Linux and Python cheatsheet

Contents

1 Introduction to Linux

Linux is one of the most flexible operating systems available, running on devices as diverse as embedded devices, smartphones, personal computers, and large-scale supercomputers. On personal computers and smartphones, Linux usually can be operated via a *graphical user interface* (GUI, or desktop environment in Linux terms), while supercomputers often only offer access via a *command line interface* (CLI, also known as *terminal*). Within the CLI, a so-called *shell* processes user input, runs commands, and shows process output and error messages. Many of the codes we will use in this school do not offer any GUI. In the following subsections, we will introduce the basic Linux commands as well as important concepts and useful tools.

1.1 Getting information about a command

Information on how to use a particular command can often be obtained by running the command with `--help` as parameter. Furthermore, many commands have entries in the *system's manual pages* (or *man pages* in Linux terminology). These can be accessed by calling `man <command>`, navigated using ↑ and ↓, searched by typing `/<search pattern>`, and, finally, exited pressing `q`.

1.2 Autocomplete

Usually, autocomplete is enabled for the CLI and can be used by pressing ⇥. Autocompletion works for commands, directory paths and file names. Furthermore, the command history can be accessed by pressing ↑ or ↓.

1.3 Wildcard characters

- Wildcard characters are „placeholders“ that represent one or more characters.
- * will be replaced by any string.
- ? will be replaced by exactly one character.
- [...] will be replaced by the character(range).

1.4 Variables

- `export <var>=<value>`: declare variable (e.g., `export i=1` assigns the value 1 to variable `i`).
- `echo $(<var>)`: prints value of a variable to screen; can be combined with normal text and/or other variables (e.g., `echo "i is $i and j is $j"`).
- `$(($i+$j))`: if the values of `i` and `j` are integers, this returns `i+j`.
- `$(($i*$j))`: if the values of `i` and `j` are integers, this returns `i*j`.

1.5 Process control and output redirection

Processes and commands can be started either in the foreground (default) or in the background. To run a process or command in the background, the symbol `&` is added at the end (e.g., `vim &` starts the text editor *vim* in the background). Note that the output of a command running in the background may still be written on the screen. Below we will introduce a few ways to avoid this behavior. First, however, we introduce some ways to control processes and commands.

- `jobs`: command to show status of processes running in the background.
- Ctrl+ z : stop a process.
- `fg`: get a process to the foreground (also works with stopped processes).
- `bg`: run a stopped process in the background.
- Ctrl+ c : abort process.

There are several ways to redirect the output of a process or command to a file or to a different command.

- `<command> > <file>`: writes the output of a command to a new file instead of the screen (overwrites existing files).

- `<command> » <file>`: appends the output of a command to an existing file instead of the screen (creates a new one if the file does not exist)
- `<command> | <another_command>`: redirects output to another command.
- `<command> | tee <file>`: redirect output to `tee`, a command that writes the output to a file and to the screen (like `>`; `tee -a` appends to a file like `»`).
- `<command> && <another-command>`: conditional execution of another command: `<another-command>` only will be executed if `<command>` terminates without error.
- `<command>; <another-command>`: execute several commands after each other (independent of whether one or more of the commands terminate with an error).

1.6 Navigating

After logging-in, one finds oneself in the home directory (e.g., /home/student). For navigating through the file system using the CLI only three commands are needed:

- `pwd`: shows the current directory.
- `ls <path>`: show the content of a directory or information about a file. E.g., `ls /home/atom` shows the content of the *home* directory of the user *atom*. If called without `path` it shows the content of the current directory.
- `cd <path>`: change the directory, if run without argument changes to home directory, e.g., `cd /home/atoms` moves to the *home* directory of the user *atom*. Use `cd ..` to move to the parent directory of the current directory.

1.7 Creating files and directories

- `mkdir <dir>`: create a directory.
- `touch <file>`: creates file `<file>`; updates access time if `<file>` exists.
- `cp <src> <dest>`: copies file `<src>` to `<dest>`.
- `mv <src> <dest>`: moves file `<src>` to `<dest>`.
- `rm <file>`: removes file (to remove a directory `rm` must be called with `-r` parameter).
- `rmdir`: removes directory if it is empty (gives error message if directory is not empty).

1.8 Extracting information with text files

To obtain information from text files, it is not necessary to open the file with a text editor. Following are some of the most useful commands for the extraction of information from text files.

- `cat <file>`: prints content of file to screen.
- `head <file>`: prints first lines of file.
- `tail <file>`: prints last lines of file.
- `grep <pattern> <file>`: searches for <pattern> in <file> and prints it; <pattern> can contain wildcard characters(see subsection 1.3).

1.9 *Vim* text editor

Vim is a text editor for the command line with a minimal interface but a very rich set of features. *Vim* offers several different modes that can be entered using different combinations of keys, but all are exited using Esc. Independent of the current mode, one can navigate using the arrow keys (↑, ↓, ←, →).

Default mode: entered when starting vim or exiting any other mode. In this mode one can:

- Navigate with the arrow keys.
- ⇧+g: jump to the last line.
- g+g: jump to the first line
- d+d: delete line.
- del: delete character
- u: undo.
- :q (:q!): (force) exit without saving.
- :x: save and exit.
- :w: save.
- :w `<file>`: save to `<file>`.
- : `help`: enter vim manual.
- Enter different modes (see below).

Insert mode (press i): In this mode one can type as in any other text editor.

Visual mode (Ctrl+v or ⇧+v): in this mode, one can select lines or blocks of text and copy (y), cut(x), paste(p), or delete (del) it.

Search mode (/+`<pattern>`): searches the text file for `<pattern>`. Press n (⇧+n) to go to next (previous) result.

1.10 Scripts

Scripts are a comfortable way to combine commands for executing more complex tasks and for automating recurring tasks.

- Script starts with "`#!`" followed by the interpreter, e.g., "`#!/bin/bash`" for *Bash* scripts.
- Making a script executable: `chmod u+x <script>`.
- Conditionals (`if`) and loops (`for, while`) can be used.

1.10.1 if conditionals

- Action is only performed, if a condition is true.
- Syntax: `if [` *condition* `]; then` *action*`; fi`
- Note the whites-spaces between the square brackets and the condition; these are needed!
- Example: print "True", if the file *file* exists:
  `if [ -e file ]; then echo 'True'; fi`

1.10.2 for loops

- Action is repeated for all given arguments.
- Syntax: `for` *variable* `in` *arguments*`; do` *action*`; done`.
- Example: print numbers 1 to 10:
  `for i in {1..10}; do echo $i; done`

2 Introduction to *Python3*

Python3 is an easy and flexible, interpreted programming language. Nowadays, *Python3* is widely used in science, mainly for the analysis and representation of data as well as for machine learning. Many useful extensions to *Python3* exist and here we will introduce two of the most useful ones, namely *Numpy* and *Matplotlib*. There are two ways to run a *Python3* script: first, one can invoke the *Python3* interpreter (`python3`) with the name of the script as argument (e.g., `python3 script.py`). Second, one can make the script executable (see above) and add `#!/usr/bin/env python3` at the top of the script. Then, the script can be executed like this: `./script.py`. Typing `python3` without any arguments invokes the *Python3* interpreter which allows executing python commands interactively. An alternative to the interactive mode of *Python3* is `ipython`.

2.1 Variables

There are almost no limits in *Python3* regarding variable names, only that they cannot start with a number or special character (except underscore). Variables can be of different types, e.g., string, float, int; however, one does not need (and with the default *Python3* interpreter cannot) explicitly specify the type (internally, the interpreter determines the type from the value the variable is assigned). Valid declarations of variables would be, for example:

- `n = 3`: integer.
- `x = 2.4`: float.
- `s = 'hello'`: string.
- `z = 2 + 3j`: complex number.
- `l = [1, 2, 3]`: list.
- `t = True`: bool with value true.
- `f = False`: bool with value false.

2.2 Basic *Python3* syntax

- `import <ext>`: imports extension to *Python3*, e.g., `import numpy`
- `import <ext> as <namespace>`: imports extension to *Python3* and assigns namespace, e.g., `import numpy as np` (numpy functions can now be accessed as `np.<func>` instead of `numpy.<func>`).
- `print()`: prints on screen whatever is between the parenthesis; can be text or variables or any combination thereof, e.g., `print('hello', s, z, l)`; different elements are separated by a `","`.
- `+, -, *, /, **`: add, subtract, multiply, divide, and take the power of some variable.
- `#`: comment line.
- `''' <comment> '''`: multi-line comment.
- `if-else`: can be used to test conditions (must evaluate to `True` or `False`) and execute based on the truth value of these conditions. The following code tests whether the value stored in `x` is larger than 10 and prints `x` if that is the case or `x/2` if not.

```
if x > 10:
   print(x)
else:
   print(x/2)
```

- `for`: allows to loop over items of iterables (e.g., lists, tuples, strings, ...). The following code loops over a list and prints each item separately:

```
for i in [1, 2, 3]:
   print(i)
```

2.3 Introduction to *Numpy*

Numpy provides many useful functions for working with matrices and much more.

- Assume numpy was imported as np: `import numpy as np`
- `np.loadtxt(filename)`: loads matrix from textfile (e.g., `M = np.loadtxt('./matrices/matrix.dat')`)
- `np.savetxt(filename, M)`: saves matrix `M` in textfile with name `filename`.
- Note: both np.loadtxt and np.savetxt only work with one- or two-dimensional matrices.
- `np.linspace(start, end, numpoints)`: returns an array with `numpoints` entries equally distributed from `start` to `end`.
- `np.exp(x), np.sin(x), np.cos(x), ...`: numpy also provides many mathematical functions, all of which can take matrices or scalars as input. For matrices, the functions are applied element-wise.
- `x = M[:, i]`: save i-th column of `M` to `x`.
- `M * i`: element-wise multiplication of matrix `M` with scalar `i`; also works with `+, -, /, **`.
- `np.amax(M)` and `np.amin(M)`: return the maximum and minimum value of a matrix.

2.4 Introduction to *Matplotlib*

Matplotlib is a framework for *Python3* to plot data. Plots created with *Matplotlib* can be saved in different formats (e.g., pdf, jpg, png, ...)

- Assume matplotlib.pyplot was imported as plt: `import matplotlib.pyplot as plt`.
- `fig = plt.figure()`: initializes a new figure.
- `ax = plt.gca()`: initialize axes.
- `ax.plot(x, y, **kw_args)`: creates 2D line plot from x and y; **`**kw_args`** can be different keyword arguments, e.g., `linewidth, color, marker, label`. The `label` keyword is useful for plotting a legend for each 2D line plot. Example: `ax.plot(x, np.sin(x), linewidth=3, label='sin')`. Can be called multiple times with different parameters.

- `ax.legend()`: creates legend for each line (needs to be called after the last `ax.plot` call).
- `ax.set_xlabel(xlabel), ax.set_ylabel(ylabel)`: set labels of x and y axis.
- `plt.show()`: shows the plot; only works on Linux with graphical user interface.
- `plt.savefig(filename)`: saves plot to file; type of file is determined by file ending (e.g., .pdf, .png, ...)

3 Self-test

In order to determine whether you possess (very) basic knowledge of *Linux* and *Python*, you can solve the following tasks:

1. Use the *man pages* to answer the following questions:
   - `ls`: By default `ls` lists the content of a directory in several columns. Which parameter makes `ls` print all entries in only one column?
   - `mkdir`: Which parameter allows to simultaneously create a directory and a subdirectory in this directory?
   - `cp`: Which parameter allows you to copy directories?
   - `head, tail`: Which parameter allows to specify the number of lines that is printed?
   - `grep`: Which parameter tells `grep` to do a case-insensitive search of the pattern?
2. `vim`: How can you delete the first character of each line (hint: use the correct visual mode)?
3. Write one *Bash* script (see subsection 1.10) named *create_files.sh* that creates the following files and directories:
   - Create a directory `exercise2` with subdirectories `x_squared` and `x_cubed`.
   - In the directory `exercise2/x_squared` create a file *squares.txt* containing two columns, with the first column containing the integer numbers from 1 to 15, and the second column containing the squares of these numbers (i.e., x^2).
   - Similarly, in the directory `exercise2/x_cubed` create a file *cube.txt* containing the numbers from 1 to 15 and their cubes (i.e., x^3) in two columns
4. Write a *Python3* script named *plot_ data.py* that does the following:
   - Load the data in the files created in the previous tasks into matrices.
   - Create a vector with the values *y=exp(x)* from 1 to 15.
   - Normalize the y values for all three data sets so that their maximum value is 1.

- Plot all three data sets with a clear legend and labels.
- Save the plot to a file.