

JULIUS-MAXIMILLIANS-UNIVERSITÄT  
WÜRZBURG

MASTER

CELL AND INFECTION BIOLOGY

---

## **F1 Internship - Bioinformatics**

---

*Author*  
Christine LIAO

*Supervisor*  
Dr. Konrad FÖRSTNER

August 14, 2018


# Contents

<b>1</b>	<b>Unix Shell</b>	<b>1</b>
1.1	Definitions . . . . .	1
1.2	Symbols . . . . .	2
1.3	Commands . . . . .	3
1.3.1	<i>How to navigate around shell</i> . . . . .	3
1.3.2	<i>How to create, copy and delete</i> . . . . .	3
1.3.3	<i>How to count, sort and pipe</i> . . . . .	4
1.3.4	<i>How to write a loop and shell scripts</i> . . . . .	4
1.3.5	<i>How to find something in a text</i> . . . . .	5
1.4	Shortcuts . . . . .	6
<b>2</b>	<b>Python</b>	<b>7</b>
2.1	Definitions . . . . .	7
2.2	How to install Python . . . . .	8
2.2.1	<i>IPython</i> . . . . .	8
2.2.2	<i>Python</i> . . . . .	9
2.3	Commands . . . . .	9
2.3.1	<i>Math</i> . . . . .	9
2.3.2	<i>Order of operation</i> . . . . .	9
2.3.3	<i>Variables</i> . . . . .	10
2.3.4	<i>Buil-in functions</i> . . . . .	10
2.3.5	<i>Defining a function</i> . . . . .	11
2.3.6	<i>String type</i> . . . . .	11
2.3.7	<i>Input and output</i> . . . . .	12
2.3.8	<i>Escape sequences</i> . . . . .	13
2.3.9	<i>Function design</i> . . . . .	13
<b>3</b>	<b>Markdown</b>	<b>15</b>
3.0.1	<i>How to convert Markdown files into pdf</i> . . . . .	15
3.1	Commands . . . . .	15
<b>4</b>	<b>Emacs</b>	<b>16</b>
4.1	Symbols . . . . .	16
4.2	Commands . . . . .	16
4.2.1	<i>How to create file, save, copy and paste</i> . . . . .	16
<b>5</b>	<b>Secure Shell (SSH)</b>	<b>17</b>
5.1	Commands . . . . .	17
5.1.1	<i>How to connect remotely</i> . . . . .	17
5.1.2	<i>How to generate SSH key</i> . . . . .	17

<b>6</b>	<b>Git</b>	<b>19</b>
6.1	Definitions . . . . .	20
6.2	Adding a new SSH key to Github . . . . .	20
6.3	Making a public repository private . . . . .	21
6.4	Commands . . . . .	21
6.4.1	<i>How to configure Git</i> . . . . .	21
6.4.2	<i>How to create and save a repo</i> . . . . .	21
6.4.3	<i>Viewing staged and unstaged changes</i> . . . . .	22
6.4.4	<i>others</i> . . . . .	23
<b>7</b>	<b>Exercises</b>	<b>24</b>
7.1	ROSALIND . . . . .	24
7.2	Konrad's repo . . . . .	25
<b>8</b>	<b>References</b>	<b>28</b>
<b>9</b>	<b>Acknowledgements</b>	<b>28</b>

# 1 Unix Shell

Unix is an operating system (OS) developed in the '60s. An OS is a group of programs which makes the computer work. Each program performs limited functions with a unified file system.

Unix systems also have a graphical user interface (GUI). An interface is a way that allows humans to communicate with the machine by using visual language like symbols. For example, the disk icon, , indicates that upon clicking, the file will be saved. An example of graphical operating system is the Microsoft Windows.

Command line interface (CLI) is another way to communicate with the machine by using commands. An example of CLI is shell, which allows users to enter commands as text and then executes the operation.

Unix shell is both command language and a scripting programming language. A shell script is a series of commands that is often used, and can be saved in a text editor, so we don't have to type them every time. A text editor is a type of computer program that edits plain text. An example of such programs is the "notepad" software. Text editors are part of the operating systems packages, and can be used to change configuration files and programming language source code. In this chapter, nano text editor is used.

The terminal emulator (or simply the Terminal) provides CLI to control UNIX-based operating system and lives in Utilities folder in Applications. When launched, it shows a UNIX command-line environment, which is the shell. There are different types of shell; Apple uses Bash.

Important: Commands in Unix are shell-specific. In this report, the commands are to be used on a Mac.

## 1.1 Definitions

`shell` language to run other programs using command-line interface instead of GUI.

`command-line interface` uses read-evaluate-print loop (REPL) to interpret when we type a command and press the Enter key: it reads, executes and prints the output.

`bash` a default shell in Mac.

`kernel` core of the computer's operating system, where controls everything.

`bin` a folder that stores built-in programs.

`temp` a folder that stores temporary files.

`flag` adds a marker to file or directory to show what they are.

`nano` text editor for Unix systems, creates only plain character data. You can save text file or commands.

`argument` tells the command what to operate on (file or folder).

`input file` the first file during an execution command.

`output file` the second file during an execution command.

`pipe` a vertical line, `|`, separating two commands, where the output of the command on the left side will be the input for the command on the right side.

`loop` a set of commands that allows an operation to be executed repetitively, automatically.

`iteration` each time the loop completes the operation is called iteration.

`grep` global/regular expression/print. It is a sequence of operations. It finds and then prints lines in a text that contains the pattern that we are looking for. A pattern can be a word.

## 1.2 Symbols

`$` a prompt.

`/` root directory when placed in front of file or folder; separator if between names.

`~` current user's directory.

`*` wildcard

`?` wildcard

`|` pipe

`>` a prompt when writing commands in a loop. This symbol is to show that the commands are not complete. In shell, it can also mean "to redirect".

`$` can also assign something to become a variable inside a loop. For example, `$filename` means to treat the variable as a variable name and substitute its value in its place, rather than treat it as a text or as a command.

## 1.3 Commands

### 1.3.1 *How to navigate around shell*

```
$ ls    shows directory, in alphabetical order.
$ man ls  shows commands manual.
$ pwd    prints working directory.
$ Ctrl-l  cleans the screen.
$ PS1 ='$ '  starts the prompt with only $ sign.
$ ls -F    adds trailing "/" to the names of directories. It is a "Flag" (option).
$ man ls    manual listing other flags.
$ ls -l    shows file/directory names, size and time of last modification.
$ ls -l-h   makes file size more readable to 2 digits.
$ ls -R    lists directories and sub-directories.
$ ls -R -t   directories are sorted by time and last change.
$ ls -F     lists content by showing what kind of file they are.
$ cd Desktop  changes directory to Desktop.
$ cd Folder   changes directory into a folder, subdirectory.
$ cd File     changes directory into a file, sub-subdirectory.
$ cd ..      goes one level up of the directory (in the direction of right to left,
ie. closer to Home).
$ cd         returns to home directory.
```

### 1.3.2 *How to create, copy and delete*

```
$ mkdir thesis  creates a directory (folder) called thesis.
$ touch file.txt  creates a file.
$ nano draft.txt  creates a file in nano. To exit Nano and return to shell
environment, type Ctrl-X, Y, and press Enter.
$ rm draft.txt   deletes the file PERMANENTLY. There is no trash in UNIX
shell.
$ rm -r thesis   deletes folder and its content PERMANENTLY.
$ mv thesis/draft.txt  moves the file into a directory.
$ mv thesis/draft.txt thesis/quotes.txt  file name is changed to quotes.txt.
$ cp fileOne.txt fileTwo.txt  copies the content of the first file into the
second file.
$ mv *.txt  moves all files ending .txt to original home directory.
$ cp one.txt two.txt thesis  copies the two files into a folder called the-
sis.
$ ls -R -t   displays directories recursively (lists their sub-directories in al-
phabetical order).
-t  lists things by the time of the last change.
$ rm -i  -i is a flag, for interactive, prompts the user whether to proceed with
the delete operation.
```

### 1.3.3 *How to count, sort and pipe*

\$ wc \*.pdb word counts (number of lines, words, characters in file).  
\$ p\*.pdb any file starting with "p" letter that is a pdb file.  
\$ p?.pdb any file that has only letter after "p", eg. pi.pdb.  
\$ p\*.p?\* any file starting with "p" and ends with "." or "p" and at least one more character.  
\$ wc -l\*.pdb counts number of lines.  
-w counts number of words.  
-c counts number of characters.  
\$ wc -l \*.pdb > file.txt redirect output to a file (because shell cannot print output on the screen) and erase the content of the file.txt.  
>> redirects output to a file by appending to the existing content without erasing it. It's different from >.  
\$ cat file.txt displays content of the file, all at once.  
\$ less file.txt shows content but stops when screen is full. Go forward with spacebar, b, q.  
\$ tail -2 animals.txt >> animalsUpd.txt adds last two items of the list to the file.  
\$ sort Letters.txt sorts in alphabetical order, without changing original file, shows output on the screen.  
\$ sort -n number.txt sorts in numerical order.  
\$ sort -n number.txt > sorted-number.txt redirects the result to second file.  
\$ head -n 1 number.txt shows the first line of the file on the screen.  
\$ tail -n 3 number.txt shows the last 3 lines of the file on the screen.  
\$ sort -n length.txt | head -n 1 executes the left command first, then using that file as input to execute command on the left. Result: it sorts in numerical order, then displays on the screen the first line.  
\$ wc -l allows you to enter texts. ctrl-D to finish.  
\$ uniq removes adjacent duplicated lines.  
\$ cut -d , -f 2 file.txt cuts out sections of each line of a file.

### 1.3.4 *How to write a loop and shell scripts*

Example of a Loop:

```
$ for filename in document1.txt document2.txt
>do
>     head -n 3 $filename
>done
```

This is a typical loop structure. The word for is the repeat command. The operation finds the first 3 lines of the file. This loop will do this operation for document 1 and 2. The word \$ filename is just a variable; it could have been any word.

`$ echo` prints the output on the screen.  
`$ history | tail -n 5` shows history of the last 5 commands.  
`$ history | tail -n 5 > recent.sh` save the last 5 commands in nano.  
`$ !123` enter number of the line you want from history.  
`$ !!` shows your last command.  
`$ !` retrieves last words of the last commands.  
`$ nano newfile.sh` creates a new file in text editor nano. This is where we write a shell script, then run the script in shell.

In nano:

```
head -n 15 octane.pdb | tail -n 5
```

we could also use nano to write a script by typing a command as such.

`$ bash newfile.sh` it runs the script written in bash.

In nano:

```
head -n 15 $1 | tail -n 5
```

this has the same result as the above nano script because `$1` means the first filename on the command line.

In nano:

```
head -n $2 $1
```

again `$2` is a position variable, which indicates second command-line argument.

### 1.3.5 How to find something in a text

`$ grep not document.txt` finds and prints lines in this document that has the word "not".  
`$ grep The document.txt` finds and prints lines that contains the word "The" but also words like "Thesis".  
`$ grep -w The document.txt` limits word boundary. It limits the search to only "the" and words like "thesis" will not appear.  
`$ grep -w "is not" document.txt` finds and prints lines that contains multiple words.  
`$ grep -n "it" document.txt` finds and prints lines that contain any words with "it", such as **it**, **with**.  
`$ grep -n -w "the" document.txt` finds lines that contain "the" and prints also the number of the line where "the" is found.  
`$ grep -n -w -i "the" document.txt` the flag `-i` means case-insensitive.  
`$ grep -n -w -v "the" document.txt` where `-v` is to invert the search, meaning lines that do not contain the word "the".  
`$ man grep` manual, shows many other flags of grep. Use `q` to exit/esc.  
`$ grep -E "^o" document.txt` finds lines that contain an 'o' in the second position, such as **toner**, **software**. The `-E` flag looks for pattern that is called **regular expression**. We put the pattern in quotation to prevent the shell trying



to interpret it. The `^` finds the match to the start of the line. The `.` can be any character (just like `?` in the shell), and `"o"` matches the letter `"o"`.

`$ find .` is similar to the command `ls` in Unix. Again, `.` means the current working directory, where the search will take place. `find` will list every file and directory under the current working directory.

`$ find . -type d` finds things that are directories. Here it doesn't list things in any order.

`$ find . -type f` finds all that are files.

`$ find . -name *.txt` find files ends with `txt` but only those saved in this directory and not in the subdirectories.

`$ find. -name '*.txt'` finds `txt` files in this directory and the subdirectories. So here `*` acts as a true wildcard.

## 1.4 Shortcuts

`Ctrl+Z` works as `esc`.

`Tab` autocompletion, revisit the most recent commands use. Avoid re-typing a command, saves time.

`Up&Down` keyboard arrows show previous commands typed.

`spacebar` forward one page.

`b` backward one page.

`q` quit and return to Terminal prompt `$`.

In nano:

`Ctrl+O` "writing out", saves the file.

`Enter`, `Ctrl+X` to exit nano back to shell.

## 2 Python

Principles of Python:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Readability counts.

Excerpts from *Zen of Python*

Python programming language was created by Guido van Rossum in 1991. Python's minimalist philosophy emphasizes on readability and fun to use, hence the name as a tribute to *Monty Python*. Therefore, code that is hard to understand is *unpythonic*.

It is an open source software that uses dynamic typing. In programming languages, a type system is a set of rules that assigns a property called type to the various constructs of a computer program, like variables, expressions, functions or modules.

Python codes can be run either via an interpreter or within a shell.

### 2.1 Definitions

Jupyter Notebook is an web-based interactive development environment, that allows Python codes to be executed (and also other programming languages like R). It can also become committed for version control repositories like github. Markdown can be used to generate a notebook environment for writing and developing code.

**type** A type is a set of values and operations that can be performed on those values. There are two numeric types :

- **int** integer, like 3, 0, -2.
- **float** an approximation to a real number, like 5.6.

**string** is data in the form of text. A string is a series of letters, numbers or symbol connected in order like a string. A sequence of characters. Starts and end with ' or ". Strings are values.

**syntax** rules that allow symbols to form valid combinations.

**semantics** if the symbols together make up a meaningful expression.

**variables** are numbers which are stored in computer memory.

**argument** an expression that appears between parentheses of a function call.

**function call** asks Python to evaluate a function.

**def** a keyword indicating a function definition.

**parameter** a variable that is within parentheses of a function definition. Then this parameter gets a value from expressions in a function call.

**return** a keyword indicating the result of a function.

**Boolean** is a data type that can either be true or false, intended to represent the two truth values of logic.

**data type** can be numbers and Booleans.

## 2.2 How to install Python

There are two ways to run Python:

### 2.2.1 IPython

The Jupyter Notebook contains an interactive shell built with Python (iPython), which supports interactive data visualization and GUI. Easy to use and commands come in different colors. Coding spacing is automatic and it's a great learning environment.

1 - Download Anaconda to install Python and Jupyter.

2 - From the Terminal, type:

\$ `jupyter notebook` to launch the Jupyter Notebook.

3 - Once in Jupyter Home, click:

New on upper right Menu, and select Kernel, Python 3 to start a new Python file.

4 - After working on a file, click on the menu:

File, Close and Halt to shut down the kernel.

**How to Python** Once in Jupyter Notebook:

1 - In `[ ]` this is a Cell, where computation takes places (can be code or text). The default format is code and it can be easily switched to text (eg. Markdown) by clicking the menu bar of Code and chose Markdown instead. Enter a code in the Input line.

2 - Ctrl-Enter

3 - Out [ ] result is computed in the Output line. And a new input line is added below.

**Shortcuts** In Jupyter notebook:

A inserts a new cell above the current cell.

B inserts a new cell below.

Ctrl-Enter runs the cells.

M changes the current cell to Markdown.

Y changes Markdown back to code mode.

D+D (pressing the key twice) deletes the current cell.

Tab click tab while completing a code, Jupyter will autocomplete it or suggest codes.

### 2.2.2 Python

This uses command-line interface.

1 - Download [Python](#) and Active TCL (Mac).

2 - Run Python either from the Terminal by typing `$ python` or run IDLE, the Python visualizer that is a friendlier programming environment, which comes with Python.

**How to Python** The prompt for Python is `>>>`. Once this appears on the Terminal, we are in Python programming environment.

## 2.3 Commands

### 2.3.1 Math

Operators	Operation	Examples	Output
+	plus	<code>3 + 2</code>	5
-	minus	<code>3 - 2</code>	1
*	multiplication	<code>3 * 2</code>	6
/	division	<code>3 / 2</code>	1.5
**	exponent	<code>3 ** 2</code>	9
//	integer division	<code>3 // 2</code>	1
%	remainder	<code>3 % 2</code>	1

### 2.3.2 Order of operation

When multiple operators are combined in a single expression, the operations are evaluated in a specific order.

Operators	Order
**	first
- (negation)	
*, /, //, %	
+, -	last

### 2.3.3 Variables

We can assign numbers to a variable, to form a general assignment statement:

```
variable = expression
```

```
>>> base = 20
>>> base * 2
40 This is the output.
```

Rules for variable names in Python:

- Name must start with a letter or \_.
- Name must contain only letters, digits, and \_.

Rules for executing an assignment statement:

- Evaluate expression on the right side of =, to produce a value. This value has a memory address.
- Store the memory address of the value in the variable on the left side of =.

```
>>> x = 2
>>> y = x + 1
>>> x = 10
```

If we run the code, x has been changed to value 10, but y = 3, where x is still 2.

### 2.3.4 Built-in functions

This is a function call, where max is the function and arguments are 4 and 2.

```
>>> max(4,2)
4
```

```
>>> dir(__buildins__)
```

we get a long list of build-in functions available in Python.

```
>>> help(abs)
```

describes what this build-in function does. For abs it will return the absolute value of the argument.

```
>>> help(pow) shows that this is the same as x**y.
>>> pow(2, 5)
32
```

### 2.3.5 Defining a function

**Function def** allows us to define our own functions. Here, `f` is the name of the function, `x` is the parameter, followed by colon, as follows:

```
def function_name(parameters):
    body
```

```
>>> def f(x):
        return x ** 2
```

If we enter `x = 3`:

```
>>> f(3)
9
```

Example: calculate area

```
>>> def area(base, height):
        return base * height / 2
```

```
>>> area(3, 4) if we enter these values for base and height.
6
```

**Docstrings and Function help** Built-in function `help` displays the docstring from a function we defined.

```
>>> def area(base, height):
        """(number, number) -> number
        Return the area of a triangle with
        dimensions base and height.
        """
```

```
>>> help(area)
area(base, height)
(number, number) -> number
Return the area of a triangle with dimensions base and height.
```

### 2.3.6 String type

Strings can be values when expressed in quotations:

```
>>> sunny_greeting = 'What a nice day!'
>>> sunny_greeting
'What a nice day!'
```

```
>>> cloudy_greeting = "Wow, you'r dripping wet."
Use double quotations to avoid this message: SyntaxError: invalid syntax
```

Strings can be concatenated using `+`, multiplied using `*`, or combined in a different order using parenthesis.

```
>>> 'May' + 'flower'
'Mayflower'

>>> puzzle_start = 'I saw May'
>>> noun = 'flower'
>>> punctuation = '!'
>>> puzzle_start + noun + punctuation
'I saw Mayflower!'

>>> 'ha' * 5
'hahahahaha'

>>> 'hi' + 'ha' * 5
'hihahahahaha'

>>> ('hi' + 'ha') * 5
'hihahihahihahihahiha'
```

### 2.3.7 *Input and output*

**Function print** prints a sequence of arguments for a user to read. It puts spaces between its arguments. Note that quotations are not part of the string

```
>>> def square_print(num):
    print(num ** 2)
>>> result = square_print(4)
16.

>>> def square_return(num):
    return num ** 2
>>> result = square_return(4)
Here there is no output. But if we added:
>>> new_result = result + 1
17

>>> print("Strings are fun!", "Isn't it?")
Strings are fun! Isn't it?
```

**Function input** get a string from the user. It prompts user to enter an answer.

```

>>> input("What is your name?")
What is your name? April
'April'
>>> name = input("What is your name?")
>>> name
'April'

>>> location = input("What is your location?")
What is your location? Toronto

>>> print(name, "lives in", location)
April lives in Toronto

```

So what we entered for name or location is referred to as type `str`.

### 2.3.8 Escape sequences

```

\n newline
\t TAB
\\ backslash(\)
\' singlequote(')
\" doublequote(")

```

### 2.3.9 Function design

The design of function has five parts:

1. **Examples:** What should the function do? Type some example calls. Then pick a name (often a verb).
2. **Type of contract:** What are the parameters types? What type of value is returned?
3. **Header:** Pick a meaningful parameter name.
4. **Description:** Describe every parameter and the return value.
5. **Body:** Write the body of the function.

Example: We want to convert Fahrenheit to Celsius.

1. Examples:  
 32 Fahrenheit is equal to 0 degrees in Celsius.  
 212 Fahrenheit is equal to 100 degrees in Celsius.

2. Type Contract:  
 (number) - >number

3. Header:  

```
def convert_to_celsius(fahrenheit):
```



4. Description:

Return the number of Celsius degrees equivalent to Fahrenheit degrees.

5. Body:

```
return (fahrenheit - 32) * 5 / 9
```

Putting all together in a python shell script:

```
def convert_to_celsius(fahrenheit):  
    ''' (number) -> number  
  
    Return the number of Celsius degrees equivalent to Fahrenheit  
    degrees.  
  
    >>> convert_to_celsius(32)  
    0  
    >>> convert_to_celsius(212)  
    100  
    '''  
  
    return (fahrenheit - 32) * 5 / 9
```

### 3 Markdown

It is a markup language with plain text formatting syntax. It is an easy-to-read, easy-to-write plain text format, contrary to markup language (Rich Text Format or HTML), which is marked up with tags and formatting instructions. It can be easily converted into HTML.

Markdown is a way to style text on the web and also for general usage. It employs # or \* to format the document words in bold, italic, or creating lists. The document written in Markdown has the file extension of .md and it is saved in plain text. Since .md files are just plain text files, they can be opened with any text editors, like TextEdit (Mac).

Pandoc is an open-source software that converts one markup format into another, for example from `.md` into `.pdf` (Portable Document Format).

### 3.0.1 How to convert Markdown files into pdf

```
1-$ pandoc -o newfilename.pdf inputfile.md
2-$ evince newfilename.pdf
```

### 3.1 Commands

From Jupyter Notebook drop-down menu, choose:

- 1 - code to begin writing plain text.
- 2 - Markdown to mark as Markdown.
- 3 - click run cell or Ctrl+Enter to convert into plain text.

Headers:

```
# huge size and in bold.
## big size and in bold.
##### small size.
```

## Lists using bullets

```
*Item 1
*Item 2
    * Item 2a
    * Item 2b
```

Emphasis:

```
*Text will be italic*
_This will also be italic_
```

## Lists using numbers

```

1.  Item 1
1.  Item 2
1.  Item 3
    1.  Item 3a
    1.  Item 3b

```

```

**Text will be bold**
__This will also be bold__

_You **can** combine them_

```

Spaces between letters:

[illegible]

## 4 Emacs

Emacs is one of the most popular and powerful text editors. It offers much longer lists of commands than other UNIX-based text editors and the ability to extend the interface. Here, GNU/Emacs is used. All GNU (GNU is Not Unix) software has this prefix.

The configuration file for emacs is `init.el`. Everytime we save a file, Emacs creates another file with the name "`filename~`". This tilde (`~`) file is the previous version of the file. It will be in the same dir. Also, Emacs auto-saves everything we type to a file with the name "`filename`". If we quit Emacs without saving, this auto-save file can be retrieved.

### 4.1 Symbols

`C` stands for control in the keyboard.

### 4.2 Commands

#### 4.2.1 *How to create file, save, copy and paste*

`C-x C-f` opens a file, asks for the file name. If cannot find it, creates the file.

`C-x C-s` saves the file without a prompt.

`C-x s` save all files with a prompt.

`M-x` to install packages.

`C-s C-w` save the file with a different name.

`C-k` deletes the whole line, puts it into a clipboard.

`C-y` pastes whatever in the clipboard at the cursor.

`C-space` starts marking/highlighting a region.

`M-w` copies this region into the clipboard.

`C-x C-c` quits Emacs.

## 5 Secure Shell (SSH)

SSH is to gain access to another computer/server with greater computational capacity and execute programs there. The first step is to log in to that computer, called remote login, using passwords. Then, the commands are passed to a shell running on the remote computer. That shell runs our commands and send back output to our computer.

The tool we use to log in remotely is the secure shell, or SSH. By entering name of computer and password, we could connect remotely. In order to avoid typing the passwords over and over again, SSH key was created to tell the remote computer that it should always trust us.

To accomplish this, a secure login must be established. This is done by a cryptographic network protocol. SSH keys come in pairs, a **public key** that gets shared with services like Github, and a **private key** that is stored in my computer. If the keys match, then access is granted.

In other words, SSH uses public-key cryptography to authenticate the remote computer and allow it to authenticate the user. This is done only ONCE. Furthermore, the cryptography behind SSH keys ensures that no one can reverse engineer my private key from the public one. Here, SSH is used to connect to Github without typing username or passwords every time.

### 5.1 Commands

#### 5.1.1 *How to connect remotely*

1 - From the Terminal:

\$ ssh username@computer connects to the remote computer specified, followed by entering a password.

Ctrl-D terminates the remote login and returns to our shell.

#### 5.1.2 *How to generate SSH key*

1 - In the Terminal:

\$ ssh-keygen -t rsa -b 4096 -C "my\_emailaddress.com" this creates a new ssh key.

2 - Generating public/private rsa key pair this lines appears next.

3 - Enter a file in which to save the key (/User/Sistina/.ssh/id\_rsa. press Enter, this accepts the default file location.

4 - You will see the key's randomart image.

```
wmi2076:~ Sistina$ ssh-keygen -b 4096 -N ""
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/Sistina/.ssh/id_rsa):
Created directory '/Users/Sistina/.ssh'.
Your identification has been saved in /Users/Sistina/.ssh/id_rsa.
Your public key has been saved in /Users/Sistina/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:X/oTG0rcfduoZ/GhbRtNRVo00XarVT/CNhh8Idm28ks
Sistina@wmi2076.infektionsbio.uni-wuerzburg.de
The key's randomart image is:
+---[RSA 4096]-----+
|      o.+o+|
|      .*=Xo|
|      +.=oB|
|      . + =o|
|      S. o. * o|
|      ...O...E.|
|      o .o. .+++|
|      o ..o.+*o|
|      o*+oo|
+---[SHA256]-----+
```

5 - In the Terminal:

\$ cd .ssh/ this goes into the directory of SSH.

6 - In the Terminal:

\$ ls you should see id\_rsa and id\_rsa.pub as listed.

7 - type: \$cat \* it will list the content of Private key and the public key.

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQACQc+cla/jceleZPv5wpiZfM0je/N4+mh98nY/u7AMN3fCE8+sta
QJ1o261u7f0UF2X6jW/Hviyp46UD8uDWfMz4nzRJHzpwrQD+o8xg6ECMSM1l1EkTExqplR5z2A0F12D
Vd2eKq435xcHiclkfm2K4RY3C6Xf51ZXBy0Wm74e1nm9Q380wofs+t4U0EoZxeCnwLx4+oYFggE0UWW
XChlk5PfQxIjzwf+1R1mzvgnmH7DF9AyMoeKehzFVdpv945080MtHArqjJ1vVixJLQGEQrFqALfd8Gc
GvwUaU4tpQW3jVw+0rrQGaaamaJCncmbjkydMibZ1++/Qf3w0z/ofaYeLvKcyjBxtISPOfsmhKbquxh
YDuB+6x2hcEWf/d9CmiHK8CFiFpX4TfUItxdvsMgWB4E7DVytLT1mWdLquQPhbyokc7ED5/BYwb7qL7
lcEFFpSpF1mJFwDA9xBMPYAM9SHmRc/tdbMbtcUd4h9ops5izt50F4Asq4Fq3VXdBFU+u3uRsDYmmTd
K9KovHoGtxsIJkHod0dMiPGJeOp1scNffNziBC2JFDWvgp1H4i2F7n1C+oHIyHLPWHoNwEpTKEKNUTj
kRKSjaBlg4FnyM2bulfSc5Ag90BR32U7rWpX3EuG0g1NFVKDxJd00TiiFnPUYA1iRBRnb5Shf4nnXL
Shw== Sistina@wmi2076.infektionsbio.uni-wuerzburg.de
```

## 6 Git

Git is a version control system for tracking changes in my source code history. Github is a hosting service for Git repositories. So they are not the same thing: Git the tool, Github the service projects that use Git.

Once uploaded onto Github, the file can be edited by selected collaborators, creating a collaboration platform among multiple users.

Git thinks of its data like a series of snapshots of a miniature filesystem. Every time we commit, Git takes a picture of the file at that moment, and stores that snapshot. If the file is not further modified, Git doesn't store the file again, just a link to the previous identical file it has stored. Git thinks of its data like a **stream of snapshots**.

Git has three main states that the file can reside in: *committed*, *modified*, and *staged*:

- committed means that the data is stored in my local database.
- modified means that I have changed the file but have not committed it to my database yet.
- staged means that you have marked a modified file in its current version to go into the next commit snapshot.

This leads to the three main sections of a Git project: the Git directory, the working tree, and the staging area.

- Git directory is where Git stores the metadata and object database of my project. This is the most important part of Git, and it is what is copied when I *clone* a repository from another computer.
- working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk so I could modify or continue to work on.
- staging area is a file, generally contained in the Git directory, that stores information about what will go into my next commit.

The basic Git workflow goes as follows:

1. I modify files in the working tree.
2. I selectively stage just those changes I want to be part of the next commit, which adds only those changes to the staging area.
3. I commit, which takes the files as they are in the staging area and stores that snapshot permanently to the Git directory.

Therefore, if a version of file is in the Git directory, it is committed. If it was modified and added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

Git directory is created by `$git init`, and it is folder that contains all the informations needed for git to work. The `.git` directory has file names like HEAD, branches, config, description, hooks (subfolders: pre-commit.sample, pre-push.sample), info (subfolder: exclude), objects (subfolders: info, pack), refs (subfolders: heads, tags);

To set up GitHub, the first step is to sign up for an account on GitHub.com. Then install Git on local computer. Github won't work if Git is not installed. Git uses the Terminal to access it.

## 6.1 Definitions

`version control` means it keeps "snapshots" of every point in time in the project's history, without being overwritten, as whenever we save in Microsoft Word file.

`repo` (for repository) is a virtual storage space of my project.

`commit` saves the state of the project, taking a "snapshot" of the repo at a point in time.

`branch` this allows multiple collaborators working on a project at the same time without Git getting them confused. There is a hierarchy: the main version is called the "master"; each user "branch off" to work on their own versions. Once it's done, the sub-branches "merge" with the "master".

## 6.2 Adding a new SSH key to Github

This step allows a secure and hussle-free authentication process whenever we connect to Github.

1 - From the Terminal:

`$ pbcopy < ~/.ssh/id_rsa.pub` copies the content of the SSH public key to the clipboard.

2 - From the github website, click "your profile photo", then click **Settings**.

3 - Click **SSH and GPG keys**.

4 - Click **New SSH key** or **Add SSH key**.

5 - Paste the key into the "Key" field.

6 - Click **Add SSH key**.

### 6.3 Making a public repository private

As a student, we can join GitHub Education and get an account for free. Once authorized, we could mark a repo to be private.

- 1 - Under my repository name, click Settings.
- 2 - Click Make Private and follow the instructions.

### 6.4 Commands

#### 6.4.1 How to configure Git

- 1 - Install Git (Xcode Command Line Tool for Mac).
- From the Terminal:
- 2 - `$ git --version` checks if git is present.
  - 3 - `git config --global user.name "Your Name Here"`
  - 4 - `git config --global user.email "YourAddress@mail.com"`

#### 6.4.2 How to create and save a repo

- 1 - Login to Github account, click on the + sign on the right top menu, select New repository.
- 2 - Give the repo a short, memorable name plus description.
- 3 - Select for public or private viewing.
- 4 - Click Create repository now we have created an online space for my project to live in. Also click Initialize this repository with a README.
- 5 - Click Clone or download so far, we have created a *remote* repository. By cloning the repository, this creates a *local* copy on my computer and sync between the two locations.
- 6 - Choose Clone with SSH and click the icon to copy the clone URL for the repository.
- 7 - On my computer: Place the file to be pushed together with .git in the same folder.
- 8 - From the Terminal:  
`$ git init` initializes a new Git repository. This is a one-time command during initial setup of a new repo.  
`$ git clone` and paste the URL copied from the clone and press Enter. For example:

```
Christines-MacBook-Pro:~ Sistina$ git clone
git@github.com:crisWendel/F1_Bioinformatics.git
Cloning into 'F1_Bioinformatics'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

- 9 - `$ git add F1.tex` F1.tex is an example of files in LaTeX. This command brings a new file to Git's attention. After adding a file, it is included in



Git's "snapshots" of the repo.

10 - `$ git commit -m` is the most important command. It takes a "snapshot" of the repo. This command is followed by "description", meaning what changes were made. For example:

```
Christines-MacBook-Pro:F1_Bioinformatics Sistina$ git add F1.tex
Christines-MacBook-Pro:F1_Bioinformatics Sistina$ git commit -m "pushing
protocol to github"
[master 92a84e0] pushing protocol to github
1 file changed, 292 insertions(+)
create mode 100644 F1.tex
```

11 - `$ git push` this makes my commits visible online on Github. I "push" the changes up to GitHub.

```
Christines-MacBook-Pro:F1_Bioinformatics Sistina$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 6.10 KiB | 6.10 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:crisWendel/F1_Bioinformatics.git
 cfb4b06..92a84e0 master -> master
```

Now the file F1.tex is stored in Github.

### 6.4.3 Viewing staged and unstaged changes

`$ git status` tells us if the file is committed or not. For example:

```
Christines-MacBook-Pro:F1_Bioinformatics Sistina$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   F1.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        F1.aux
        F1.bcf
        F1.log
        F1.pdf
        F1.run.xml
        F1.synctex.gz
        F1.tex.bak
        F1.toc
        git_3sections.png
        git_clone.png
        git_commit.png
        git_directory.png
        git_lifecycle.png
        git_push.png

no changes added to commit (use "git add" and/or "git commit -a")
```

"Changes not staged for commit" means that this file has been modified in the working directory but not yet staged. (To stage it, we need to run `$ git add`, which begins to track new files, to stage files.)

If we run `$ git add`, the message will be "Changes to be committed". Now

the file is staged and ready to be committed.

Note: Suppose I make some changes at this point, and then run `$ git status`. I will get both messages: "Changes to be committed" and "Changes not staged for commit". If I commit now, Git will save the version when I ran `$ git add`, but not the modifications I did after that. So the next step is to run again `$ git add` to stage the latest version of the file.

`$ git diff` very detail description of which modifications were made that are still unstaged. If the changes were staged (`$ git add`), then `git diff` will not give any output.

`$ git log` is a running record of commits and tells us who made the changes (the author of the commit).

#### **6.4.4 others**

`$ git help` shows common git commands.

`$ git rm` removes a file from Git by removing it from the staging area, and then commit. It also removes the file from the working directory, so it doesn't appear as an untracked file.

## 7 Exercises

The best way to learn bioinformatics is by solving computational problems related to molecular biology using Python. We can find such exercises in websites like [ROSALIND](#) or in [GitHub repositories](#) of Bioinformatics scientist Dr. Konrad Förstner .

### 7.1 ROSALIND

To find the exercises, go to [ROSALIND](#) and click Problems on the top menu. For example:

#### Exercise: Counting DNA nucleotides

How many A, C, T, G are in this string of letters?

```
Strings =
'ATTCAGATCGGCGTGATCGACTTCATCAGATCAGCACGTTCCCTCCCGATTGCCCCCTGA
TGACGAAACCTCTGGTAGACGTTGCCCCACAGGAAGGTGCCGGTTTGTTTCAGTAGAGAT
AAAGTTACGAGTCAGCCGTCCGCAAGAACCTTGGTGCAATTGCTGTATACATTAGGATCA
AGGTTATTAGTGCAGGATTAGAAGGCATCGAAGTTACCTCCTATTGATCGTCAGCCAAT
TCCTATGCTGTGCTACTCTGAGCAAGTTGTAATGGCTCCGTCCTAGGGCGGGGAAATGGC
AGCCACCGAGCACACATTGATCCGGTTGAATTAGGGGCGTTTGATTAGTCGAGGCGCTC
CCAGGGCTGGTGCGGGGTTACGGATAGTAGTCCCCCTTGGCAGTGCCAATGGGGATGGG
CGTTTCGGTGTATTCTGCAGTCGTCAGACTGAAAGAAGGCACAATCCCATTGCCGGCTG
CCGGTTCCTCTAAATGGCTCGTACCCCCAGAGGCCTACGGTTTGCCCGCACCTTGAGGGT
ACAATGAGTTGACTGCCACGGCAGCACGCGCACTTTCAAACCCAATGCTACACGCTGGCT
GTCACGGGGCAGCGATATAGAAAACGATTCGGGCGTGACACGAGGCCCCGATCTGGGAC
GTTAATCACAAATAAGTGTGTGTGTGGGTCAAAGGCTGACGGTAAATCCCATGTG
TACATTGCGGACAAGTAGAGGGGAGCCGCGGGTTGTTCCCTGGTGGCGGCAAGGCCTATTC
ATTTATTTACTGTTTGTGAATATTGCCTGGGGAAGTTTGACTATATTTTATATGGCTG
ACTTGTCGATAGGGGTCGAGGATAACGACCTCCATGTCCTGATAG'
```

Solution:

```
print (Strings.count('A')), print (Strings.count('C')), print (Strings.count('T')),
print (Strings.count('G'))
```

Output:

```
199
210
221
253
```

What if we want the output to list the numbers right after another?

Solution:

```
a = Strings.count('A')
b = Strings.count('C')
```

```
c = Strings.count('T')
d = Strings.count('G')
print (a, b, c, d)
```

Output: 199 210 221 253

### Exercise: Computing GC content

What is the GC content in the following DNA string?

```
String_s =
'CTACCCCTCCATTAATACCTCGCACCTCCTCCCTAGTCTTGGCCACATTGCACTGTAAC
CCGCGTGTAGGAGAAGAGTATTCCGAGTAGAGCGGTTGTCGACATGTCAAAGCTCTCCGG
ATTGATTTCTTCTTTGTTTCGCACTGTTTCGACGGCAGCTCGTATAGTGCAGTTAGAGAAG
TGGGATACTAAATTATCGGTGCAGGTACGCTCCGAATGTATGTTGGTTTCGTTAAACTCC
AATAGATGGCCGATCACCGCTTGTTCGGTCTCTGGCTTTTCGTGACAGGGGCGTGGTGACTA
ACCCGTACTACAATCATCGAGCACCGGAACAAATTCCTTAAGGGAGAAATTTTTTTAGTC
GCAACGAAGAAGCACTATTCTGCGAACGGTGGGTCCGTGAGTCCCTTGTCCACAGTATGC
AAGCGTCGGGTGCTGATGGATAATACAGCTCCTGGATGTTTACCAAGGACCTCCATTACA
TCTACCCAAAGAATGCCCATTCATTGTCAACAGTTTCACGCCGCTTGTGACGACCGTGTG
GAAATACGAGATTCCAGGCTTTGTCCGCTTTTGGCGTTAATTGGAGCACGATAGAGTTT
AGGAGCCTAGCTAGTGTGGAGCTTATGAGGCAATACATTTAGTGATTCAATTTGATCTA
GAGAGGGCAGTGACCGGTGGCACTCACAGAACGATCATATTGTCGAACCTACTTGCCAGA
GAATCCGCTTCACTATGGAGCGCTGCCAGGCACTCCCCGTAGAATGGTTGGAGTCTTTGA
GCTCAGTCGACATACAACCCCACTTAGTCCCGACAAGCGATGCAGCGTTAATAGACCACC
TAAGACGCGGGCCGTGCACGCGCATCTGCTGTAGGTAGAGGAGAATGCTACTCTGAACGG
CAGCTATTTGCGCAGTGTCCGGGGAGTGAATAAGTCGGTTCGACCACGTGCACCTTTCGA
ATAATCCTGACTGTACAGCCGGCTGGC'
```

Solution:

```
a = String_s.count('G')
b = String_s.count('C')
c = len(String_s)
d = a + b
print ((d/c) * 100)
```

Output:  
50.3553

## 7.2 Konrad's repo

Here, the exercises are about analyzing gff3 files. Gff3 (*generic feature format version 3, gene-finding format*) is a file to annotate gene and features of DNA, RNA and protein sequences. Format: in plain text. Advantages: data are separated by tabs, and if there were in an excel sheet (in nine columns). Download the gff file to do the exercises.

**Exercise: What is in the second column of the gff file?**

Solution:

```
for line in open (enter the gff file):  
  
    sep_line = line.split()  
    print(sep_line[1])
```

**Exercise: Count the word "genes" in gff file**

Solution:

```
for line in open (enter the gff file):  
    if line.startswith("#"):  
        continue  
    sep_line = line.split()  
    column2 = sep_line[2]  
    if column2.startswith("gene"):  
        print(column2.count("gene"))
```

**Exercise: Count the occurrence of all features (genes, CDS, rRNA, tRNA) in the gff file, using libraries**

Solution 1:

```
from collections import defaultdict  
gffdict = defaultdict(int)  
gff_file = open(enter gff file name)  
for line in gff_file:  
    if line.startswith("#"):  
        continue  
    sep_line = line.split("\t")  
    gff_key = sep_line[2]  
    gffdict[gff_key] = gffdict[gff_key] + 1  
gffdict
```

Output:

```

defaultdict(int,
    {'CDS': 5140,
     'RNase_P_RNA': 1,
     'SRP_RNA': 1,
     'antisense_RNA': 1,
     'direct_repeat': 3,
     'exon': 122,
     'gene': 5089,
     'ncRNA': 11,
     'pseudogene': 170,
     'rRNA': 22,
     'region': 4,
     'repeat_region': 17,
     'riboswitch': 7,
     'tRNA': 85,
     'tmRNA': 1})

```

Solution 2:

```
gffdict = {}
```

```
gff_file = open(enter gff file name)
```

```

for line in gff_file:
    if line.startswith("#"):
        continue
    sep_line = line.split("\t")
    gff_key = sep_line[2]

    if gff_key in gffdict:
        gffdict[gff_key] +=1
    else:
        gffdict[gff_key]=1
print(gffdict)

```

Output:

```

{'region': 4, 'gene': 5089, 'CDS': 5140, 'pseudogene': 170, 'riboswitch': 7, 'rRNA': 22, 'exon': 122, 'tRNA': 85, 'SRP_RNA': 1, 'ncRNA': 11, 'antisense_RNA': 1, 'tmRNA': 1, 'direct_repeat': 3, 'repeat_region': 17, 'RNase_P_RNA': 1}

```

## 8 References

<https://github.com/konrad>

<https://swcarpentry.github.io/shell-novice/>

<https://swcarpentry.github.io/git-novice/>

<https://www.codecademy.com>

<https://www.coursera.org/learn/python>

<https://www.coursera.org/learn/learn-to-program/home/info>

[https://en.wikipedia.org/w/index.php?title=Type\\_system&oldid=849096813](https://en.wikipedia.org/w/index.php?title=Type_system&oldid=849096813)

<https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

<https://medium.freecodecamp.org/understanding-git-for-real-by-exploring-the-git-directory-1e079c15b807>

<http://rosalind.info/problems/dna/>

<https://www.quora.com>

<https://en.wikipedia.org>

## 9 Acknowledgements

I would like to thank the lab for their support and making Bioinformatics-learning fun and a bit less intimidating:

Dr. Konrad Förstner  
Vivian Monzón  
Mandela Fasemore  
Till Sauerwein  
Silvia Di Giorgio  
Dr. Thorsten Bischler  
Dr. Kristina Döring  
Dr. Panagiota Arampatzi  
Dr. Richa Bharti

Core Unit Systemmedizin, Institute für Molekulare Infektionsbiologie (IMIB),  
Josef-Schneider-Str. 2, Bau D15, Würzburg.