Other languages and C++
Writing bash scripts

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se
Fysikum, Hus A, Room 403
Visiting time: 11:00-12:00 Every day
other times: send me an email
Alternative: Use Canvas!

Notation

- There's a set of symbols and idioms that are commonly used in command line tutorials and you should know about. The description of the grammar of a command is often called **synopsis,** or brief summary**.**
- **Spacing.** In general there is **always** a space between a command an every of its options, that is, every word of a command that is shown in these slides.
  However, in some cases it might be tricky to see it, and I will use the symbol ▮ . For example **man▮bash**
- `command`
  This graphics above is meant to represent a **command**. You are supposed to write exactly as it looks.
- `command` *`<argument>`*
  The **<>** (**angle brackets**) are used to identify a **mandatory** argument of the command. The command will NOT work without the things in the angle bracket.
  The above usually means to run the command and to **substitute** the string <argument> with the argument **without angle brackets.**
  Remember, in most languages brackets have a special meaning. The special meaning of the angle brackets was shown in the CLI tutorial.
- `command` *`ARGUMENT`*
  In man pages, sometimes **capital letters** are used instead of the angle brackets <>. The meaning is exactly the same as the angle brackets, the capitalized string means **mandatory. We will not use this notation in this tutorial** because it might be confusing, but you will find it in the linux **`man`** pages
- `command` *`<argument> [<argument>]`*
  The **[ ]** (**square brackets**) are used to identify and **optional** part of the command. The command will work if you omit the content of the square brackets [].
  However, if you add a second argument, it must be as defined within the angle brackets <>.
- `command` *`[<argument1> | <argument2>]`*
  In command descriptions, the **|** (**pipe symbol**) is used to identify a **mutually exclusive** part of the command. You can use **EITHER `<argument1>` OR `<argument2>`** but **NOT both of them**.
  This is inherited from formal grammar notations.
  In code snippets or pieces of code, the pipe is part of the code and must be copied/written as it is.

Outline

- Goals
- Datasets
- Automation using scripting
  - Genesis of an algorithm
- Introduction to scripting
- Bash
  - Scripts
  - Variables in bash: environment, binding, scope
  - Control structures

Goals and non-goals of this tutorial

- Goals:
  - Being able **NOT TO PANIC** when somebody gives you something you've never seen before (will happen in your entire career)
  - Being able to **search for information depending on a task** one wants to achieve. (see references at the end of these slides!!!!)
    - Google is NOT always your friend if you don’t know what you’re searching for.
  - Being able to identify **which language is best for which task**
    - And to **compose** different languages to achieve a goal
  - Being able to write a **bash script**.
  - Understanding the concepts of **Variable, Environment**, **binding**, **scope**.
- Non-goal:
  - Become a script-fu master. It takes long time for the black belt :)
  - Become a coder. We cannot do this in a lecture, there's plenty of dedicated courses out there

Handling datasets

Typical scientist workflow

- **Someone** (usually your supervisor, today is ME) gives you reference to some data and some obscure code written by elders who now moved to the end of the known universe
  - Nobody knows what the **data** looks like and what it contains – just that is it about your science!
  - Nobody has any idea what the **code** is like and how to change it. No documentation, no one left alive to tell you
- **You** have to figure out all the details by yourself
- Today **I** will teach you a few tricks to survive the data part

Datasets

- A dataset is some digital collection, maybe a file or a set of files, that contains data we want to use.
- A dataset usually has his own **format**.
  - A format is a **set of rules** that define in a rigorous manner how the content of the dataset should be read, what are their meanings and the relationship among the dataset information
  - The format can be a well know data format, more or less standardized, or some custom data format that one needs to learn
  - A **description** of the format is usually provided by the community that generated the dataset. It is very rare that a dataset contains information about its format.
  - Very common format names
    - CSV (comma separated values)
    - XML (eXperimental Markup Language)
    - JSON (JavaScript Object Notation)
  - **Exercise 3.1** : Search for those names and “specification” on Google and learn about what they look like.

Some dataset examples

- Click these links
  - https://github.com/floridop/MNXB01-2019/blob/master/floridopag/lecture3/lecture3examples/data/nintendowiigames.xml
  - http://musicbrainz.org/ws/2/artist/5b11f4ce-a62d-471e-81fc-a69a8278c7da?inc=aliases&fmt=json
  - http://www.smhi.se/pd/klimat/ozone/data//oz2019.vin
- Could you guess what is this dataset about?
- What is the format?
- What is the information contained?
- Hints:
  - look at the link
  - Check the top of the file
  - Look for recurring keywords
  - Look how the keywords repeat, can you guess there is a structure?
    - Try to distinguish between data (values) and metadata (description of values/structure)
  - Look at the numbers/text. Can they be related to something you know, just by common sense?

Sample data file: investigation

```
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitl
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>890</id>
<GameTitle>Rayman Raving Rabbids TV Party</Game
<ReleaseDate>11/18/2008</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>908</id>
<GameTitle>Super Mario Galaxy 2</GameTitle>
<ReleaseDate>05/23/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

What can we say by observing this data?

- It seems to be structured in some way.
- There is some metadata information at the top that might hint at some known format. Search “XML” on google?

Can we guess something about the structure?

- It seems to have opening and closing tags <tag></tag>
- The tags seems to represent a tree structure

Can we guess something about the content? Anything you may know about?

- Clearly seems to speak about games of some kind
- Platform seems to hint to some kind of device, there is the name of some company in it

Datasets can be “dirty”

- The data is not always as you expect. Close inspection might reveal inconsistencies and corner cases that have to be “*sanitized*” or “*validated*”, or simply you need a subset of the whole dataset.
  In any case, something that requires special care.
- In most cases you will need to *rework the dataset* in order to process it with your code
- In any case, **never tamper the original dataset**. Do all the changes on a **separate copy**.
- Example: take only games whose name starts with B, G or Z.
- Devil is in the detail:
  - **Encodings**. To show the content of a file, especially a text file, an operating system has to know the encoding of a file.
  - look for **invisible or non-ASCII characters.** These are usually symbols for non English-US languages, or **control characters**

Encodings

- The encoding is a **table** that maps bytes contained inside the file to a set of graphical representations. Some files carry this information at the beginning of the file, but for most text files this needs to be guessed using a *magic number.*
  Most editors can guess automatically and allow you to force-save in some encoding. In linux you can check the encoding of a file with the `file` command.
  ```
  file <filename>
  ```
- Most common encoding sets for text files are:
  - ASCII (US-english, Latin)
  - UTF-8 (US-english and Latin with extended chars like öää̊)
  - UTF-16 (Symbol languages (Asian, Arabic, Hindi…))
- On the geany editor, you can read the encoding at the bottom of the window.

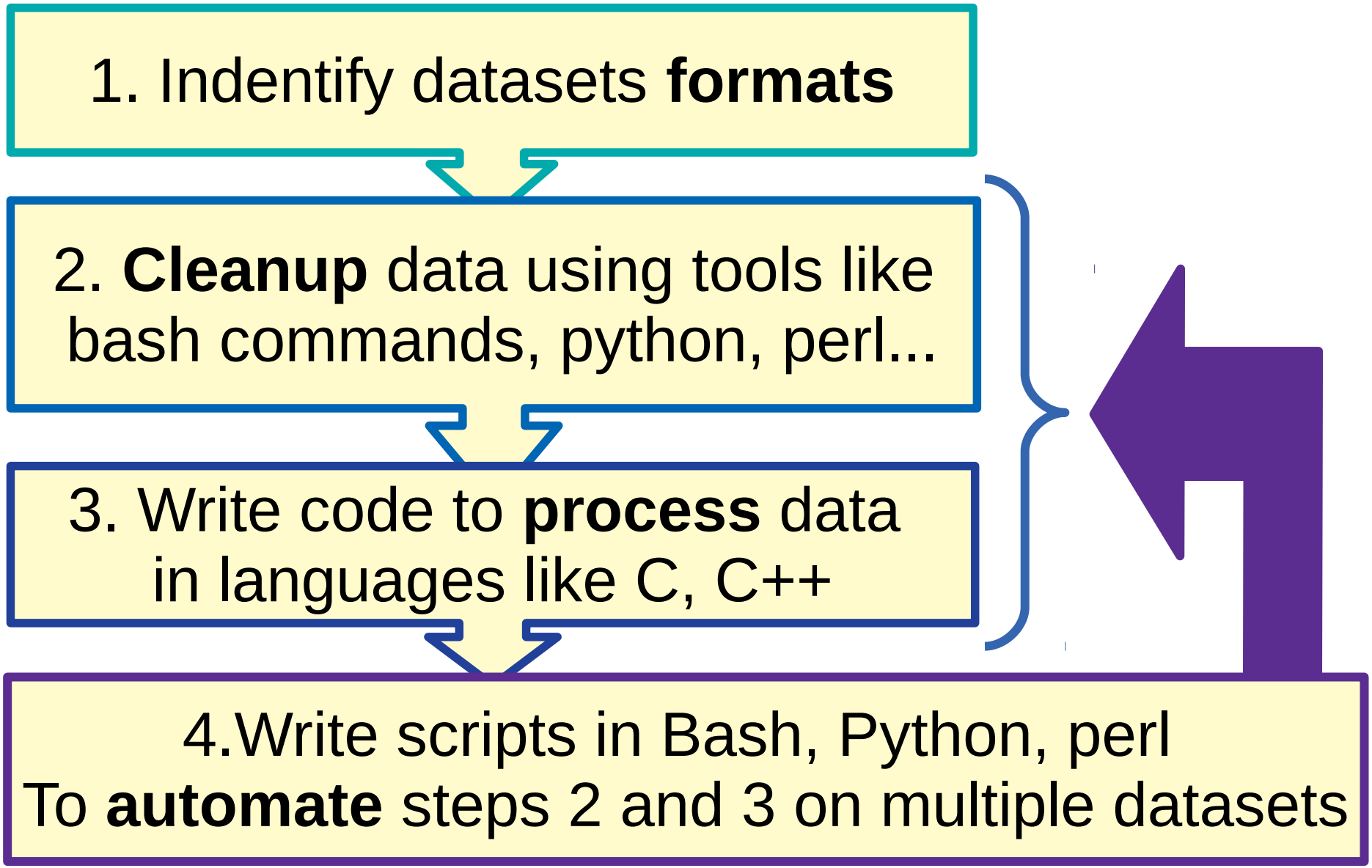
Control Characters

- They’re always there especially in text files. Common examples are newlines:
  - (Most linux-unix) Line Feed (**LF**): Makes a text file go to the next line. Usually represented as `\n`
  - (Mac OS) Carriage Return (**CR**): makes a text file go to the beginning of a line. Usually represented as `\r`
  - (Windows) (**CR LF**): makes a text file go to the beginning of the line and then to a new line. Usually represented in programming languages as `\r\n`
  - More info on https://en.wikipedia.org/wiki/Newline
- You can see them with `less -u <filename>` or with geany through the menu View→”Show line endings”
- The symbols \ is used to represent “escape sequences”, used for special control characters. Some other common ones:
  - \t : tab, a fixed size set of spaces
  - \s : a space
  - On geany, enable with View→”Show white space”

Cleaning up a dataset

- In the case of text-file datasets, usually the best is to use tools that were created to handle text – or the so-called **string** datatype
- C and C++ are notoriously **very bad with strings**
- The cleanup is easier if done with some other language like python, perl or some of the **bash commands**
- In what follows we will learn how to automate a workflow where the data needs to be cleaned up first
- I am teaching bash, but there's nothing preventing you from using any other tools you fancy such as python or Microsoft Excel or Google Spreadsheet. The use if bash is just practical if you have hundreds of text files to parse, and all the tools are for free.
  - You can run an ubuntu/linux-like shell also on windows (e.g. cygwin) without virtualization. The use of these tools will become more and more common.

Typical scientist workflow summary



Workflows with various tools

- In the free software/open source community everyone shares knowledge about coding.
  - This usually means that someone's work is based on someone else's
  - This generated a style of creating software that is a mix of different programming languages, tools, practices:
    composing different applications to achieve a goal
- It is very common that your C++ code will require some **preparation** before the build for which leads to **tedious repetitive** commands to type in
- **"tedious repetitive"** is what a computer is good at.
  Let it do it on your behalf.
  A human has better things to do in life than monkey-coding!
- *Scripting languages* are very good to automate boring work.

Automation and composition of languages

- Cornerstone of open source programming: if something exist that does a task, and it does it good, use it and do not rewrite code
- **Automation** of repetitive tasks
- Make use of interoperability within languages
- Technique: identify subproblems and separate tasks, increasing “debuggability”
- Choose the right command/language for each subtask

Genesis of an algorithm: a top down approach

- Write a list of each main task translating the description of the problem.
- Open your favorite text editor and start writing down as *comments* the steps to the algorithm. You can even write that on paper first.
- An example of this process is this pseudocode in git taken from last year's homework:
  - https://github.com/floridop/MNXB01-2018/blob/master/floridopag/HW3/fortuneteller.sh.pseudocode#L134

Homework 3

- The goal of the homework is to familiarize with dataset cleanup using bash as a tool to automate such cleanup.
- The homework will be published on Canvas, the instructions to carry it on will be on this year's github repository:
  - https://github.com/floridop/MNXB01-2019/blob/master/floridopag/tutorial3/homework3/README.md
- Hint: Check the solutions of previous year assignments on github or the course webpage:
  - http://www.hep.lu.se/courses/MNXB01/index-2018.html
    - https://github.com/floridop/MNXB01-2018/tree/master/floridopag/HW3
  - http://www.hep.lu.se/courses/MNXB01/index-2017.html
    - https://github.com/floridop/MNXB01-2017/tree/master/flopaganelli/HW3b
  - http://www.hep.lu.se/courses/MNXB01/index-2016.html
  - http://www.hep.lu.se/courses/MNXB01/index-2015.html

Introduction to BASH

Scripting vs coding

- The word script is taken from a theatrical play script: a description of the environment on stage, a sequence of lines and gestures to do
- There is no practical difference between writing code in a compiled language and a scripted one.
- The main difference is that scripted languages **do not require compilation**.

BASH

- Bash stands for **B**ourne-**A**gain **SH**ell. It’s a rewrite by the GNU member Brian Fox of one of the unix **`sh`** shells, called Bourne shell by its author surname (Stephen Bourne).
- Yesterday you learned a few commands in bash, today we will write programs with them.

A bash script and its components

- **Bash** is not really a programming language. It is more like a **command scripting language** for automation of tasks, with some programming language features.
- Instead of libraries you will mainly use the **GNU/Linux userland** and **GNU/Linux coreutils** software, a set of commands that help automate common tasks, or other bash scripts.
- A **bash script** is nothing more that a sequence of commands written in a file.
- The bash interpreter will process those in sequence, from the top line to the bottom
- Like C++, is possible to define **variables** and **control structures** in the scripting language.
- However, the bash script language has little to share with the complexity of C++. All that it can do is to **execute commands, test conditions, and store things in variables**.
- Most commands we will see today are documented on man. You can type `man bash` to read the full documentation.
- Consider the following code, a script called `getcpuinfo.sh`:

```
#!/bin/bash

# 1. use the cat command to print the file /proc/cpuinfo
# 2. extract the first two lines of the above output with head
# 3. store the output of head in the CPUINFO variable
# it is all done in the following one line!
CPUINFO=$(cat /proc/cpuinfo | head -2)

# write the content of CPUINFO to screen
echo "$CPUINFO"
```

Anatomy of a bash script

```
#!/bin/bash

# put the output of cat in the variable CPUINFO
CPUINFO=$( cat /proc/cpuinfo | head -2 )

# write the content of CPUINFO to screen
echo "$CPUINFO"
```

The first line has a special syntax: #! tells bash which **interpreter** to use. It might be another shell!

Every other line starting with a hash # is a **comment**. The interpreter ignores everything that follows until the end of line. Useful to describe code to human readers.

This tells bash to execute a command and return its output.

A **variable definition** is any string followed by a **=** symbol. It is a convention to use capital letters.
Remember that *case matters*, `cpuinfo` is different from `CPUINFO`!

A **variable call** is any **variable name** prefixed by the **$** symbol. Case does matter here. The quotes affect the output, that in this case depends on how the `echo` command works.
The **$** symbol stands for "**give me the value contained in that variable**"

Executing a script

- The script can be **made executable** as if it was a command.

```
pflorido@tjatte:~> chmod +x getcpuinfo.sh
```

  - If you forgot to make the file executable, you will get the following error:
    `bash: ./getcpuinfo.sh: Permission denied`

- To **run** or **execute** a file in the current directory, prefix it with **./**

```
pflorido@tjatte:~> ./getcpuinfo.sh
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 15
model name : Intel(R) Core(TM)2 CPU 6400 @ 2.13GHz
stepping : 6
cpu MHz : 2127.650
```

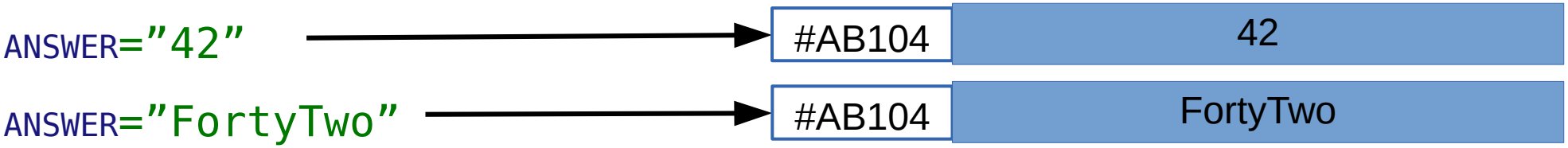
Variables: definition, initialization

- A **variable** is an identifier, a name, for a memory location.
- To **define** a variable means to tell the **interpreter** to find a free memory space for that variable. This memory space has an index.
- In bash, you define a variable by simply writing a string with CAPITAL LETTERS (by convention) that starts with a letter **on the left side** of the symbol =
  Note: BASH doesn't like spaces that much!

```
ANSWER=
```

- If a bash variable is **not initialized**, the memory space at that index contains the blank/empty string

- To **initialize** a variable is to **assign** a **value** to it.
  It means putting such value inside the memory location identified by that variable name.
- In bash this is done by writing a value on the **right side** of the equal sign **=**



Variable types in bash

- In BASH, variables have *no explicitly defined type*, because actually there is **only one type**.
- It is **implicitly assumed** that the content is a **string: a sequence of characters**.
  The maximum size depends on the system.
  - Memory Allocation is always done dynamically depending on the assigned value
  - **Consequence**: Doing arithmetic with bash is a **bad idea**. Bash does not understand numbers so well...

Var name	Var type	Associated size	Initial tentative logical memory location pointer	value
ANSWER	Always string	Depends on system configuration	#AB104	42 as a string
ANSWER	Always string	Depends on system configuration	#AB104	FortyTwo

Variables: retrieving values

- So far we've seen how to assign a value to a variable. But how to *read* or *retrieve* such value from the computer's memory?
- In bash one simply prefixes the variable with the `$` (dollar) sign.
- `$ANSWER` returns the value of the `ANSWER` variable.

Calling variables values in different ways

- `$VAR` returns the value contained in the variable called `VAR`.
- `${VAR}` returns the value contained in the variable called `VAR` but it makes easier to spot the boundaries of the variable name. It can be used to concatenate string values and strings, like in the previous code:

```
${TARGETDIR}/*;
```

it shows clearly that the name of the variable is `TARGETDIR`

Using the suggested file editor **geany**

Creating and editing a file

- During the tutorial you'll be asked many times to do things with files. For those of you not familiar with **file editing**, here's a small how-to.
- There are many ways of creating a file.
  One way is by using a **text editor**
- The favorite text editor for this course is called **geany**. Can you find the icon in the menu? Open it by clicking on the icon.

- Alternatively, open a terminal 

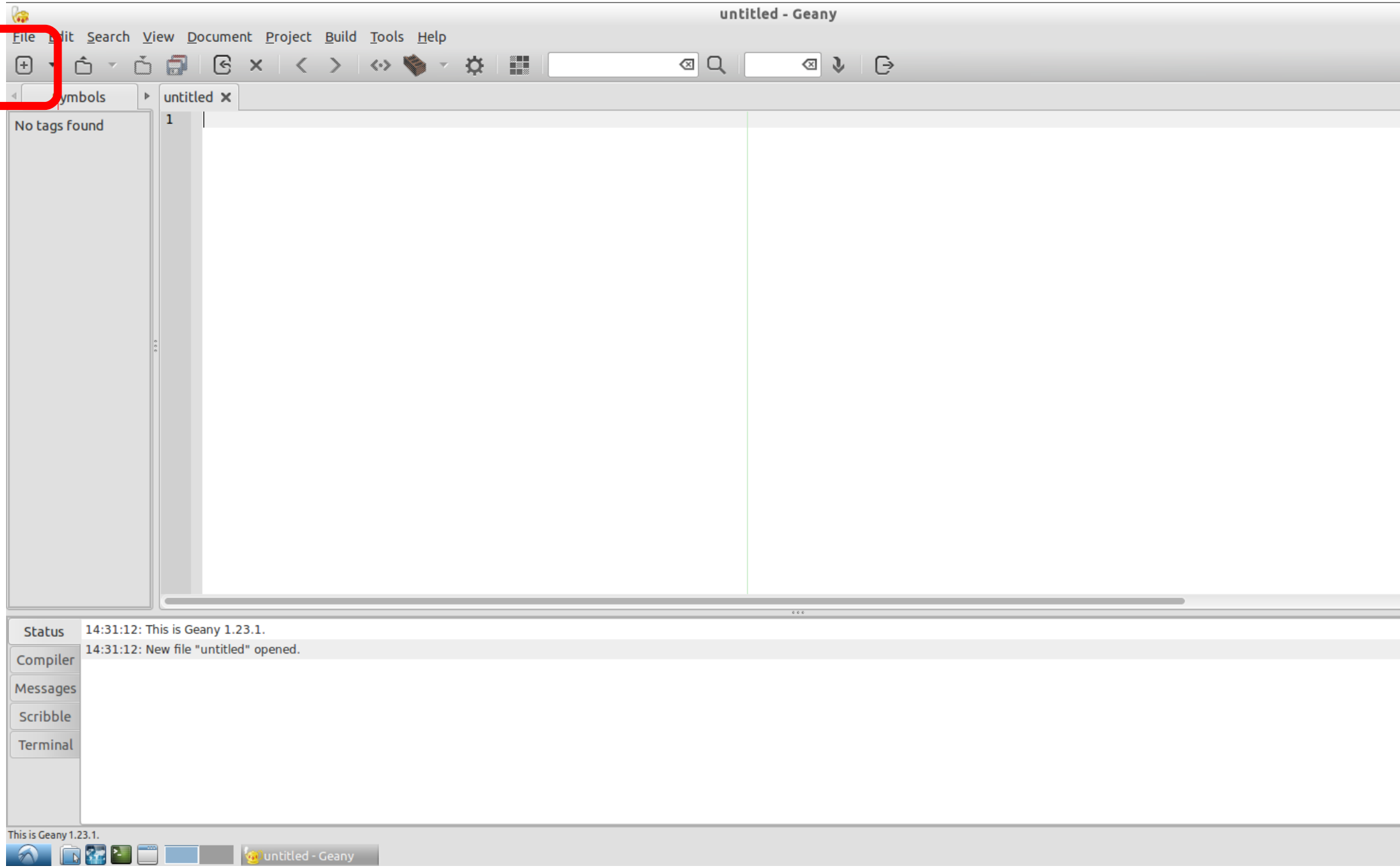
and write the command:

```
geany &
```

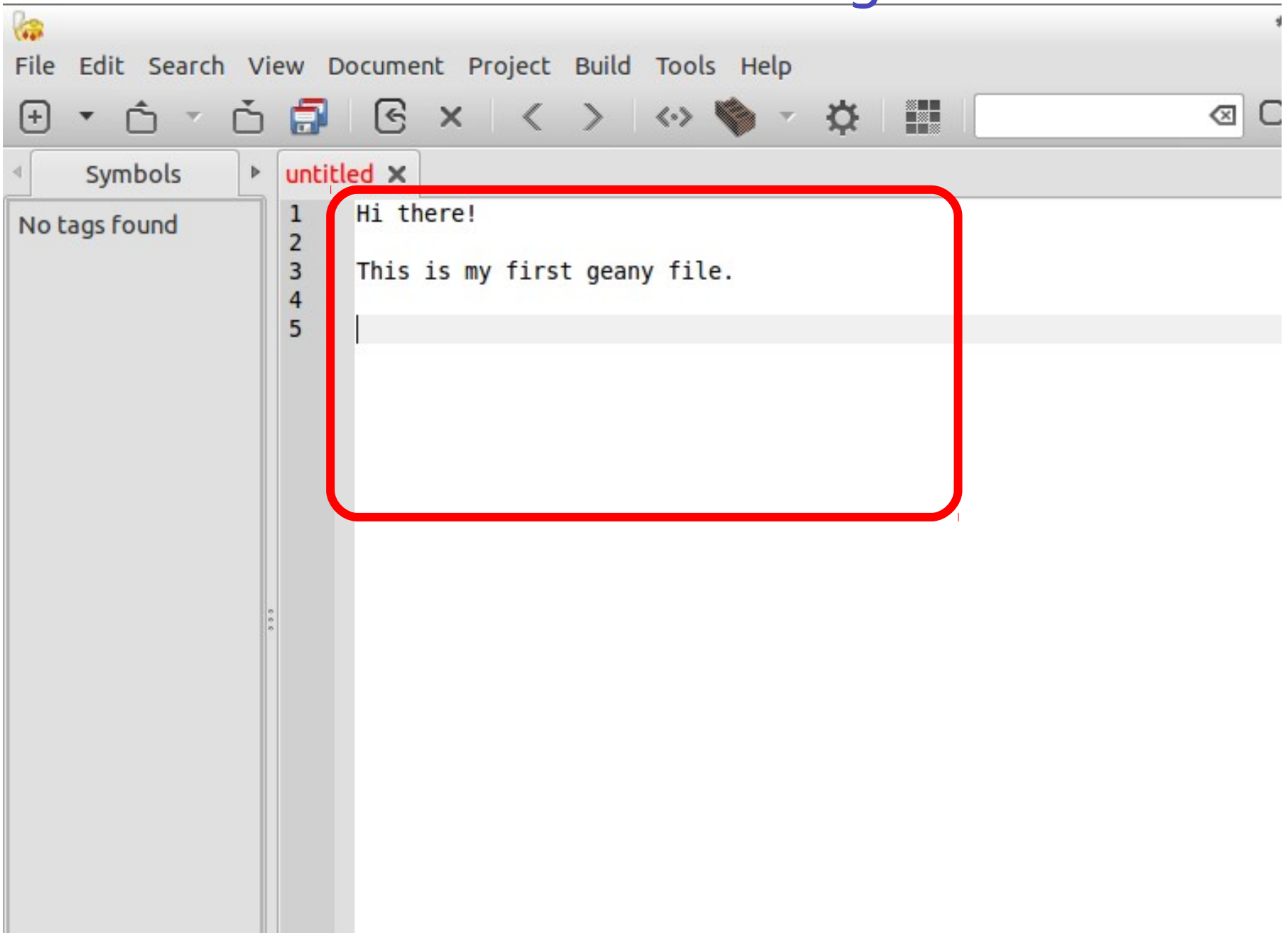
(the & symbol sends the command execution in background, see tutorial 2!)

- Tip: you can switch between windows using the combination Alt + TAB

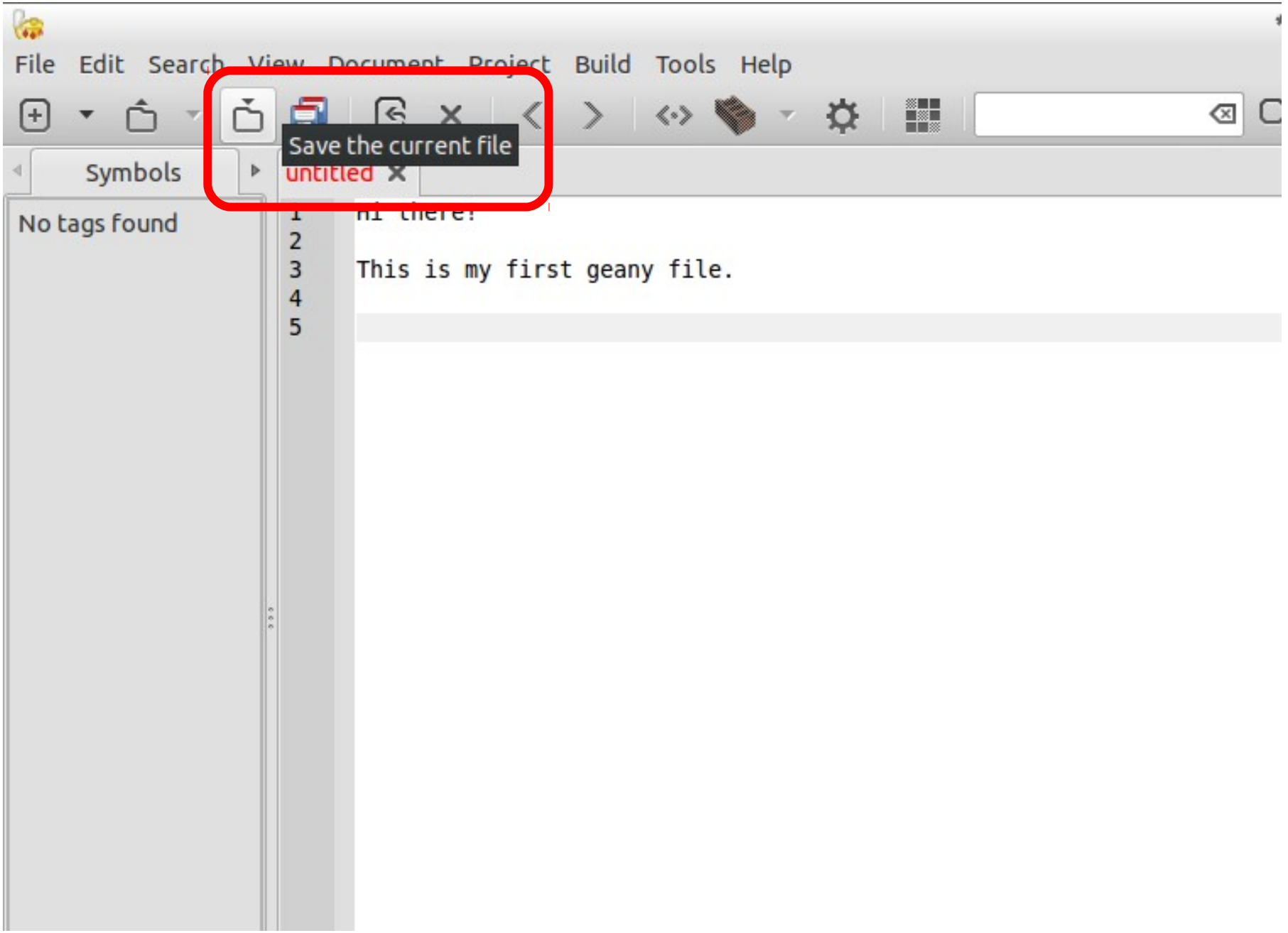
Editing and saving a file: create new



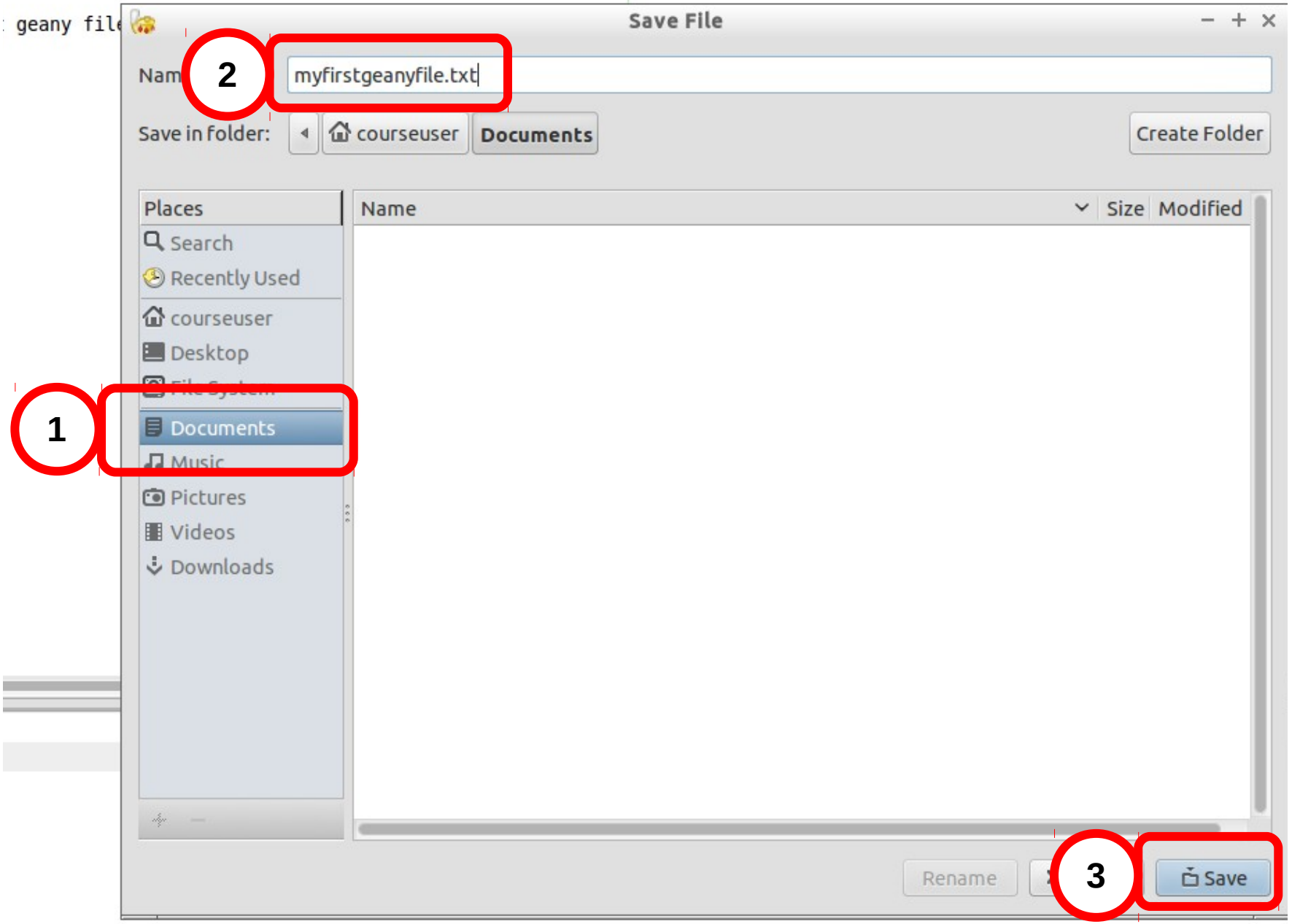
Editing and saving a file: write something



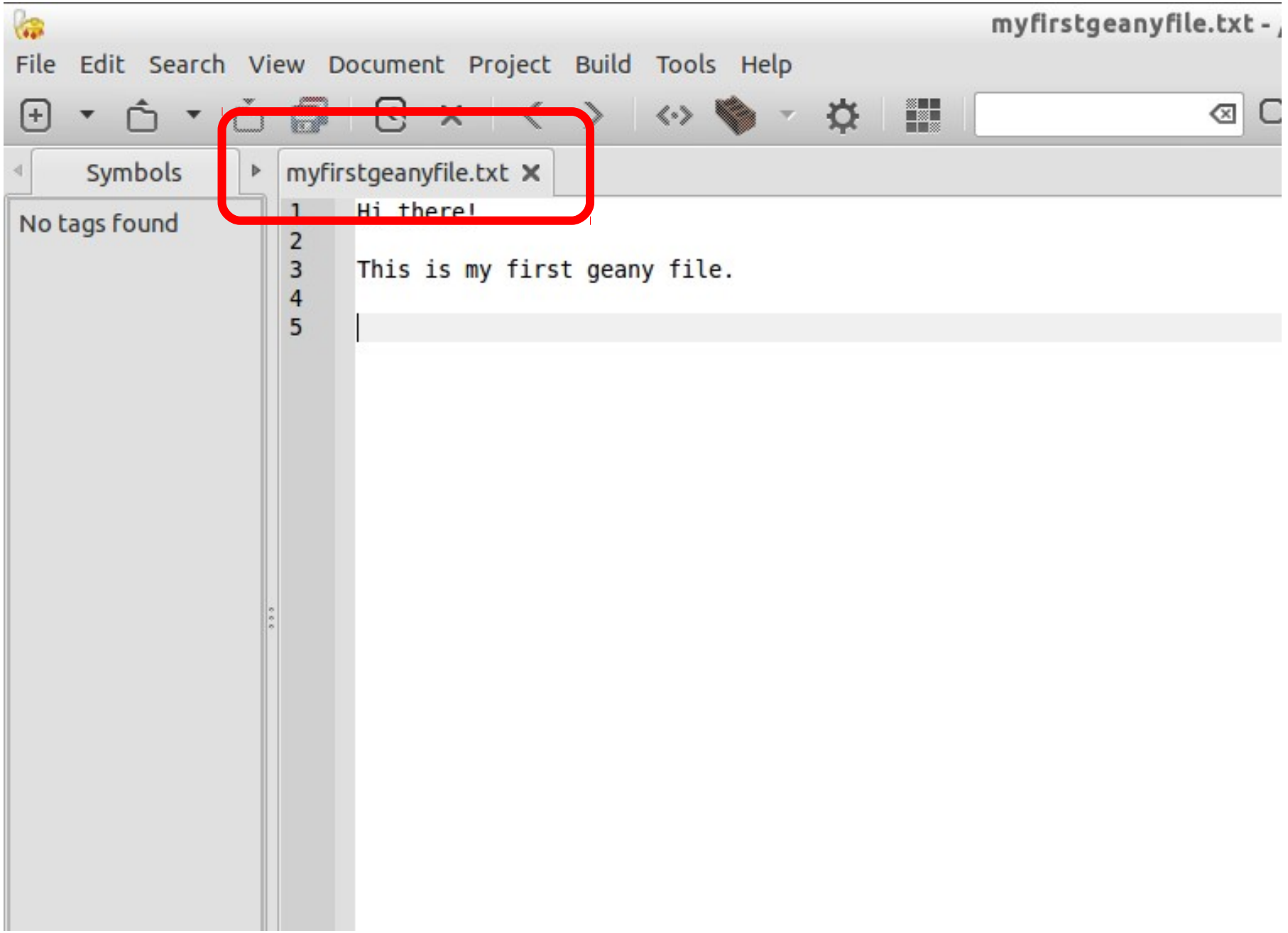
Editing and saving a file: save or save as



Editing and saving a file: choose location and filename

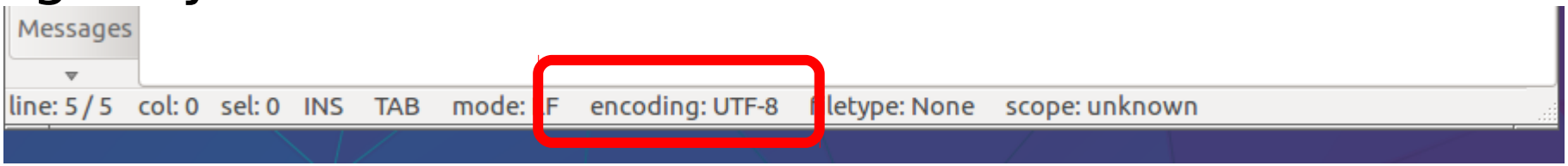


Editing and saving a file

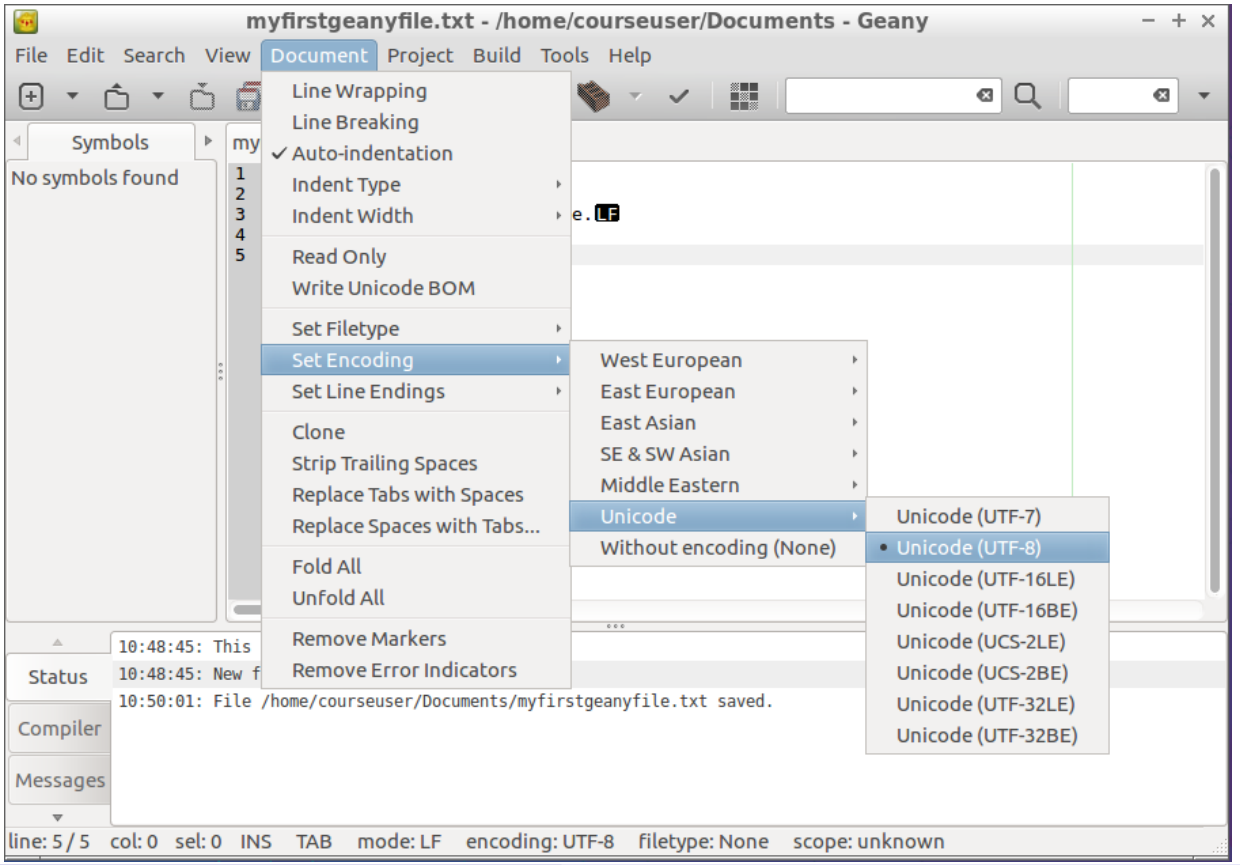


Encodings

- One can see the encoding of a file at the bottom of geany's window:

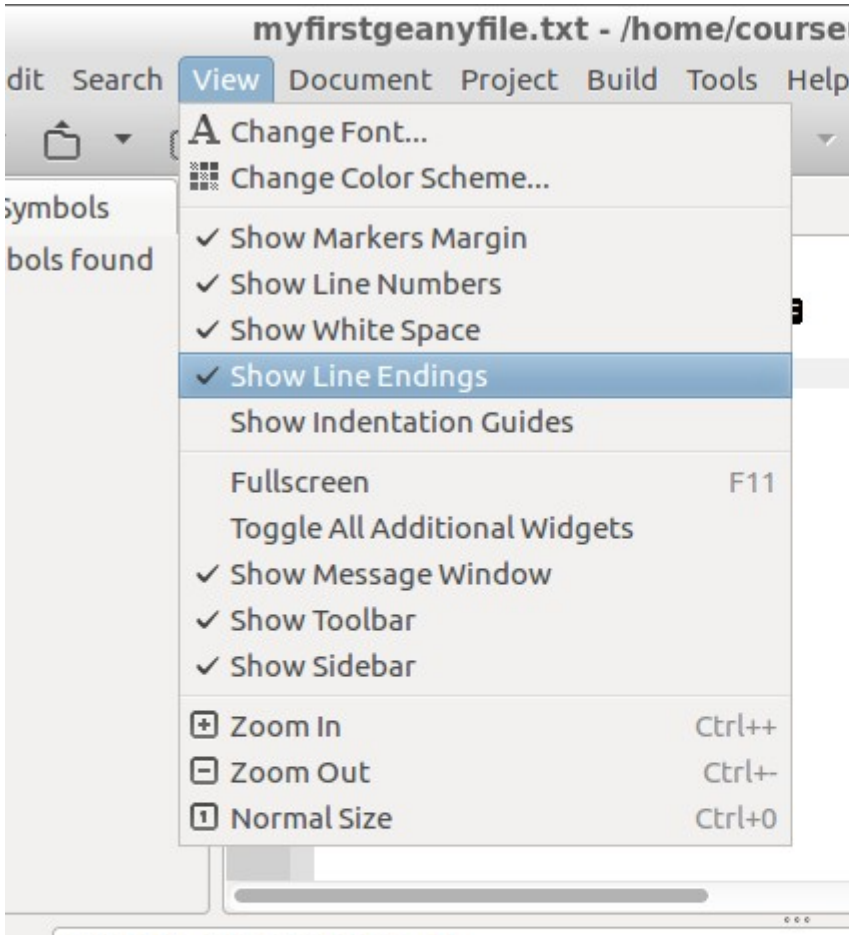


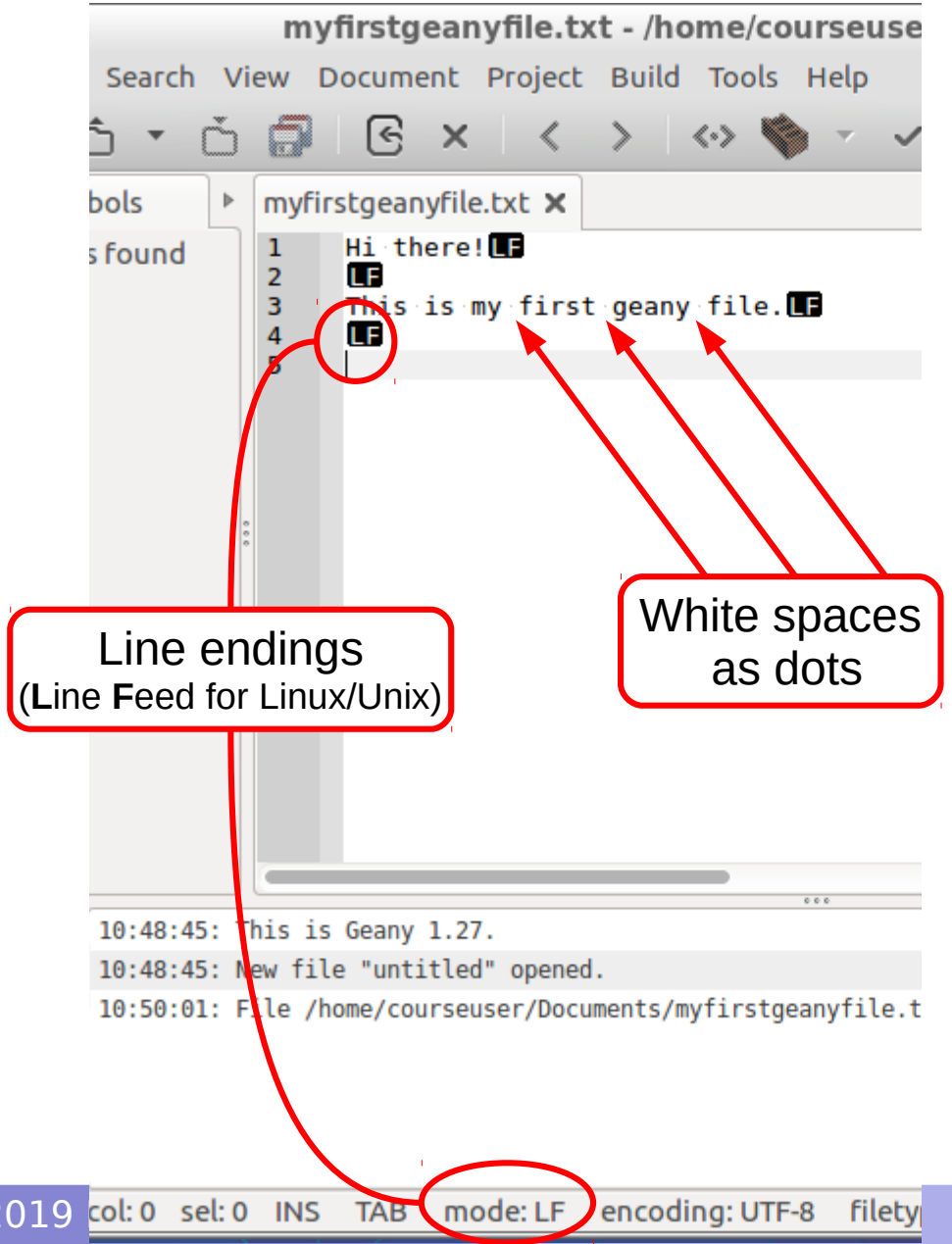
- And can eventually set/change the file encoding if required:



Control/invisible characters

- Enable view of control characters:





Bash Tutorial part 1

- Create a script
- Execute a script
- Variables
- Print out
- Command substitution and pipes

Exercises

- Read slides 21-28 if you get lost on one these exercises.
- **Exercise 3.2** (write a bash script)**:** Open geany, write and save the following code as file `answers.sh`

```
#!/bin/bash

# define and initialize the ANSWER variable
ANSWER=42

# write the content of ANSWER to screen
echo "$ANSWER"
```

- **Exercise 3.3** (execute a bash script)**:** make the script `answers.sh` executable and execute it as described in slide 24.
- **Exercise 3.4** (echo)**:** Familiarize with the `echo` command. It is used to print out information to the screen.
Edit `answers.sh` so that at the end of the program it prints out "`The content of the variable answer is 42`"
- **Exercise 3.5** (modify scripts)**:** Modify the content of answer with 42+42, save and execute again. What happens? Can you make the code print out the content of the ANSWER variable?

Predefined variables in scripts

- **Prefixed by the $ symbol**, they are instantiated automatically in bash at the start of the script.
- **Various**:
  - `$PATH`: list of paths where executable commands are
  - `$PS1`: prompt format
  - `$SHELLOPTS`: options with which the shell is run
  - `$UID`: User ID of the user running the script
- **Process info and status codes**:
  - `$$`: process identifier (PID) of the script itself.
    The **PID** is an **integer number** that the operating systems assigns to a binary file once it is ran, that is, when it becomes a process. **It uniquely identifies a running program** until the machine is shut down. See Lecture 3 slide 76 and Balasz slides for Tutorial 2
  - `$?`: exit code of the last executed command (0 if it ended without errors, any other number otherwise). More about it later in the tutorial.
  - `$!`: PID of last command executed in background
- **Script parameters/arguments**: `$#`, `$0, $1, $2`....
  - `$#` is the number of arguments passed to the script
  - `$0` is the name of the script itself as called to be executed
  - `$1..n` is each string that follows the name of the script.

Exercises

Exercise 3.7:
What is the predefined PATH variable?

During the course we ran commands that did not need a **./** in front. The reason is: the directory where our code is placed is not known by the system as a place where executables are.

This list is contained in the predefined variable **PATH**.

Add the following line at the end of the `answers.sh`:

```
echo “PATH value is $PATH”
```

Save and execute the script again: this line above will show the folder path where the the system looks for executables.

Exercises

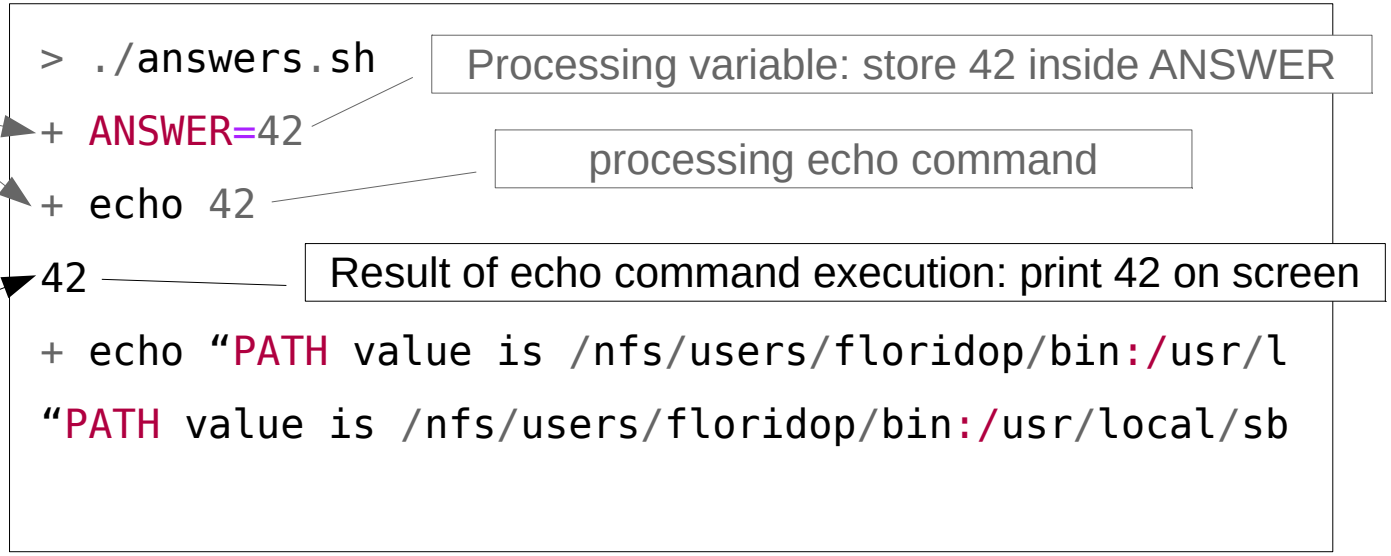
Exercise 3.8 (enable/disable debugging mode):

Enable **Debugging** to debug the script, that is, see what is doing while running, modify the *first line* of `answers.sh` as below:

```
#!/bin/bash -x
```

Save the file and execute it again. See the differences in the output.

- The lines starting with + show what line the interpreter is **processing**
- the lines **without** + are the output **result** of the process.



You may delete ‘**-x**’ after you’re done, and just add it when you do not understand what the code is doing.

Using parameters and quotes

- **Exercise 3.9:** Let's modify the `answers.sh` script to take in input the number it has to print. Using the predefined variable `$1` in slide 40:

```
#!/bin/bash

# set the variable to the first input parameter
# to the answers.sh script
ANSWER=$1

# write the content of ANSWER to screen
echo "You asked me to write: $ANSWER"
```

- **Exercise 3.10:** execute `answers.sh` passing a value of your choice, for example: `./answers.sh FortyTwo`
- **Exercise 3.11:** Pass the string `(42)`. What happens?
  - Certain characters are special in Bash (see Tutorial 2). If you want to pass them as string, you must enclose them in quotes ‘ or double quotes “.
    Try again with the following:
    - “`(42)`”
    - ‘`(42)`’
    - “`$PATH`”
    - ‘`$PATH`’
- The meaning of the quotes is different:
  - The single quote ‘ is *verbatim*, that is, what is inside the quotes is taken exactly as it is.
  - The double quote “ allows to resolve/fetch the value of variables, as in *echo*

Tutorial continued: download the bash examples from git

- Open a terminal.
- Change Directory into your home.

```
cd ~
```

- Create a directory for GIT. We will see it later in the course, just execute the commands below.

```
mkdir git
cd git
```

- Use GIT (we will see it later in the course) to download the examples for this tutorial. This will create a directory called MNXB01-2019.

```
git clone https://github.com/floridop/MNXB01-2019.git  MNXB01-2019
cd MNXB01-2019
```

- Create a directory for your own activity and change directory to it. The directory name should be **your name** and the **first three letters of your last name**. For example, my name is Florido Paganelli and I will use *floridopag* ***but you must use yours.***

```
mkdir namesur
cd namesur
```

- Copy the content of my **tutorial folder** inside your newly created folder
  - `cp -r ~/git/MNXB01-2019/floridopag/tutorial3 .`
- Enter the tutorial folder you just copied, there is a folder called bash:
  - `cd tutorial3/examples/bash`
- List the folder contents to see the scripts with
  - `ls -l`

Predefined variables example

```
#!/bin/bash

# predefinedvars.sh
# call with: ./predefinedvars.sh arg1 arg2 arg3
#

# print out info about arguments to this script
echo “Number of arguments: $#”
echo “Name of this script: $0”
echo “Arguments: $1 $2 $3 $4”

# print this script's Process IDentifier:
echo “PID is $$”
```

Let’s consider the predefined variables in slide 18.

Exercise 3.12: Run the script. Remember to `chmod +x predefinedvars.sh` to make it executable!

Exercise 3.13: modify the script so to check the output of some other predefined variable, in particular $* and $@
Ask me if you don’t understand what is going on!

BASH power: Command substitution and pipe

- Consider the `getcpuinfo.sh` script.

```
#!/bin/bash

# put the output of cat in the variable CPUINFO
# take only the first 2 lines using head
CPUINFO=$(cat /proc/cpuinfo | head -2)

# write the content of CPUINFO to screen
echo "$CPUINFO"
```

  - Notice the use of the `$( … )` construct. It is used to *capture* the output of the commands between parentheses.
  - Notice the use of the | (*pipe*) symbol. It sends the output of the **`cat`** command as an input to the **`head`** command

- **Exercise 3.15:** execute `getcpuinfo.sh` as described in slide 18.

- **Exercise 3.16:** Familiarize with the `head` command. It selects a certain number of lines from the beginning of a file. Make it select 10 lines instead of just 2.

Bash Tutorial part 2

- Functions
- Environment, Binding and Scope
- Customizing your environment
- Conditions
- Control structures
- The exit status
- Useful commands

Functions

- Sometimes we need to do the same task a certain number of times, and it's a bit boring to copy paste. Consider the following example `getsomelines.sh` where we want different lines from different files:

```
#!/bin/bash

# put the first two lines of /proc/cpuinfo in CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -2)
# write the content of CPUINFO to screen
echo "First 2 lines of /proc/cpuinfo:"
echo "$CPUINFO"

# put the first four lines of /proc/meminfo in MEMINFO
MEMINFO=$(cat /proc/meminfo | head -4)
# write the content of MEMINFO to screen
echo "First 4 lines of /proc/meminfo:"
echo "$MEMINFO"

# put the first line of /etc/hostname in HOST
HOST=$(cat /etc/hostname | head -1)
# write the content of MEMINFO to screen
echo "First 1 line of /etc/hostname:"
echo "$HOST"
```

Functions – identifying parameters

- When you have code like this, it’s good to identify similarities that could be parameters to a function: can we simplify the code?

```
#!/bin/bash

# put the first two lines of /proc/cpuinfo in CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -2)
# write the content of CPUINFO to screen
echo "First 2 lines of /proc/cpuinfo:"
echo "$CPUINFO"

# put the first four lines of /proc/meminfo in MEMINFO
MEMINFO=$(cat /proc/meminfo | head -4)
# write the content of MEMINFO to screen
echo "First 4 lines of /proc/meminfo:"
echo "$MEMINFO"

# put the first line of /etc/hostname in HOST
HOST=$(cat /etc/hostname | head -1)
# write the content of MEMINFO to screen
echo "First 1 line of /etc/hostname:"
echo "$HOST"
```

Functions - definitions

- One can define functions to reduce complexity and increase readability
- A bash function has:
  - A **name**, so that is possible to reuse the function, usually followed by two parentheses **()**; example: `myfunction()`
  - A **definition**, where the operations that the functions will do are defined. It is also called the **body** of the function.
    - The body of the function MUST be enclose in curly brackets **{ }**. These delimit a **block of code**
    - The body of the function is executed ONLY when the function is *called*, not when it is defined.
      Example: `{ echo “this is the body of the function” }`
  - **Parameters**, that are handled the same as command line arguments with the predefined variables $#, $0,...$n. $0 is the name of the function!
    Example: `{ echo “the first parameter is $1” }`
  - Several **calls.** A call is when the name of the function appears with parameters to the function.
    - The call will trigger an **instantiation** of the parameters inside the body of the function, that is, the values of the $1, $2 variables will be *substituted with the parameters.*
    - the function **body will be executed** with the values of the *passed* parameters.
      Example:
      ```
      myfunction param1
      myfunction param2 param3
      myfunction param...
      ```

Functions – example refactored

- Please take your time to look at the *refactored* code for `getsomelines.sh`, `getsomelines_function.sh`:

```
#!/bin/bash

# Function DEFINITION:
# Function that takes in input a filename and a number of lines
# outputs a message about the printed lines
# function NAME
printlinesoffile()
{ # start function BODY
  # the first parameter is a filename
  FILENAME=$1
  # the second parameter is a number of lines
  NUMLINES=$2

  # RESULT is a variable with side effect: the result is stored
  # in a global variable
  # be CAREFUL when to extract the value outside the function!
  # It changes at every function call!
  RESULT=$(cat $FILENAME | head -$NUMLINES)

  # Print out the lines
  echo "First $NUMLINES line(s) of $FILENAME:"
  echo "$RESULT"

} # end of function BODY

# function CALL: put the first two lines of /proc/cpuinfo in CPUINFO
printlinesoffile /proc/cpuinfo 2
CPUINFO=$RESULT

# function CALL: put the first four lines of /proc/meminfo in MEMINFO
printlinesoffile /proc/meminfo 4
MEMINFO=$RESULT

# function CALL: put the first line of /etc/hostname in HOST
printlinesoffile /etc/hostname 1
HOST=$RESULT
```

Side effects

- Mathematical functions only return values.
- A programming language function or procedure not only returns a value, but usually changes the **environment** of the process running. This is usually called a **side effect**.
- The content of `$RESULT` is a *side effect* of the `printlinesoffile()` function as it changes the *environment* of the process at every function *call*

Environment, binding

- **Environment**: All the variable and function names “live” in a space called **environment**. You can think of it **as a table** in the compiler or interpreter memory containing all variable names and their associations with memory chunks.
- **Binding:** A name is said to be **bound** to that environment when its value is associated to a memory index in that environment. In the table below we can see some bindings.
  - Binding can be:
    - **Static**, that is, decided at **compile time**
    - **Dynamic**, that is, decided at **runtime**
      (yes one can change where in the memory that variable is pointing)
- When we **define** a variable or a function, the variable/function name is **added** to the **environment**

Environment	Variable name	Value
global	PWD	Current dir
global	SHELL	Current shell
global	PATH	Executable paths
`cpuinfo.sh`	CPUINFO	First 2 lines of /proc/cpuinfo
`getsomelines_function.sh`	RESULT	18458
`getsomelines_function.sh`	printlinesoffile()	3515

Visibility, scope

- A variable is **visible** in an environment when its binding is present in that environment, that is:
  - There **exists** a variable **name** in the environment
  - That variable name is **associated to a memory location** (this depends on languages)
- Usually a function has its own environment, that is, a set of variables in its own environment, and can see the variables in other environments according to some rules. These rules define the **scope**, or **visibility**, of a variable.
- In the case of BASH, **functions do not have own environment**.

  The scope or visibility of a variable in bash is **limited to a bash instance and all its children**. Let's see some examples.
- In BASH there are two kinds of environment:
  - The **set** environment, which only belongs to a running process;
  - The **export** environment, which is a subset of the set environment which is *exposed* to child processes, or processes run inside the same bash.

The BASH environment: export

Everytime one opens a terminal, the program bash is executed and a **new environment** is created.

1. Open a terminal LXTerm.

2a. Run the **set** command. You'll see all the variables in the current bash session.

Everytime a variable is initialized it ends up in the set environment.

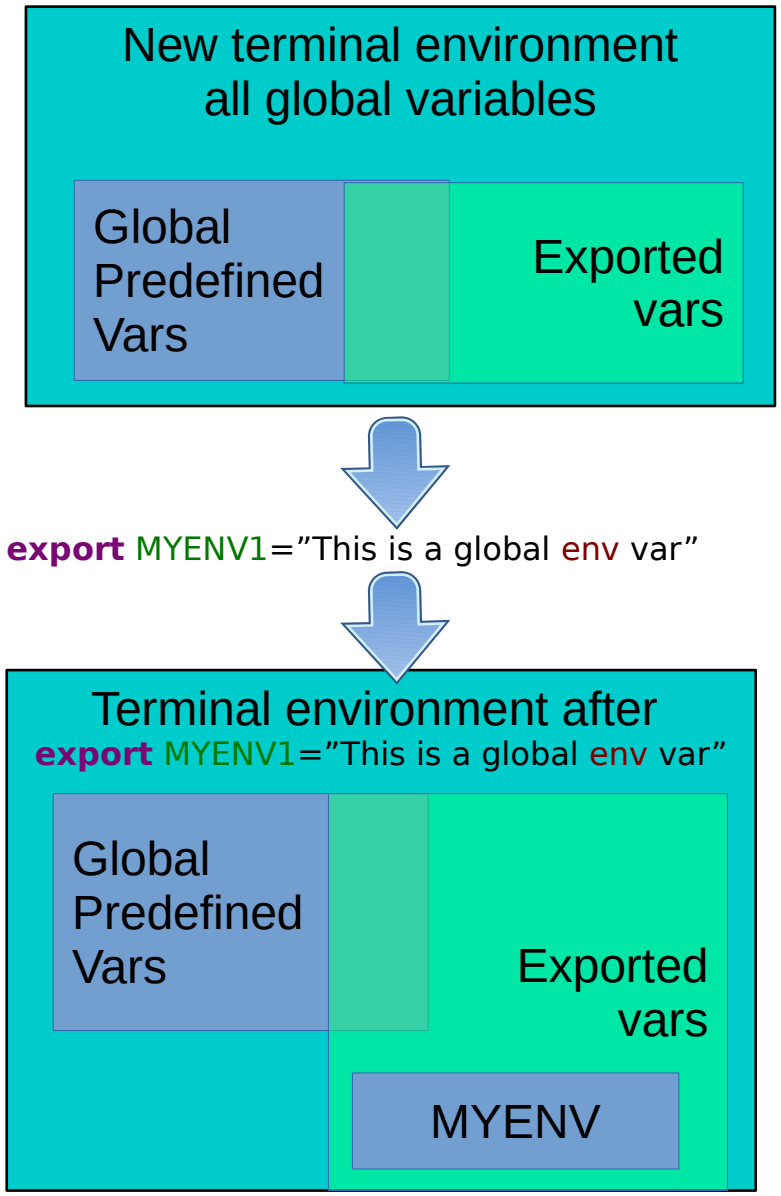
2b. Run the **export** command. You'll see all the environment variables in the current bash session that will be **exported** to any child process.

3. Create and initialize a new **exported** environment variable:

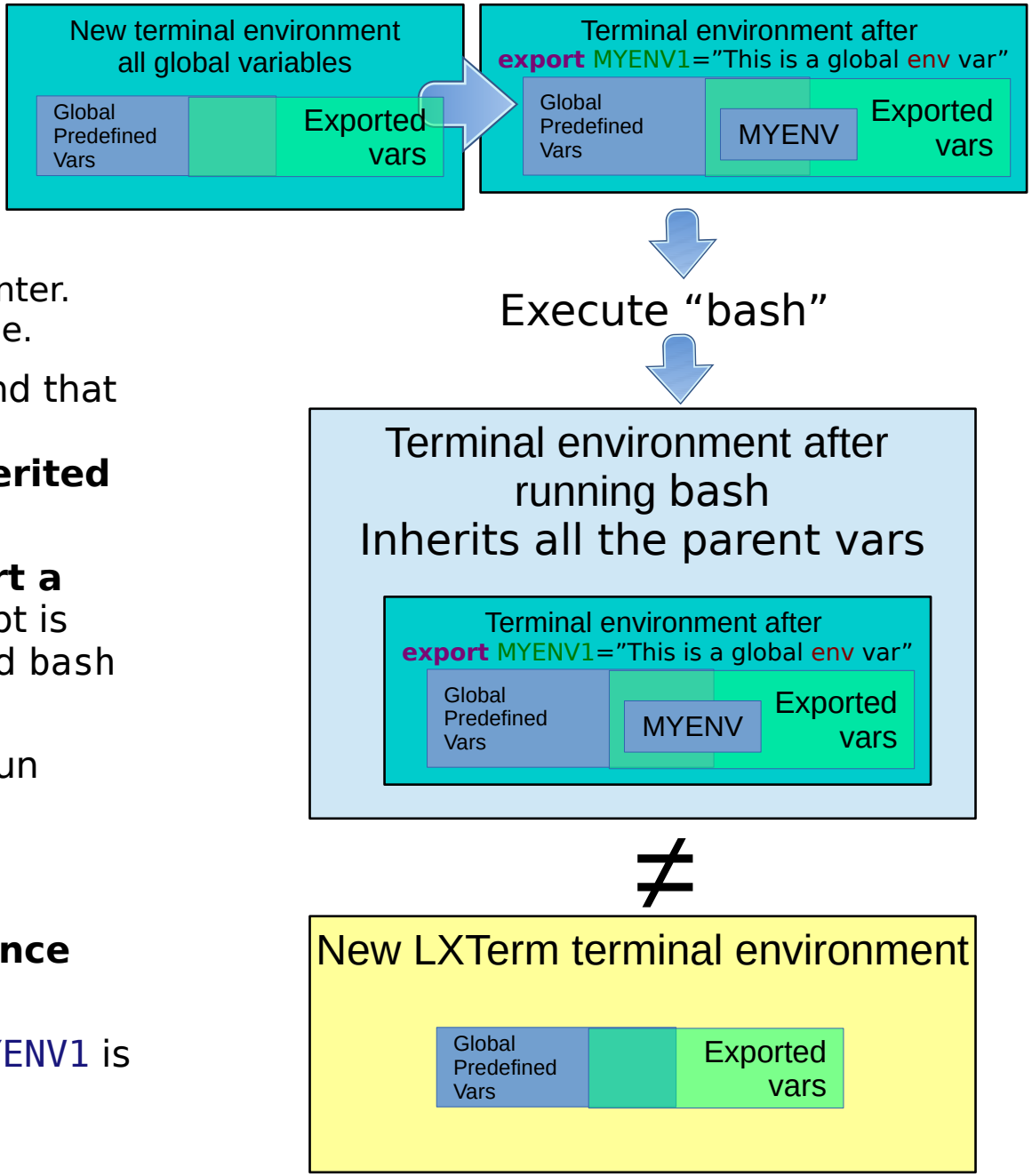
```
export MYENV1="This is a global env var"
```

4. Search for the variable after running **export**, or just print its content with

```
echo $MYENV1
```



The BASH environment: export



5. Now open another bash *instance*:

- Write the command **bash** and press enter. You are now in a new bash command line.
- Run the command **export**. You will find that `MYENV1` is still there.
  The environment is said to be **inherited** from the father process.
- **This happens every time you start a bash script =>** Starting a bash script is equivalent to executing the command bash and then a sequence of commands.

6. Open another terminal *LXTerm* and run

```
export
echo $MYENV1
```

- `MYENV1` should not be there.
  There is no environment inheritance between terminal windows.
- Switch back to the terminal where `MYENV1` is defined.

BASH environment: scope

- **Exercise 3.17:** Consider the bash script `envtest.sh` in the **tutorial folder** with the following content:

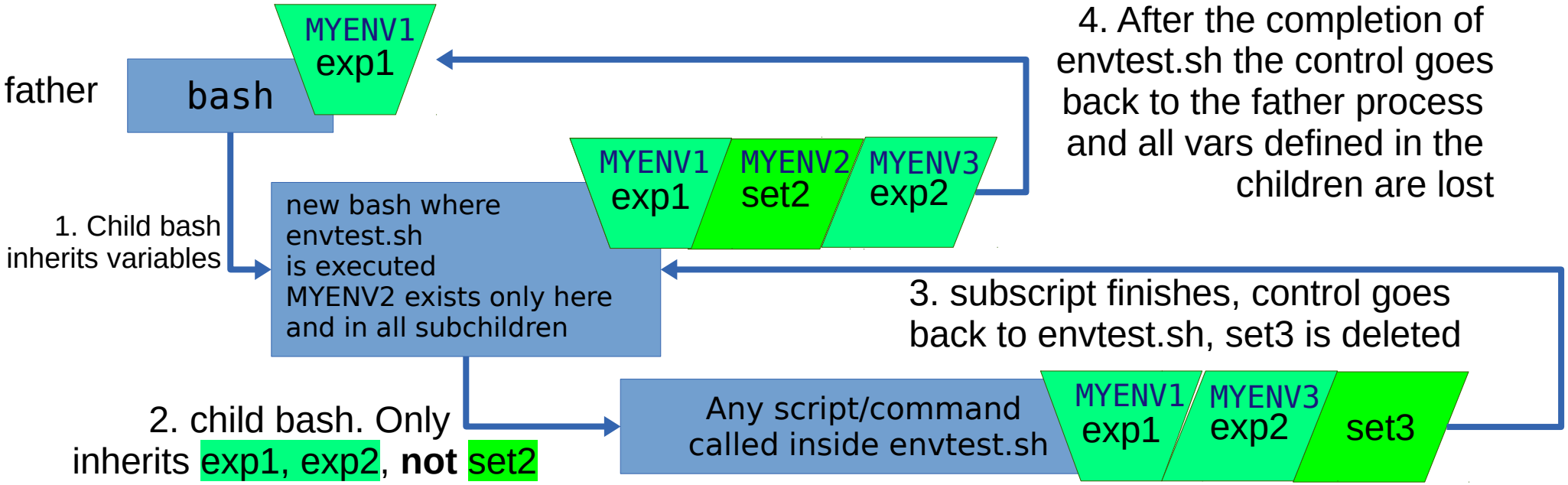
```
#!/bin/bash

# test if an environment variable is defined
if [ "x$MYENV1" == "x" ]; then
        echo "MYENV1 not defined in the environment or empty. Please run"
        echo 'export MYENV1="This is my first environment variable"'

        # I had to comment/remove the next line otherwise sourcing this
        # script will close your terminal if MYENV1 is not defined!
        # Uncomment to try ;)
        #exit 1;
fi

# create an environment variable
MYENV2="This is my second environment variable"

# write the content of the environment vars to screen
echo "Content of MYENV1: $MYENV1"
echo "Content of MYENV2: $MYENV2"

echo "Now check if MYENV2 still exists, with the command"
echo 'echo $MYENV2'
```

BASH environment: scope

- Run it: `./envtest.sh`
- Try to run the commands:

```
echo "Content of MYENV2: $MYENV2"
echo "Content of MYENV3: $MYENV3"
```

- When you ran a script, a new bash instance is generated for the script, that inherits the father environment
- Once the script finishes, all variables defined or exported **inside the script** are **cleared from the environment table** and the control goes back to the father process.
  The "father" environment (where you ran the bash command) DOES NOT inherit from "children" (executed script), but bash scripts executed inside it have their own environment that **inherits** from the father.



Importing an environment

- In bash, there is a command that allows you to copy the environment defined in a script to another script or bash instance, so that it survives the termination of a script. This command is **`source`**
- **Careful! The command also executes EVERYTHING inside the BASH script!**
- If you now try
  - **`source ./envtest.sh`**
  - **`echo`** `“Content of MYENV2: $MYENV2"`
    `echo` `“Content of MYENV3: $MYENV3"`
    You'll see that the output of `export` will contain also `MYENV3`.
    but **not** `MYENV2`, which is only in the **set** or local environment
- If you now write **bash** and run the **export** command again you will **not** see MYENV2 anymore, it is lost in the parent process.
  Only the **exported environment** survives – which is why in bash the exported environment is usually the only one called **the environment.**

Environment summary

- Every new terminal window *LXTerminal* creates a new environment. Environments are **not** shared within terminal windows.
- An initialized variable only “exists” in the environment of the bash instance where it was *initialized*.
- To make sure a variable survives in all script launched inside a bash instance, one must **`export`** it
  - Exported variables are only inherited by child processes and not by parent processes.
- One can **import** the environment that a script generates by using the **`source`** command
  - Remember: do not write any **exit** in code you plan to source!
- The variables in the export environment are commonly called **environment variables.**

Customizing your environment

- When opening a terminal or starting bash, there are a few key files that are processes to *initialize* your shell environment.
- Depending on the distribution and the shell, these may vary. Some are *system* files and you cannot change them, these are processed **first** when opening a shell. But you can override them inside your *user files,* that are processed after the system ones.
- System files:
  - `/etc/profile`
  - All files in `/etc/profile.d/`
  - `/etc/bash.bashrc`
- User files. These are hidden, hence their names starts with a dot.
  You can see them with `ls -a ~`
  - `~/.profile`
  - `~/.bashrc`
  - `~/.bash_profile`
- You can inspect the content of those files using **`cat`**, **`less`** or **`gedit`**. Ask me about things you do not understand.
- IMPORTANT: `.bashrc` should **NEVER** contain code that generates **output** when .bashrc is executed.

Customizing your environment exercise

- We will add an *alias* – kind of a macro or shortcut - to the `cd` command, `cdMNXB01`, that allows us to quickly access the git folder.
- The `alias` command is used for that. Try it and you will see the list of active aliases.
- **Exercise 3.18 – add cdMNXB01 alias**
  - 1. backup your existing .bashrc file:
    - `cp ~/.bashrc ~/bashrc_20190913backup`
  - 2. Open `.bashrc` with *geany*
    - `geany ~/.bashrc &`
  - 3. Add at the end of the file the command:
    - `alias cdMNXB01='cd ~/git/MNXB01-2019'`
  - 4. Import the newly created alias by sourcing the new bashrc:
    - `source ~/.bashrc`
  - 5. It should now appear in the list if you write `alias`
  - 6. Test that you can use the newly added `cdMNXB01` command! It will move you directly to the git folder with the exercises.

Conditions

- Conditions are of different kinds depending on the languages.
  The only condition that BASH can check is whether a command execution terminates successfully.
  - An exit value of **0** is **TRUE (termination successful)**,
    all other values are FALSE (termination unsuccessful).
- The way to specify conditions is as follow:
  - The square bracket **[ ]** or the **test** command can be used.
    Documentation: **man test**
    - Example: **test -e** <filename> checks if a file exists; if the file exists, the predefined variable $? will contain 0, otherwise 1.
    - Try **echo $?** after running a test to see the exit value of the test command.
  - The double square bracket or extended test
    [[<some test command> **]]**
    Documentation: execute **man bash**
    and search for “conditional expression”
    - Example: **[[-e** /etc/services **]]**
  - The double parentheses for arithmetical expansion and logical operations.
    <a> and <b> should be integers.
    ((<a> && <b>))
    Documentation: execute man bash
    and search for “Arithmetic Expansion”
- Tips:

  - to search while in man, type the **/** character followed by a search string and then press Enter.
  - To exit man, use the key **q**
  - To move around use the arrows.

Conditions: Exercises

- **Exercise 3.19**: Execute the following commands:
  - The /etc file exists, so test should exit with no errors
    ```
    test -e /etc
    ```
  - Hence the following should be 0
    ```
    echo $?
    ```
  - This file for sure does not exist! It should put an error in the exit status
    ```
    test -e /thisfiledoesnotexist
    ```
  - What is the exit status now? Should be 1, means error, the file did not exist
    ```
    echo $?
    ```
  - The brackets are equivalent to the above. Try!
    ```
    [ -e /etc ]
    echo $?
    [ -e /thisfiledoesnotexist ]
    echo $?
    ```
  - The double brackets are also equivalent for this case, but they can do also logic and arithmetic evaluation if required, which the others above don't.
    ```
    [[ -e /etc ]]
    echo $?
    [[ -e /doesnotexist ]]
    echo $?
    ```

Conditions: Exercises

- **Exercise 3.21**: Execute the following commands. Do you understand the meaning and results? If not, ask me.
  - `true`
  - `echo $?`
  - `false`
  - `echo $?`
  - Parentheses are Aritmetic Expansion, and the logical operator && is the boolean AND. Check Lecture 3 and remember than in bash 0=true 1=false
    `(( 0 && 0 ))`
  - `echo $?`
  - `(( 1 && 0 ))`
  - `echo $?`
  - `(( 1 && 1 ))`
  - `echo $?`

Control structures

- Enable the machine to **decide** on actions depending on certain **conditions**. (**`if..then...else..fi`**)
- Allow the code to **loop until a certain condition** is met (**`while...do...done`**)
- Allow the code to **loop** for a definite number of times or **over a list** of objects (**`for...do...done`**)

Control structures: if ... then ... else .. fi

- The BASH syntax is as follows:

```
if <condition>; then
    <command1>;[<command2>;…]
else
    <commandA>;[<commandB>;…]
fi
```

Control structures: if ... then ... else .. fi

- `-le` = less than or equal

```
#!/bin/bash
# testif.sh
# run with: ./testif.sh arg1 arg2 arg3
#
# test that at least two arguments are passed to the script

if [[ $# -le 2 ]]; then
   echo "Not enough arguments. Must be at least 3!";
   # exit with error, not zero
   exit 1;
else
   echo "More than 2 arguments. Good!";
   # exit without error, zero
   exit 0;
fi
```

Exit values: the `exit` command

- The `exit` command is used to terminate the program exactly where exit is called, that is, to break cycles and exit the program.
- It takes in input the **return value** of the process:
  - **0** for SUCCES
  - **1** for ERROR
- If you code cannot continue due to an error, you should always **`exit 1`**. Otherwise the code will continue running without the required information. This is useful in your script to detect if a command you run caused an error or did not complete properly.
- You can *check the exit value* by getting the value of the $? variable:

```
echo $?
```

- This works with **any linux program or command**: if there is an error, the process should exit with $? $\neq 0$
- **Exercise 3.22**: check the exit value when you input no argument or three arguments to `./testif.sh` *`[<argument1> <argument2> ...]`*

Control structures: for ... do ... done

- Repeat something for a predefined number of times or for each element in a list.
- Syntax:

```
for <i> in <list>; do
    <command1>;[<command2>;…]
done
```

- The interpreter will substitute <i> with an element in <list> inside the code block **do … done** and execute the code for each element.

Control structures: for ... do ... done

- Print types of files in some directory, default to the /etc directory

```
#!/bin/bash
# listfilestypes.sh
# run with: ./listfilestypes.sh <directory>
#
# Print the argument values
TARGETDIR=$1

# A typical use of IF: if no TARGETDIR defined, then x == x and the expression in brackets will
# be false, so the else branch will be executed and an error message will be shown.
if [ "x$TARGETDIR" == "x" ]; then
     TARGETDIR=~
     MESSAGE="No argument found. Listing filetypes for the $TARGETDIR directory by default"
else
     MESSAGE="Scanning filetypes for the ${TARGETDIR} directory"
fi

echo "$MESSAGE"

# scan all files in TARGETDIR
for somefile in ${TARGETDIR}/*; do
    echo "This is the file $somefile, with type:";
    # the file command tells you the type of a file.
    file $somefile
done
```

Control structures: for ... do ... done

- Print the arguments using different condition approaches

```
#!/bin/bash
# testfor.sh
# run with: ./testfor.sh arg1 arg2 arg3 ...
#
# Print the argument values

echo “Using lists of elements”
index=1          # Reset argument counter
for arg in "$@"; do
    echo "Arg #$index = $arg"
    let "index+=1"
  done             # $@ sees arguments as separate words.


echo  “Using C syntax for the condition”
for ((i=1 ; i <= $# ; i++ )); do
   echo "Argument $i is ${!i}";
done
```

- #$var forces the content of `var` to be a number
- Parameter substitution `${!var}` Gets the **value** of a variable with the name $var instead of `var`

Control structures: while ... do ... done

- Keeps doing something as long as *<condition>* is satisfied.
- Syntax:
  while *<condition>*; **do**
  <command1>;[<command2>;…]
  done
- The code contained inside **do ... done** keeps being executed. It will stop when <condition> is false.

Control structures: while ... do ... done

- Ask the user to enter a variable value (using the read command) until the string end is entered

```
#!/bin/bash
# testwhile.sh
# run with: ./testwhile.sh
#
# Continue asking numbers until the user writes “end”

while [ "$var1" != "end" ]; do     # while test "$var1" != "end"
  echo "Input variable value (end to exit) "
  read var1                        # Not 'read $var1' (why?).
  echo "variable value = $var1"   # Need quotes because of "#" . . .
  # If input is 'end', echoes it here.
  # Does not test for termination condition until top of loop.
echo
done
exit 0
```

Control Structures: Exercises

- **Exercise 3.23**: Change the `iftest.sh` code to complain if the user did not write at least **5** command line arguments
- **Exercise 3.24**: Change the `listfiletypes.sh` code to list the types of files in the folder `/tmp` *by default*, that is, *if no command line argument is passed*.
- **Exercise 3.25**: Change the `testwhile.sh` code to exit when the user writes `bye!`

A bunch of commands you should know about

The “GNU userland”, the collection of commands that are usually shipped with linux, it’s a great collection of command line tools that can do a lot for you. Here I write some that are notable, with links to examples.
They mostly do string operations and can be used to cleanup or reformat a dataset.

- **grep** – find all the occurences of a substring inside a file.
  Example: `grep expressiontofind filename.txt`
  https://www.geeksforgeeks.org/grep-command-in-unixlinux/
- **sed** – substitute strings. The most used form is
  `sed 's/patterntofind/patterntosubstitute/' filename.txt`
  https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/
- **cat** - print a file
  `cat filename.txt`
- **head, tail** – print n lines from the top/bottom of a file, see Tutorial 2 slides
  `head -10 filename.txt ; tail -10 filename.txt`
- **cut** – remove section from each line of a file – can be used to extract columns
  `cut -d, -f5 filename.txt`
  https://www.thegeekstuff.com/2013/06/cut-command-examples/
- **tr** – translate (substitute) characters
  `cat /etc/services | tr s z` (will make every s -> z)
  https://www.thegeekstuff.com/2012/12/linux-tr-command
- **awk** – a powerful line editor that can be programmed for tasks
  https://likegeeks.com/awk-command/
- **curl** and **wget** – programs used to download files
  https://www.keycdn.com/support/popular-curl-examples
  http://www.linuxandubuntu.com/home/12-practical-examples-of-wget-command-on-linux
- **sort** – orders lines of a file give a certain criteria, eventually based on columns or fields
  `sort -r -k2 -h /etc/services`
  https://www.geeksforgeeks.org/sort-command-linuxunix-examples/
- **wc** – bytes, chars and lines counter
  `wc -l /etc/services`
  https://www.tecmint.com/wc-command-examples/

References

- Bash scripting: http://tldp.org/LDP/abs/html/
- Interactive aid: https://explainshell.com

Picture reference (incomplete)

- http://www.shorewatch.co.uk/cruester/images/tabkey.png
-