

B1 Scientific Coding

3 Lectures Michaelmas Term 2024

Jack Umenberger

1 Coding practical

jack.umenberger@eng.ox.ac.uk

Notice. This is a new course, running for the first time (MT24). All feedback in appreciated.

Contents

0	About this course	2
0.1	Learning outcomes	2
0.2	Which videos to watch	2
0.3	How to use these notes	5
1	Integrated Development Environments	6
1.1	The 'what?' and 'why?' of IDEs	6
1.2	VS Code: download and extensions	6
1.3	VS Code: Jupyter Notebooks	6
1.4	VS Code: debugging	7
2	Version control with Git and GitHub	8
2.1	The 'what?' and 'why?' of version control	8
2.2	Git'ing started and basic workflow	8
2.3	Staging and committing	8
2.4	Checking out earlier versions of code	10
2.5	Branching and fast-forward merges	10
2.6	Merge conflicts	11
2.7	Introduction to GitHub	12
2.8	Publishing and pushing	12

2.9	Cloning and sharing	13
2.10	Fetching and pulling	13
2.11	Remote branches	13
2.12	GitHub Issues and Pull Requests	13
3	Coding best practice	15
3.1	What is 'good code'?	15
3.2	Introduction to data types	15
3.3	Common data types	15
3.4	Operations on data and the effect of data type	16
3.5	Range and precision in data types	16
3.6	Functions: the basics	17
3.7	Considerations when writing functions	17
3.8	Introduction to objects and classes	18
3.9	Modules, packages, and libraries	19
3.10	Modularity: 'what?', 'why?', and 'how?'	19
3.11	Comments, documentation, and style	20
A	Command Line Interface	21
A.1	The 'what?' and 'why?' of CLI	21
A.2	Basic commands	21
A.3	Shells and scripts	22
A.4	Aliases and functions	22
B	Virtual Environments	24
B.1	The 'what?' and 'why?' of virtual environments	24
B.2	Python's inbuilt tools: pip and venv	24
B.3	Conda, Miniconda, and Ananconda	25

0 About this course

0.1 Learning outcomes

The learning outcomes for this course, as specified in the syllabus, are as follows:

1. Be able to explain example IDE (e.g. VS Code) and how it can be used to run code in multiple languages and aid debugging.
2. Understand the basic use and capabilities of Integrated Development Environments.
3. Understand GitHub and approaches to collaborative coding, as well as integration in VS Code.
4. Understand why modularity is important and be able to give examples of good practice.
5. Be able to discuss code structures and data types.
6. Understand the importance of commenting and documentation.
7. Gain an understanding of AI-based coding assistants such as GPT4 and Github Co-pilot along with usage best-practices.
8. Understand how scientific coding principles can be applied to solve simple problems by applying the above concepts to a detailed case-study.

0.2 Which videos to watch

The following subset of videos attempt to cover the learning outcomes as quickly as possible:

1. **The 'what?' and 'why?' of IDEs** (Video 1.1). This video will introduce (most) of the key features that Integrated Development Environments (IDEs) offer for software development. We'll take a brief tour of all these features (including AI coding assistants like GitHub Copilot) in the VS Code IDE.

2. **VS Code: debugging** (Video 1.4). This video will introduce the process of debugging, i.e. finding and fixing errors in your code. We'll go through a simple but thorough case study, using the builtin VS Code debugging tools.
3. **Git'ing started and basic workflow** (Video 2.2). This video will get you started with Git, allowing you to install Git (if you don't already have it) and create your first repo. It will also take you through the basic workflow of tracking, staging, and committing changes to your code.
4. **Staging and committing** (Video 2.3). This video will walk you through the process of staging and committing files, using VS Code and the terminal.
5. **Branching and fast-forward merges** (Video 2.5). This video will introduce you to the essential concepts of a branching and merging, and walk you through how to do this in VS Code.
6. **Introduction to GitHub** (Video 2.7). This video will introduce you to GitHub, which is related to but distinct from Git. You'll also create your own GitHub account if you don't have one.
7. **Publishing and pushing** (Video 2.8). This video will walk you through the process of integrating GitHub and VS Code, to help you 'push' your code from your local computer to the GitHub servers.
8. **Cloning and sharing** (Video 2.8). This video will walk you through cloning (downloading) code from GitHub, and adding collaborators to your GitHub projects.
9. **Fetching and pulling** (Video 2.10). This video will walk you through the process of downloading (fetching) and code from GitHub, and incorporating (pulling) it into your local codebase.
10. **Operations on data and the effect of data type** (Video 3.4). This video will introduce you to key data type concepts (static vs dynamic typing, strong vs weak typing, mutable vs immutable types) that determine how different data types respond to operations.

11. **Introduction to objects and classes** (Video 3.8). This video provides a conceptual introduction to Object-Oriented Programming, explaining the benefits of using objects/classes for structuring your code.
12. **Modularity: ‘what?’, ‘why?’, and ‘how?’** (Video 3.10). Modularity is at the core of what it means to write good code. This video will explain what it is, why it’s so important, and how you can write modular code.
13. **Comments, documentation, and style** (Video 3.11). This video explains why coding style is important for writing readable code, and for getting the most out of AI coding assistants. We also address the question of whether comments are really necessary.
14. **Writing technical reports**. This is a bonus video that provides a brief ‘how to’ guide for writing technical reports. It’s not directly related to ‘Scientific Coding’, but will hopefully be useful when writing the reports for your ‘coding practical’ assignments (as well as other technical reports throughout your education/career).

The ‘recommended videos’ are available on Canvas. All videos for the course are available in the following OneDrive folder.

Why are there so many extra videos? This course has been designed to be accessible to people with a diverse range of programming backgrounds, including those with very little programming experience. If you don’t know an array from a list, or you’re still a bit unsure about functions, don’t worry - this course is for you! The recommended videos in Section 0.2 above (attempt to) cover the learning outcomes as concisely as possible; however, if you find that these videos contain unfamiliar concepts, there’s a good chance that concept has been covered in an earlier video. In fact, if you don’t feel very comfortable with programming, you may wish to begin by watching all the videos in Section 3, which starts from the basics (variables, data types, functions, etc). You’ll be up to speed in no time!

What's the deal with the appendices? I asked current third year students what they know now that they wish they'd learned at the beginning of second year. The same three topics came up often: Git, virtual environments, and 'the terminal'. Appendix A covers the basics of the Command Line Interface (CLI), a.k.a. the terminal. The basic goal of this section is to make the terminal less intimidating, and to persuade you that CLI is a useful complement to (not replacement for) the familiar Graphical User Interface (GUI). You'll even get to write your own bash scripts! Appendix B covers virtual environments, both conceptually and practically (with Python's inbuilt tools and Conda). In my opinion, **using virtual environments is an essential part of software development best practice**. Although not mandated by the learning outcomes, I'd recommended watching Appendix B if you don't already feel comfortable using virtual environments (at least in Python).

0.3 How to use these notes

These notes are quite different to the ones you usually get, as this course covers material that is more practical than theoretical. As such, it's the videos, rather than the notes, that are intended to teach you the material. (It's probably nicer to watch someone explain how to resolve a Git merge conflict with VS Code, rather than read about it). Instead, these notes are intended to serve two purposes:

Helping you decide which videos to watch Each video has a corresponding subsection, and each subsection explains what you'll learn in each video. If you already know what the video is trying to teach, you should feel free to skip the video.

Helping you follow along with the videos Where appropriate, there is additional content to help you keep track of the operations that have been performed in the video. For example, when staging and committing in Git from the terminal, Section 2.3 lists the commands to perform those actions.

1 Integrated Development Environments

1.1 The 'what?' and 'why?' of IDEs

Learning outcomes After watching this video you should be able to:

- list (though not yet fully understand) the key features, tools, and capabilities of Integrated Development Environments (IDEs), including text editors, run/debug
- understand the benefits of using an 'integrated' development environment compared to a non-integrated development environment (i.e. a collection of tools that are not configured to interactive with one another within the same software application);
- understand where to find all of the key development tools in the Visual Studio (VS) Code IDE;
- avoid getting into pointless arguments about whether a given text editor is a 'true' IDE or not.

Consider skipping if . . . you already use an IDE that allows you to develop software in multiple languages.

1.2 VS Code: download and extensions

Learning outcomes After watching this video you should be able to:

- download and install VS Code;
- add extensions to VS Code to make it more useful for developing software in your preferred language(s), e.g. Python.

Consider skipping if . . . you are already familiar with extensions in VS Code.

1.3 VS Code: Jupyter Notebooks

Learning outcomes After watching this video you should be able to:

- use Jupyter Notebooks within the VS Code IDE;
- understand how using Jupyter notebooks provides additional interactivity when developing software;
- appreciate the capabilities of Jupyter Notebook for developing interactive documents combining text, code, and graphics.

Consider skipping if . . . you already know how to use Jupyter Notebooks in VS Code.

1.4 VS Code: debugging

Learning outcomes After watching this video you should be able to:

- understand the basic capabilities of a debugger;
- run and debug Python scripts in VS Code;
- begin to appreciate some basic debugging strategies.

Consider skipping if . . . you already feel comfortable using debuggers to debug code.

2 Version control with Git and GitHub

2.1 The ‘what?’ and ‘why?’ of version control

Learning outcomes After watching this video you should be able to:

- understand what version control is and why it is useful;
- understand how version control facilitates collaboration;
- understand that Git is an (open-source) software tool for version control, while GitHub is a company that provides server space for hosting projects that use Git for version control.

Consider skipping if . . . you have already encountered some form of version control, and understand the basic principles.

2.2 Git’ing started and basic workflow

Learning outcomes After watching this video you should be able to:

- install Git on your computer;
- understand what a Git repo is and how to create one;
- understand the basic Git workflow of tracking, staging, and committing files.

Consider skipping if . . . you already understand the basic concepts behind Git (details on actually performing operations in Git will come in subsequent videos).

2.3 Staging and committing

Learning outcomes After watching this video you should be able to:

- check the status of your Git repo, using VS Code and the Git CLI;
- stages changes in a Git repo, using VS Code and the Git CLI;

- commit staged changes in a Git repo via VS Code, using VS Code and the Git CLI;
- understand the role of the `.git/config` and how to modify it, directly and via the Git CLI;
- install and use the Git Graph extension in VS Code to visualize your Git repo as a graph of commits;
- view differences between files from different commits in VS Code.

Consider skipping if ... you already know how to stage and commit changes in a Git repo.

2.3.1 Follow along

Modify your Git config file via Git commands:

```
user@host:~$ git config user.name "<name>"
user@host:~$ git config user.email "<email>"
```

Check the status of your Git repo (which files are tracked, modified, etc) via:

```
user@host:~$ git status
```

Stage individual files:

```
user@host:~$ git add <filename>
```

Stage all changed files that are tracked, within the current directory:

```
user@host:~$ git add .
```

Stage all files, both tracked and untracked, from all directories within the repo:

```
user@host:~$ git add -A
```

Commit staged files:

```
user@host:~$ git commit -m "<Commit message>"
```

View your commits from the command line:

```
user@host:~$ git log --oneline
```

2.4 Checking out earlier versions of code

Learning outcomes After watching this video you should be able to:

- understand what is meant by the HEAD in a Git repo, as well as the purpose of a 'detached head' state;
- checkout previous commits;
- revert (undo) commits that you wish to disregard;
- change the default Git text editor from Vim to Nano.

Consider skipping if . . . you already know how to revert commits in Git.

2.4.1 Follow along

Return to a previous commit. You can find the commit hash (identity) using the Git graph:

```
user@host:~$ git checkout <commit_hash>
```

Change the Git text editor to Nano (if you want):

```
user@host:~$ git config core.editor "nano"
```

Revert ('delete') a commit:

```
user@host:~$ git revert <commit_hash>
```

2.5 Branching and fast-forward merges

Learning outcomes After watching this video you should be able to:

- understand what a branch is, and what role it plays in safely developing new features for a codebase;
- create a new branch, and switch between branches, using VS Code;

- understand what a fast-forward merge is;
- perform a (fast-forward) merge using VS Code.

Consider skipping if . . . you already know how to create and merge branches in Git.

2.6 Merge conflicts

Learning outcomes After watching this video you should be able to:

- understand what is meant by a ‘merge conflict’ and how they can arise;
- create a new branch, switch between branches, and merge branches using the command line;
- understand what is meant by a ‘three-way merge’;
- be able to resolve a merge conflict with VS Code’s merge editor;
- be able to abort a merge if it goes horribly wrong.

Consider skipping if . . . you already feel comfortable manually resolving merge conflicts.

2.6.1 Follow along

Create a new branch **without** switching to it:

```
user@host:~$ git branch <branch_name>
```

List the branches in the repo:

```
user@host:~$ git branch
```

Switch to an existing branch:

```
user@host:~$ git checkout <branch_to_switch_to>
```

Merge branch `feature_branch` into the **current** branch that you’re on:

```
user@host:~$ git merge <feature_branch>
```

Abort a Git merge if something goes wrong:

```
user@host:~$ git merge abort
```

2.7 Introduction to GitHub

Learning outcomes After watching this video you should be able to:

- understand what GitHub is and what role it plays in version control and collaboration;
- appreciate that GitHub provides both server space for hosting projects, as well as tools to support collaboration with Git;
- create your own GitHub account.

Consider skipping if . . . you already have (and use) a GitHub account.

2.8 Publishing and pushing

Learning outcomes After watching this video you should be able to:

- publish a local Git repo to GitHub using VS Code;
- integrate GitHub and VS Code; 🎓
- understand the distinction between local and remote repos;
- understand the `.git/config` file settings that specify your remote repo and username, and the importance of setting these values correctly;
- push commits to a remote repo on GitHub using VS Code;
- understand when it is and is not appropriate to push directly to your remote `main` branch.

Consider skipping if . . . you already feel comfortable pushing code to GitHub, and you always think very carefully before you push directly to `main`.

2.9 Cloning and sharing

Learning outcomes After watching this video you should be able to:

- clone (“download”) a Git repo from GitHub using VS Code;
- add collaborators to one of your Git repos hosted on GitHub.

Consider skipping if ... you already know how to clone repos.

2.10 Fetching and pulling

Learning outcomes After watching this video you should be able to:

- understand what the ‘fetch’ command does;
- perform a fetch using VS Code;
- checkout changes on a remote repo from VS Code, using the Git CLI;
- understand what a ‘pull’ command does;
- perform a pull using VS Code.

Consider skipping if ... you already know how to pull from a remote repo.

2.11 Remote branches

Learning outcomes After watching this video you should be able to:

- understand what is meant by a ‘remote branch’;
- push/pull arbitrary branches from GitHub.

Consider skipping if ... you already know the distinction between the local branches `new-feature` (say) and `origin/new-feature`.

2.12 GitHub Issues and Pull Requests

Learning outcomes After watching this video you should be able to:

- create a GitHub Issue and understand how Issues can be used during code development;
- understand the role that GitHub Pull Requests play in code development;
- create and 'land' a Pull Request on GitHub.

Consider skipping if . . . you've already landed a Pull Request on GitHub before.

3 Coding best practice

3.1 What is 'good code'?


Learning outcomes After watching this video you should be able to:

- give a persuasive argument that good code is code that: (i) works, and (ii) is easy to change;
- appreciate the qualities of good code that this definition implies (e.g. readability, maintainability, and performance);
- understand the distinction between verification and validation;
- understand that we write code for three audiences: 'computers', 'people', and 'Artificial Intelligence'.

Consider skipping if . . . you are already a professional software developer with a distinguished career of writing good code. (Or if you came to the first lecture).

3.2 Introduction to data types


Learning outcomes After watching this video you should be able to:

- understand the value of data types for thinking about data at different levels of abstraction;
- begin to appreciate why understanding data types is relevant to writing good code;
- “discuss data types” .

Consider skipping if . . . if you have some programming experience, and feel comfortable with the basics of data types.

3.3 Common data types


Learning outcomes After watching this video you should be able to:

- name and understand what data can be represented by the most common data types encountered in computer programming (e.g. integers, floating point numbers, etc);
- understand how to check the type of variable in Python;
- understand the importance of knowing the type of the data you are working with;
- “discuss data types” .

Consider skipping if ... if you have some programming experience, and know the common data types.

3.4 Operations on data and the effect of data type


Learning outcomes After watching this video you should be able to:

- understand the effect that the data type can have on operations;
- understand the distinction between statically and dynamically typed languages;
- understand the distinction between strongly and weakly (loosely) typed languages;
- understand what a mutable (or immutable) variable is, and the implications for certain operations and memory management;
- “discuss data types” .

Consider skipping if ... if you are an experienced programmer, and understand the distinction between, e.g. statically and dynamically typed languages.

3.5 Range and precision in data types


Learning outcomes After watching this video you should be able to:

- understand that computers can only represent a finite range of numbers, and appreciate the consequences of this;
- understand floating point representation of fractional numbers, and the limitations on precision inherent to this representation;
- appreciate the consequences of floating point error, and understand how this phenomenon can be mitigated;
- “discuss data types” .

Consider skipping if . . . if you already understand how floating point arithmetic works.

3.6 Functions: the basics

Learning outcomes After watching this video you should be able to:


- understand the role of functions in computer programming;
- understand the concept of encapsulation in the context of computer programming;
- understand the basic ‘anatomy’ of a function;
- write a basic function in Python, and understand how Python performs type checking;
- “discuss code structure” .

Consider skipping if . . . if you have some programming experience, and feel comfortable with the basics of functions.

3.7 Considerations when writing functions

Learning outcomes After watching this video you should be able to:

- understand the ‘Single Responsible Principle’, and the benefits of adhering to this principle when writing functions;


- understand the difference between ‘pass-by-reference’ and ‘pass-by-value’, and the implications for modifying data in functions;
- understand how the data type determines how arguments are passed to functions in Python;
- understand what is meant by the scope of a variable or function, and why scope is important;
- understand the principle of scoping rules, and how they rules work in Python;
- understand how to create global variables, and when it’s considered bad practice to do so;
- understand what is meant by a ‘side effect’ in the context of functions, as well as why and when they are considered to be bad coding practice;
- “discuss code structure” .

Consider skipping if ... you already write functions that adhere to the Single Responsibility Principle, and understand that the type of a variable can influence whether it’s passed-by-reference or passed-by-value to a function.

3.8 Introduction to objects and classes

Learning outcomes After watching this video you should be able to:


- appreciate the role that objects and classes play in structuring code;
- understand that objects are instances of classes;
- appreciate that objects are about more than just grouping together data and function;
- understand the value of controlling the access to your data, and how this can be achieved with classes;
- understand how inheritance and polymorphism can be used to define flexible interfaces;

- begin to appreciate the difference between the Object-Oriented Programming and Functional Programming paradigms;
- “discuss code structure” .

Consider skipping if . . . if you have some experience with Object-Oriented Programming, and feel comfortable with the basic principles of objects and classes.

3.9 Modules, packages, and libraries

Learning outcomes After watching this video you should be able to:

- understand the value of organizing your code into re-usable modules and packages;
- understand the distinction between modules and packages in Python;
- configure Jupyter notebooks to automatically reload changes in external modules and packages;
- understand the purpose of the `__init__.py` file in Python packages;
- update your Python path to include your own modules/packages using both your virtual environments `.pth` file and your VS Code `config.json` file;
- “Understand why modularity is important and be able to give examples of good practice.” .

Consider skipping if . . . you already understand the distinction between modules and packages in python, and know how to configure your IDE (e.g. VS Code) to make sure your code is on the correct path.

3.10 Modularity: ‘what?’, ‘why?’, and ‘how?’

Learning outcomes After watching this video you should be able to:

- understand what is meant by modularity in the context of computer programming;
- understand the benefits of writing modular code (encapsulation, re-usability, etc);
- understand how to write module code;
- “Understand why modularity is important and be able to give examples of good practice.” 🎓

Consider skipping if . . . you are already a professional software developer with a distinguished career of writing good code.

3.11 Comments, documentation, and style

Learning outcomes After watching this video you should be able to:

- explain what is meant by coding style, and understand the importance of coding style for writing readable code, especially in collaborative contexts;
- understand what is meant by a style guide, and what aspects of writing code they influence;
- name and find some widely adopted style guides;
- explain different types of comments (in-line, multi-line, and comments that can automatically generate documentation, e.g. Python docstrings);
- understand when, why, and how to use comments;
- “Understand the importance of commenting and documentation.” 🎓

Consider skipping if . . . you are already a professional software developer with a distinguished career of writing good code.

A Command Line Interface

A.1 The ‘what?’ and ‘why?’ of CLI

Learning outcomes After watching this video you should be able to:

- understand (at least conceptually) what a Command Line Interface (CLI) is, and how it can be used to control a computer;
- appreciate that the CLI is a complement, not an alternative, to a graphical user interface (GUI);
- begin to understand which tasks/operations are better suited to CLI vs GUI;
- access the terminal on your (mac or Linux) computer.

Consider skipping if ... you already feel comfortable with the basic concept of using the terminal as an complementary interface to your computer (practical details about actually using the terminal will come in subsequent videos).

Disclaimer This video was recorded on a mac using the ‘Linux-like’ macOS terminal. The concepts are the same for Windows OS, but the actual command syntax differs at times.

A.2 Basic commands

Learning outcomes After watching this video you should be able to:

- navigate your file system using the CLI;
- create and delete files and directories with the CLI;
- create and manipulate variables in the CLI;
- understand the difference in behavior of single ‘ ’ and double “ ” quotes.

Consider skipping if ... you already know how to use the terminal to perform basic operations described in the learning outcomes above.

A.3 Shells and scripts

Learning outcomes After watching this video you should be able to:

- understand what is meant by a 'shell';
- understand what is meant by a 'shell script' and be able to write and run your own;
- understand how to specify the shell in a script using the `#!/` ('hashbang') command;
- configure your command line interface (and entire computer) by modifying your `.zshrc` or `.bashrc` file.

Consider skipping if . . . you already feel comfortable writing shell scripts.

A.3.1 Follow along

Specify the shell (Z-shell) to be used in your shell script (first line):

```
#!/bin/zsh  
# the rest of your code
```

Create a new `.zshrc` file ('run commands'):

```
user@host:~$ touch .zshrc
```

A.4 Aliases and functions

Learning outcomes After watching this video you should be able to:

- understand why aliases and functions are important for using the command line interface (CLI) efficiently;
- create your own aliases and functions, that you can call from the CLI.

Consider skipping if . . . you already like to write your own aliases and functions in shell scripts.

A.4.1 Follow along

Create a new alias:

```
#!/bin/zsh
alias <alias_name>=<alias_code>
```

Create a function called `mkcd` to simultaneously create and change into a new directory:

```
#!/bin/zsh
mkcd() {
    local name="$1"
    mkdir ${name}
    cd ${name}
}
```


B Virtual Environments

B.1 The ‘what?’ and ‘why?’ of virtual environments

Learning outcomes After watching this video you should be able to:

- understand what a virtual environment is;
- understand what problems virtual environments solve, and how they solve them (conceptually);
- understand why using virtual environments is good practice.

Consider skipping if ... you already understand the concept of a virtual environment (practical details about creating and using virtual environments will come in the subsequent videos).

B.2 Python’s inbuilt tools: `pip` and `venv`

Learning outcomes After watching this video you should be able to:

- understand the role that `pip` and `venv` play in the development of Python code;
- create new virtual environments using `venv`;
- create a shell function to conveniently activate your virtual environments;
- configure VS Code to make use of a virtual environment of your choosing;
- understand that virtual environments are not portable.

Consider skipping if ... you already know how to use `venv` for creating and managing virtual environments.

B.2.1 Follow along

First, create a new directory where your `venv` virtual environments will live:

```
user@host:~$ mkdir .venvs
```

To create a new virtual environment named `venv_name`, run the following:

```
user@host:~$ python -m venv .venvs/<venv_name>
```

To create a function that allows you to activate your virtual environment, add the following lines to your `.zshrc` file:

```
#!/bin/zsh
activate_venv() {
    local env_name="$1"
    local env_dir="${HOME}/.venvs/${env_name}"
    source "${env_dir}/bin/activate"
}
```

The function can be used as follows:

```
user@host:~$ activate_venv <venv_name>
```

To deactivate a `venv` virtual environment:

```
user@host:~$ deactivate
```

B.3 Conda, Miniconda, and Ananconda

Learning outcomes After watching this video you should be able to:

- understand the relationship between conda, pip, and venv;
- understand the relationship between conda, miniconda, and anaconda;
- install miniconda via the command line;
- regain control of your `.zshrc` or `.bashrc` file after installing conda, and manually activate conda when you need it;
- create new virtual environments using conda;
- make use of conda virtual environments in VS Code.

Consider skipping if . . . you already know how to use conda.

B.3.1 Follow along

To create a new conda virtual environment called <name> using Python 3.xx:

```
user@host:~$ conda create -n <name> python=3.xx
```

To activate a conda virtual environment called <name>:

```
user@host:~$ conda activate <name>
```

To deactivate a conda virtual environment:

```
user@host:~$ conda deactivate
```