Programming for Life Scientists

Course Plan

February, 2025

Contents

Preface

The purpose of the course 'Programming for Life Scientists' is to provide its participants with a general understanding of the concepts in programming, utility of the UNIX terminal environment, shell scripting, and Python programming language, as well as the utility of Large Language Models and AI-assisted program solving. The course is designed for life scientists with mock or real-world datasets and problems that convey typical characteristics of what researchers often encounter in data-enabled era of life sciences, with the intention to help them develop dry-lab skills the way they developed wet-lab skills.

This document includes everything you need to understand the contents and the requirements of this course. Please pay particular attention to the 'Course Responsibilites and Grading' section at the very end of this document if you are interested in getting a grade for this course.

Here are a few important details of the course:

- **Course number**: 5.13.622 (WiSe24/25)
- **Dates:** February 3 to February 14, 2025, Monday to Friday, 09:00 to 17:00
- **Format:** Lectures, discussions, hands-on exercises, and assignments

Faculty and Communication

The course will be directed by Prof. Dr. A. Murat Eren (Meren) and Prof. Dr. Sarahi Garcia. While the lectures and exercises will be primarily delivered by Meren, additional experts may take part in the delivery of various sections. The following table lists individuals who will be directly or indirectly involved in the course, and their contact information:

Name	Role	Expertise	Contact information
Meren	Professor	Microbial Ecology, Computer Science	meren@hifmb.de
Sarahi	Professor	Microbiology, Microbial Ecology	sarahi.garcia@uni-oldenburg.de
Florian Trigodet	Senior Scientist	Microbiology, Bioinformatics	florian.trigodet@hifmb.de
Iva Veseli	Postdoc	Microbial Ecology, Computer Science	iva.veseli@hifmb.de

Throughout the course (and beyond) you can reach out via email with any question to Meren, who should be your first contact for anything related to the course activities unless specified otherwise.

Technical Recommendations

Here are a few recommendations that will help you throughout this course

Shell Environment

Throughout this course we will use a terminal that gives access to the Unix shell. If you are using Linux or Mac OSX, you have native access to a reasonable shell. If you are using windows, you will need to install Linux for Windows to access to a reasonable shell. We will spend a lot of time learning about these, but here are a few resources if you would like to take a look:

- Beginner's Guide to the BASH (a video introduction to the command line environment – although Joe Collins is talking about Linux, the topics are relevant to anyone who uses a command line environment and Meren strongly recommends everyone to take a look).
- Learning the Shell (a chapter from the open book "*The Linux Command Line*" by William Shotts – Meren highly recommends).

Code Editors

Throughout this course you will be writing and editing lots of code. Writing good code comfortably requires a good and comfortable environment designed to write code, which excludes editors such as Microsoft Word or NotePad that are more appropriate for daily writing tasks. While Meren exclusively uses vim for his everyday coding tasks, students are encouraged to consider using a **graphical code editor with syntax highlighting** that supports multiple programming languages. Below are a few recommended options based on operating system:

Windows

- VS Code – Feature-rich, Git integration, extensions for Python & Bash
- Notepad++ – Lightweight, supports many programming languages

Mac

- VS Code – Great cross-platform editor with extensions
- Sublime Text – Fast, minimal, and powerful

Linux

- VS Code – Available as a snap package
- Geany – Lightweight and efficient

In addition, it may be a good idea to familiarize yourselves with nano, a very simple terminal-baseed text editor.

Version Management System

The delivery of assignments will reqire you to use Git version management system, and have an account on GitHub, which is an online service built to store Git repositories in the cloud. We will discuss both git and GitHub in detail, but you should open an account on GitHub unless you already have one as soon as you read these sentences.

Course Schedule Overview

The course is designed to be delivered in two weeks. Which will be a sprint rather than a marathon.

The **first week** will offer insights into foundations of programming, UNIX Tools, BASH, and AI-assisted problem solving, and the **second week** will cover a quick introduction to Python, and a ton of exercises and a few assignments.

Here is a quick day-by-day breakdown of the course schedule:

W	D	Topic	Morning Session (9 AM - 12 PM)	Afternoon Session (1 PM - 5 PM)
1	1	**Introduction to Programming**	Overview of programming languages	UNIX basics & file navigation
1	2	**Essential UNIX Tools (Part 1)**	File manipulation (`cat`, `less`, `cp`, `mv`, etc.)	Searching files (`grep`, `find`, `awk`)
1	3	**Essential UNIX Tools (Part 2)**	Pipes and redirection (`|`, `>`, `>>`)	Hands-on: Processing biological data
1	4	**Introduction to Git & GitHub**	Basics of Git (`init`, `clone`, `commit`, `push`, etc.)	Hands-on: Creating repositories & pushing scripts
1	5	**Bash Scripting & AI Tools for Coding**	Bash scripting basics (`for`, `while`, `if`, `case`)	Using AI tools like ChatGPT for programming
1	1	**Python Basics**	Variables, data types, operators	Control flow (`if`, `for`, `while`)
1	2	**Working with Files & Data**	Reading/writing files (FASTA, CSV)	Parsing FASTA files
1	3	**Data Analysis with Pandas & NumPy**	DataFrames, array operations	Analyzing gene expression data
1	4	**Bioinformatics with Biopython**	Sequence objects, BLAST queries	Hands-on: BLAST searches
1	5	**Final Project & Presentations**	Working on small biological dataset	Presentations & discussions

As you can see, this is not a conventional course structure. But this course structure will likely work for most participants since it has multiple qualities:

- **Balanced Approach**. Rather than focusin on a single topic in great depth, the course structure covers fundamental programming, UNIX tools, Git, BASH scripting, and Python in a structured way.
- **AI Tools Introduction** – Placed at an optimal time to help participants learn efficient debugging and script generation with ChatGPT before diving into Python.
- **Real-World Applications** – Exercises and assignments resemble real-world problems researchers often run into during their day-to-day workflows rather than unrelatable hypothetical programming tasks.
- **Version Control & Reproducibility** – Introduction to Git and GitHub ensures that participants develop good coding habits early in their journey.
- **Project Focus**. Participants get to apply everything they have learned to solve an actual problem.

Assignments

TBD

Course Content

Day 1, Monday (03/02/25)

The day of course logistics, discussions over microbial diversity, and a very general introduction to programming languages and what they are good for.

09:00 - 10:00: Course Logistics

Discussion over what will happen throughout the next two weeks. A great time to take a look at the course syllabus online together.

10:00 - 12:00: Installation check

Making sure that everyone have their computer properly setup for the next two weeks. This includes having access to a terminal environment with BASH, being able to run `git`, and having a username on GitHub.

13:00 - 17:00: Essential Unix Programs

The purpose of this section is to familiarize you with some of the features of the shell environment (such as BASH) along with some of the most commonly used programs that help you navigate through files, directories, and their contents.

We will discuss these in a few individual sections, but as we go through some examples, you will realize that an effective use of the shell environment requires a dynamic orchestration of everything.

Navigating files and directoris `ls` – List directory contents

Lists files and directories in the current directory.

```
ls        # List files in the current directory
ls -l     # Long listing format (permissions, owner, size, date)
ls -a     # Show hidden files
ls -lh    # Human-readable file sizes
```

`cd` – Change directory

Moves between directories.

```
cd /home/user/Documents  # Change to Documents folder
cd ..                    # Go one level up
cd /                     # Go to root directory
```

`pwd` – Print working directory

Displays the absolute path of the current directory.

```
pwd  # Show full path of the current directory
```

`mkdir` – Create a new directory

Creates a new empty directory.

```
mkdir my_folder       # Create a directory named my_folder
mkdir -p parent/child # Create nested directories
```

`rmdir` – Remove an empty directory

Removes an empty directory.

```
rmdir my_folder  # Deletes my_folder if it's empty
```

`rm` – Remove files or directories

Deletes files or directories.

```
rm file.txt         # Delete a file
rm -r my_folder     # Delete a directory and its contents
rm -rf my_folder    # Force delete (dangerous)
```

`cp` – Copy files and directories

Copies files and folders.

```
cp file1.txt file2.txt         # Copy file1.txt to file2.txt
cp -r dir1 dir2                # Copy directory dir1 to dir2
cp file1.txt file2.txt backup/ # Copy both files into the backup folder
```

`mv` – Move or rename files

Moves or renames files and directories.

```
mv oldname.txt newname.txt   # Rename a file
mv file.txt my_folder/       # Move a file into my_folder
mv folder1 folder2           # Rename folder1 to folder2
```

`touch` – Create an empty file

Creates a new empty file.

```
touch newfile.txt  # Create an empty file called newfile.txt
```

`find` – Search for files and directories

Finds files in a directory hierarchy.

```
find /home -name "document.txt"  # Search for document.txt in /home
find /var -size +100M            # Find files larger than 100MB in /var
find . -type f -name "*.log"     # Find all .log files in current directory
```

The last command uses *, a special character (so-called 'wildcard') that is extremely useful to target multiple files that match to a particular pattern. This is not the only special character, and most shells will process user input commands and interpret a series of special characters when they are found. Before we continue with more fun programs, let's take a look at a list of commonly used special charaters first.

Special Characters `$` – Variable substitution and command substitution

Used to reference variables and execute commands.

```
echo $HOME       # Prints the home directory
echo $(date)     # Runs the date command and prints the result
```

`#` – Comment

Everything after `#` on a line is ignored by Bash.

```
# This is a comment
echo "Hello World"  # Prints Hello World
```

`*` – Wildcard (matches multiple characters)

Matches all files and directories in a given location.

```
ls *.txt    # Lists all files ending in .txt
```

`?` – Single character wildcard

Matches any single character.

```
ls file??.txt  # Matches file01.txt, file02.txt, etc, but not fileX.txt
```

`.` – Current directory

Refers to the current directory.

```
ls .          # List files in the current directory
./script.sh   # Execute script.sh in the current directory
```

`..` – Parent directory

Refers to the directory one level above the current directory.

```
cd ..   # Move to the parent directory
```

`~` – Home directory

Represents the current user's home directory.

```
cd ~         # Go to the home directory
ls ~/Docs    # List files in the "Docs" directory inside home
```

`>` – Redirect output (overwrite)

Sends output to a file (overwrites if file exists).

```
echo "Hello" > file.txt  # Writes "Hello" into file.txt (overwrites)
```

`>>` – Append output to a file

Appends output to the end of a file instead of overwriting.

```
echo "Hello" >> file.txt  # Writes "Hello" into file.txt (appends)
```

`<` – Input redirection

Reads input from a file.

```
sort < unsorted.txt   # Reads unsorted.txt as input for the sort command
```

`&&` – Logical AND (run second command only if first succeeds)

Runs the second command **only if the first one succeeds**.

```
mkdir newdir && cd newdir  # Creates and moves into newdir if successful
```

`||` – Logical OR (run second command if first fails)

Runs the second command **only if the first one fails**.

```
mkdir mydir || echo "Directory creation failed"  # Prints message if mkdir fails
```

`;` – Command separator

Allows multiple commands on the same line.

```
echo "Hello"; echo "World"  # Prints Hello then World
```

`$?` – Exit status of the last command Stores the exit status of the most recent command.

```
ls /notexist
echo $?   # Prints the exit status (non-zero means failure)
```

`"$@"` – All command-line arguments (individually)

Expands arguments as separate words.

```
#!/bin/bash
echo "Arguments: $@"
```

`"$*"` – All command-line arguments (single string)

Expands arguments as a single string.

```
#!/bin/bash
echo "Arguments: $*"
```

`\` – Escape special characters

Prevents special interpretation of characters.

```
echo "This is a quote: "Hello""  # Prints: This is a quote: "Hello"
echo \$HOME  # Prints $HOME instead of its value
```

`'` (Single Quotes) – Preserve literal text

Prevents expansion of variables and special characters.

```
echo '$HOME'  # Prints: $HOME (without expanding)
```

`"` (Double Quotes) – Allow variable expansion

Expands variables inside.

```
echo "Home: $HOME"  # Prints: Home: /home/user
```

`!$` – The last argument from the last command

Repeats arguments from the previous command.

```
ls -l /etc
echo !$  # Expands to: echo /etc
```

`>` and `2>` – Redirect standard and error output

Redirects normal output (`>`) and error output (`2>`).

```
ls > output.txt    # Saves normal output
ls nonexistent 2> error.txt  # Saves error output
ls nonexist &> all.txt  # Redirects both normal and error output
```

`|` – Pipe (send output of one command to another)

This one implements one of the most important and powerful concept in the shell environment: it passes the output of one command as input to another.

```
ls | grep "file"   # List files and filter for those containing "file"
ps aux | grep ssh  # Show running processes related to SSH
```

There are also many things that are great for working with files. Here are a few:

`cat` – Display file contents

Concatenates and displays file content.

```
cat file.txt         # Show full file content
cat file1.txt file2.txt > merged.txt  # Merge two files into one
```

`tac` – Display file in reverse order

Prints a file **from bottom to top**.

```
tac file.txt  # Show file.txt in reverse order
```

`less` – View file contents page by page

Allows **scrolling** through large files.

```
less largefile.txt   # Open a large file for viewing
```

You can press `q` anytime to quit and go back to your terminal, or type `/search_term` to find a term.

`more` – View file (similar to `less`)

Similar to `less`, but **only allows forward movement**.

```
more file.txt  # View file.txt one page at a time
```

`head` – Show the first few lines of a file

Displays the first 10 lines by default.

```
head file.txt       # Show the first 10 lines
head -n 5 file.txt  # Show the first 5 lines
```

`tail` – Show the last few lines of a file

Displays the last 10 lines by default.

```
tail file.txt        # Show the last 10 lines
tail -n 5 file.txt   # Show the last 5 lines
tail -f log.txt      # Continuously show new lines (useful for logs)
```

`grep` – Search for text in a file

Finds lines containing a specific pattern.

```
grep "error" logfile.txt       # Find lines containing "error"
grep -i "warning" logfile.txt  # Case-insensitive search
grep -r "function" /home/code  # Search in all files under /home/code
```

`sed` – Stream editor (modify file content)

Used for **text replacement** and processing.

```
sed 's/old/new/g' file.txt   # Replace "old" with "new" in file.txt
sed -i 's/foo/bar/g' file.txt # Edit file in place
```

`sort` – Sort lines in a file

Sorts text **alphabetically** or **numerically**.

```
sort names.txt         # Sort lines alphabetically
sort -r names.txt      # Reverse order sort
sort -n numbers.txt    # Sort numerically
sort -u names.txt      # Remove duplicate lines
```

`uniq` – Remove duplicate lines from sorted text

Filters out repeated lines in a file.

```
sort names.txt | uniq    # Remove duplicates after sorting
uniq -c names.txt        # Show duplicate counts
```

`wc` – Count words, lines, and characters in a file

Displays **line count, word count, and byte size**.

```
wc file.txt     # Show lines, words, and characters
wc -l file.txt  # Show only the number of lines
wc -w file.txt  # Show only the number of words
wc -c file.txt  # Show only the number of bytes
```

`diff` – Compare two files line by line

Finds differences between two files.

```
diff file1.txt file2.txt    # Show line-by-line differences
diff -y file1.txt file2.txt # Show side-by-side comparison
```

`awk` – Pattern scanning & processing

AWK is a data-driven language that excels in text processing, data extraction, and reporting, and it is one of the most amazing little tools you will find in the UNIX shell environment (fun fact, AWK is not a meaningful acronym since it simply comes from the names of the developers: Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan :)).

In a nutshell, AWK scans a file line by line, splits input lines into multiple fields based on a separator, enables pattern-based filtering of lines, allows user the perform actions on matching lines, and help produce highly formatted reports.

It is really difficult to demonstrate the utility of AWK without a few examples, so I put together the following file of all chancellors of Germany where the columns indicate (1) the name of the chancellor, (2) their education, (3) the age at which they became a chancellor of Germany, (4) the number of year they served at this position, and (5) the year they assumed this position. You can save the contents of this file as `german_chancellors.txt` in your working directory, and follow the examples below:

```
Konrad_Adenauer   Law          73   14   1876
Ludwig_Erhard     Economics    65   3    1897
Kurt_Georg_Kiesinger  Law      62   3    1904
Willy_Brandt      History      55   5    1913
Walter_Scheel     Law          54   1    1919
Helmut_Schmidt    Economics    56   8    1918
Helmut_Kohl       History      52   16   1930
Gerhard_Schröder  Law          54   7    1944
Angela_Merkel     Physics      51   16   1954
Olaf_Scholz       Law          63   Ongoing  1958
```

Print the name, age when becoming chancellor, and birth year:

```
awk '{ print $1, "became chancellor at age", $3, "and was born in", $5 }' german_chancellors.txt
```

Find chancellors born before 1920:

```
awk '$5 < 1920 { print $1, "was born in", $5 }' german_chancellors.txt
```

Calculate each chancellor's age today (assuming we are still in 2025 by the time you're seeing this):

```
awk '{print $1, "would be", 2025 - $5, "years old today" }' german_chancellors.txt
```

Find chancellors born after women gained voting rights in Germany (after 1918):

```
awk '$5 > 1918 { print $1, "was born in", $5 }' german_chancellors.txt
```

Categorize chancellors by birth century:

```
awk '{
    if ($5 < 1900)
        print $1, "- 19th century born";
    else if ($5 >= 1900 && $5 < 2000)
        print $1, "- 20th century born";
    else
        print $1, "- 21st century born";
}' german_chancellors.txt
```

Find the average age at which chancellors took office:

```
awk '{ sum_age += $3; count++ } END { print "Average age when becoming chancellor:", sum_age/count }' german_cha
```

Print the first letter of the first name and last name of each chancelleor who studied law:

```
awk '{if ($2=="Law") print $0}' german_chancellors.txt | awk '{print $1}' | awk 'BEGIN{FS="_"}{print(substr($1,
```

Course Responsibilities

The evaluation in this course will be based on **five parts** of a portfolio that we can divide into **two major components**:

- The first major component is '**class citizenship**' emails, described below in the section **Part I** (20% of your grade).
- The second major component is '**programming exercises**', described below in sections for **Part II, III, IV, and V** (80% of your grade).

Part I (20% of your grade) – Class Citizenship

Class citizenship emails **will track your attendance and engagement to the course** and will help the course director to have an overall understanding of the evolution of the course.

The class citizenship demands every participant to send a class citizenship email to meren@hifmb.de and sarahi.garcia@uol.de at the end of each day (8 emails in total). The class citizenship email must be composed of two parts:

1. A brief summary of the main concepts discussed during the day, interpreted by the attendee in their own words.
2. A short question that is relevant to a concept or idea discussed during the lecture.

The last 10 minutes of every course day will be dedicated to writing the class citizenship emails, therefore the attendees will leave the class without having to remember doing it later. **The class citizenship emails that are sent after the end of the class will not be taken into consideration as a mark of attendance**.

The title of the class citizenship email **must follow this pattern word-by-word** where you will need to replace DD/MM/YY with the date, month, and the year (despite the simplicity of this request many students have failed to follow these instructions, so you are our last hope):

> PFLS Citizenship: DD/MM/YY

The best class citizenship emails are those that are brief, genuine, and insightful. In an ideal world the emails should be no less than 50 words, and no more than 150 words. Please do not send notes you take throughout the class – your notes are for you, not for us. You should use the last 10 minutes of the day to gather your thoughts, and come up with a summary of what you can remember.

Here is an example class citizenship email:

> Summary: Today we discussed what is phylogenomics, how phylogenomic trees are built, and why single-copy core genes are suitable for building phylogenomics trees. We also discussed the relationship between phylogenetics, phylogenomics, and pangenomics with respect to the fraction of genome used and the evolutionary distance that they can cover.
>
> Question: Since phylogenomics and pangenomics are both useful for inferring evolutionary distances, it seems to me that integrating both methods in a systematic way would yield a more reliable tree. But it looks like the field only uses phylogenomics and pangenomics separately, is there a reason for that?

Part II, III, and IV (30% of your grade)

This part will be composed of three mini programming exercises that you will have to implement and return. Each programming exercise will provide you with explicit instructions regarding the nature of the data and question, and what the program is expected to achieve. You will use your learnings in the course to complete the programming tasks and submit the resulting source code.

Part V (50% of your grade)

This part will be composed of a single large programming exercise that will require you to orchestrate multiple programming tools and languages.

Grading

The grading scale for this module is as follows:

Grade	Threshold
1.0	95%
1.3	90%
1.7	85%
2.0	80%
2.3	75%
2.7	70%
3.0	65%
3.3	60%
3.7	55%
4.0	50%