer architecture, memory caches and bandwidths

1. CPUs are much faster than memory by 200 times
2. Optimisation comes from fed CPU with enough data or using parallelisation
3. Three common measures of performance:
   (a) **Latency**: time to perform some action

   (b) **Size**: How much data can be stored

   (c) **Bandwidth**: amount of data per unit time

4. Two things to speed up code

   (a) Keep data local to the CPU

   (b) Reuse data

9 Public Release

9.1 Sharing Code

To share code, you should always include a License and readme file.

9.1.1 Licenses

Use a standard open source license, such as MIT, BSD, Apache, GPL, etc.

9.1.2 Readme

A file that tells user about the project. It should include the following:

1. title
2. description
3. contents
4. Installation
5. how to run
6. features
7. Frameworks used (Language/test/CI/Containerisation)
8. Build status, known bugs, future work
9. License

9.2 Sharing Data

Do not put data in github.
Data should be available in a public server, and provide bash script to download the data.

9.3 Sharing Environment

For python, this can be done by using a conda environment.

```
    $ conda env export -n my_env -f environment.yml
```

This is not ideal as it is not platform independent and operating system independent.
A better way is to use a containerisation software such as Docker.

9.3.1 Docker

Docker has these three components:

1. Dockerfile: a text file that contains all the commands to build a docker image
2. Docker image: a read-only template with instructions for creating a Docker container. It is stored in layers from OS up to the application. This allows images to build on top of each other.
3. Docker container: a runnable instance, i.e. computing environment of a Docker image

To create a docker image, use the following command:

```
    $ docker build -t my_image .
```

To create the container automatically for when we move machine or for sharing the project with people, in order to do so we need to create a Dockerfile, which is a text file that contains all the commands to build a docker image.