

Linux (for Robotics)

Carmin Tommaso Recchiuto

Linux



Why Linux for Robotics?

- The focus will be always on robotics developers!

Yes, but why Linux?

- Have you ever heard of ROS? (Yes, probably last week). The Robotics Operating System, at present, **ROS only fully supports Linux systems.**

As a matter of fact, Linux is much more widespread than you may think: Google, Facebook, or any major internet site uses Linux servers. If you are using an Android phone, you are using Linux. 498 out of 5000 world's speediest supercomputers use Linux. And of course, almost all robots uses Linux.

Linux

Versatility

Linux is available under the GNU GPL license, which means it can be freely used on almost any product or service you're developing, as long as the license terms are respected. Also, Linux development is community-based. This means that you can work with other Linux developers to share knowledge and learning.

Security

Linux is one of the most secure operating systems around, from devices/files to programs, access mechanisms, and secure messaging.

Real-time

RTLinux is a hard realtime operating system microkernel. Many RTOS (RTAI,...) are based on the Linux kernel



Linux

While there are hundreds of Linux commands, there are really only a handful that you will use repeatedly. Do you know the Pareto Principle (also known as the 80/20 rule)? For many event, roughly 80% of the effects come from 20% of the causes.

We will focus on learning the fundamentals, the handful of commands and tools that you will use most frequently.

To follow the examples that will be proposed during the course, you should have a linux distribution:

- Native linux OS
- Linux subsystem for windows (WSL) (not recommended)
- Virtual Machine
- Docker!



Linux

Distributions?



A **Linux distribution** (often abbreviated as **distro**) is an operating system made from a software collection that is based upon the Linux kernel

A typical Linux distribution comprises a Linux kernel, GNU tools and libraries, additional software, documentation, a window system (the most common being the X Window System), a window manager, and a desktop environment.

Which one? One of the most popular desktop Linux distribution, and also well compatible with ROS, is **Ubuntu**

In the following, we will always refer to the Ubuntu distribution for the commands description.

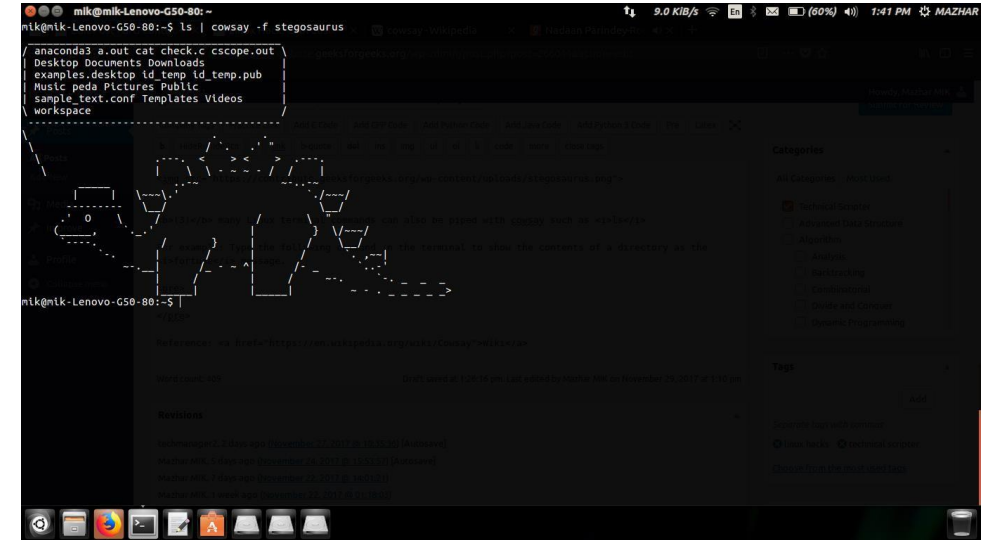
Linux Shell

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

The shell is an environment in which we can run our commands, programs, and shell scripts.

The prompt, **\$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command. It contains some basic information like the current user or the current path you are on.

Es. **\$date**



Linux Shell – apt-get

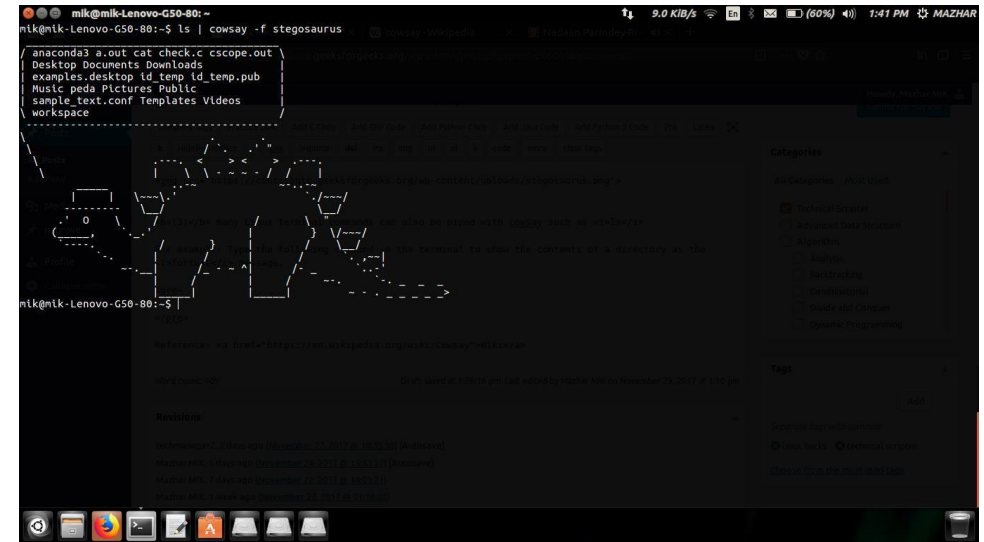
apt-get

APT (Advanced Package Tool) is the command line tool to interact with the Ubuntu packaging system. What is a packaging system??

A packaging system is a way to provide programs and applications for installation. This way, you don't have to build a program from the source code.

apt-get basically works on a database of available packages. If you don't update this database, the system won't know if there are newer packages available or not. In fact, this is the first command you need to run on any Debian-based Linux system after a fresh install.

\$ sudo apt-get update



Superuser privileges

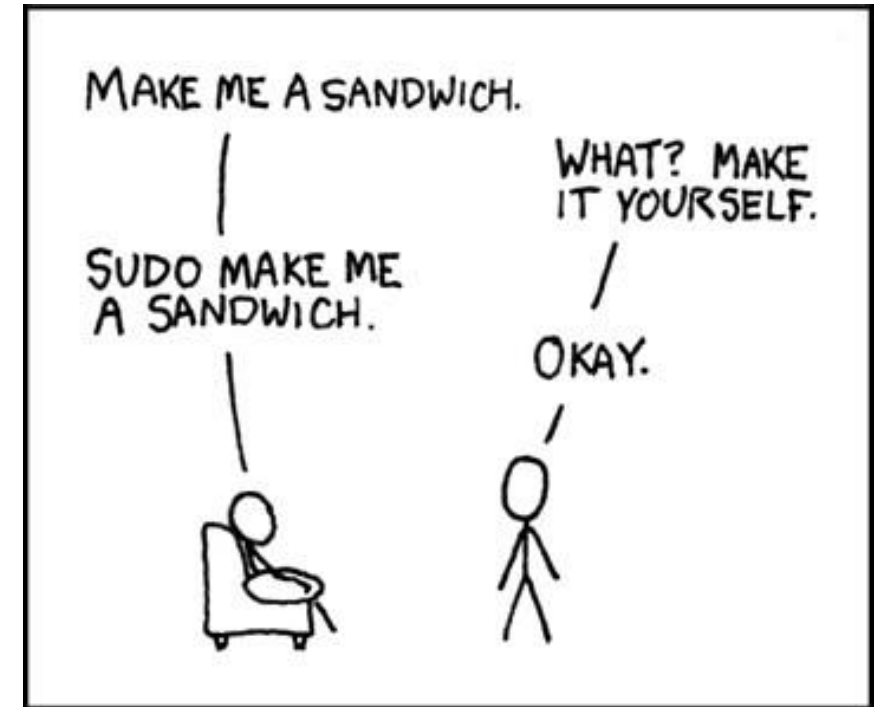
Wait, what is sudo?

It stands for *superuser do*: it allows users to run programs with the security privileges of another user, by default the superuser.

Who is the superuser?

A **superuser** is a special user account used for system administration which is capable of making unrestricted, potentially adverse, system-wide changes. In order to possibly become a “superuser”, the user should be listed in the **/etc/sudoers** file.

By default, **sudo** requires that users authenticate themselves with a password. By default, this is the user's password,



Explore the folder Hierarchy

Let's go back to the Linux shell. As many OS, Linux systems are composed of 2 main elements: **folders** and **files**. The basic difference between the two is that files store data, while folders store files and other folders. The folders, often referred to as directories, are used to organize files on your computer. The folders themselves take up virtually no space on the hard drive.

Let's check out the directory (i.e. **folder**) structure of the **my_ros/src** folder, the workspace folder for ROS. We want to do that using the command `$ tree`

If the program is not installed you can run the command

```
$sudo apt-get install tree
```

Now, how to reach the correct directory?

```
$cd /home/my_ros/src
```

The **cd** command is one of the most important ones in Linux. It allows you to go into a specific directory. When using commands like "cd" you can use the Tab key to autocomplete file and directory names

Explore the folder Hierarchy

Once you reach the correct folder, you may execute the command

\$tree

to get a snapshot of the folder hierarchy

How we move up of one folder?

\$cd ..

How to list all files in a directory?

\$ls

Directories and files will be listed on screen, with different colours depending on the read, write and execution permission.

```
root@193a91ce3203:/home/my_ros/src# tree
.
├── CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
├── ExpRobolutions
│   ├── README.md
│   └── exercisel
│       ├── README.md
│       ├── custom
│       │   ├── CMakeLists.txt
│       │   ├── package.xml
│       │   ├── src
│       │   │   └── PositionServer.cpp
│       │   └── srv
│       │       └── Position.srv
│       └── exercisel
│           ├── CMakeLists.txt
│           ├── package.xml
│           ├── src
│           │   └── exercisel.cpp
│           └── world
│               ├── exercise.world
│               └── uoa_robotics_lab.png
├── custom_messages
│   ├── CMakeLists.txt
│   ├── include
│   │   └── custom_messages
│   ├── msg
│   │   └── Two.msg
│   ├── package.xml
│   ├── src
│   └── srv
│       └── Sum.srv
└── first_package
    ├── CMakeLists.txt
    ├── package.xml
    └── src
        └── example.cpp

16 directories, 19 files
```

Explore the folder Hierarchy

\$ls does not list hidden files. To list hidden files, you should run:

\$ls -a

explicitly lists all the files. You will see a **.** in front of the name of the hidden files.

~

But how to know all the options for a command??

\$man ls or **\$ls --help**

Other must-know things:

\$pwd (it returns the current path)

\$~ (it's equivalent to the home folder of your system)

```
-bash-4.1# cd /home/
-bash-4.1# ls -alh
total 732K
drwxr-xr-x 13 root root 4.0K Sep 7 11:07 .
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 ..
-rw-rw-rw- 1 root root 3.1K Feb 17 2016 api.php
drwxr-xr-x 6 root root 4.0K Sep 7 11:07 app
-rw-rw-rw- 1 root root 2.9K Feb 17 2016 cron.php
-rw-rw-rw- 1 root root 1.7K Feb 17 2016 cron.sh
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 dev
drwxr-xr-x 7 root root 4.0K Sep 7 11:07 downloader
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 errors
-rw-rw-rw- 1 root root 1.2K Feb 17 2016 favicon.ico
-rw-rw-rw- 1 root root 5.9K Feb 17 2016 get.php
-rw-rw-rw- 1 root root 6.3K Feb 17 2016 .htaccess
-rw-rw-rw- 1 root root 5.3K Feb 17 2016 .htaccess.sample
drwxr-xr-x 2 root root 4.0K Sep 7 11:07 includes
-rw-rw-rw- 1 root root 2.6K Feb 17 2016 index.php
-rw-rw-rw- 1 root root 2.3K Feb 17 2016 index.php.sample
-rw-rw-rw- 1 root root 6.4K Feb 17 2016 install.php
drwxr-xr-x 12 root root 4.0K Sep 7 11:07 js
drwxr-xr-x 16 root root 4.0K Sep 7 11:07 lib
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE_AFL.txt
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE.html
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE.txt
-rw-rw-rw- 1 root root 2.2K Feb 17 2016 mage
drwxr-xr-x 6 root root 4.0K Sep 7 11:07 media
-rw-rw-rw- 1 root root 886 Feb 17 2016 php.ini.sample
-rw-rw-rw- 1 root root 577K Feb 17 2016 RELEASE_NOTES.txt
drwxr-xr-x 2 root root 4.0K Sep 7 11:07 shell
drwxr-xr-x 5 root root 4.0K Sep 7 11:07 skin
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 var
-bash-4.1#
```

Create Files and Directories

So, up until this point, you've been learning about some very important tools in order to be able to navigate around any Linux-based machine. In the following section, though, you are going to start learning about some tools that will allow you to actually interact with the system, which basically means that you will be able to modify it.

The **mkdir** command is another essential tool in Linux. It allows you to create a new directory. For instance, try executing the following commands:

```
$cd ~ (it will bring you to the home directory)
$mkdir my_folder
```

Now, if you execute the **ls** command again, you will be able to visualize your new folder.

```
root@193a91ce3203:~# ls -l
total 24
drwxr-xr-x 1 root root 4096 Apr 19 2018 Desktop
drwx----- 1 root root 4096 Jul 2 2019 Downloads
drwxr-xr-x 1 root root 4096 Jun 13 2019 model_editor_models
drwxr-xr-x 2 root root 4096 Sep 28 15:27 my_folder
-rw-r--r-- 1 root root 0 Sep 21 14:18 myfile
root@193a91ce3203:~#
```

Create Files and Directories

On the other hand, you can also create new files. There are several ways in which you can create a new file in Linux. I would say that the most commonly used is with the **touch** command. Go inside the folder you created in the previous section, and create a new file named **my_file.txt**.

```
$cd ~/my_folder  
$touch my_file.txt
```

Now, you will be able to visualize our new file using the **ls** command.

Great! So we have created a new file but...it's empty!

vi visual editor

vi is the default tool in Linux systems for text editing. Of course there are many tools, offering a nice graphical user interface, but there can be some situations (e.g., you are working on a remote robot) in which you may need to modify text using *vi*.

Basically, **vi** has 2 different modes: **Command** and **Insert** modes:

- *Command Mode*: This mode allows you to use commands in order to interact with the file. For instance, go to a certain line of the file, delete certain lines, etc...
- *Insert Mode*: This mode allows you to insert text into the file.

By default, *vi* opens with the **command** mode activated. In order to switch to the **insert** mode, you will have to type the character **i**. After pressing **i**, any character that you type now will be written into the file. Now, type any phrase or word you want into the file, and save it.

...

How to save it?

vi visual editor

You should go back to the **command** mode, by pressing the **esc** key of your keyboard.

Now for example you may delete one character by pressing the **x** key.

In order to save the file, you have to enter the following sequence of characters -> **:wq**.

Now, just type **enter** on your keyboard, and the file will be saved. The character **w** stands for **write** and the character **q** stands for **quit**. So basically, you are telling your file to save and exit.

Now, you can enter your file again and you should see whatever you've written into the file.

What if you want to exit the text editor without saving?

You may only type **:q**, but if you have added some new text to your file, you should explicitly tell the editor to ignore the changes you have done, by entering **:q!**

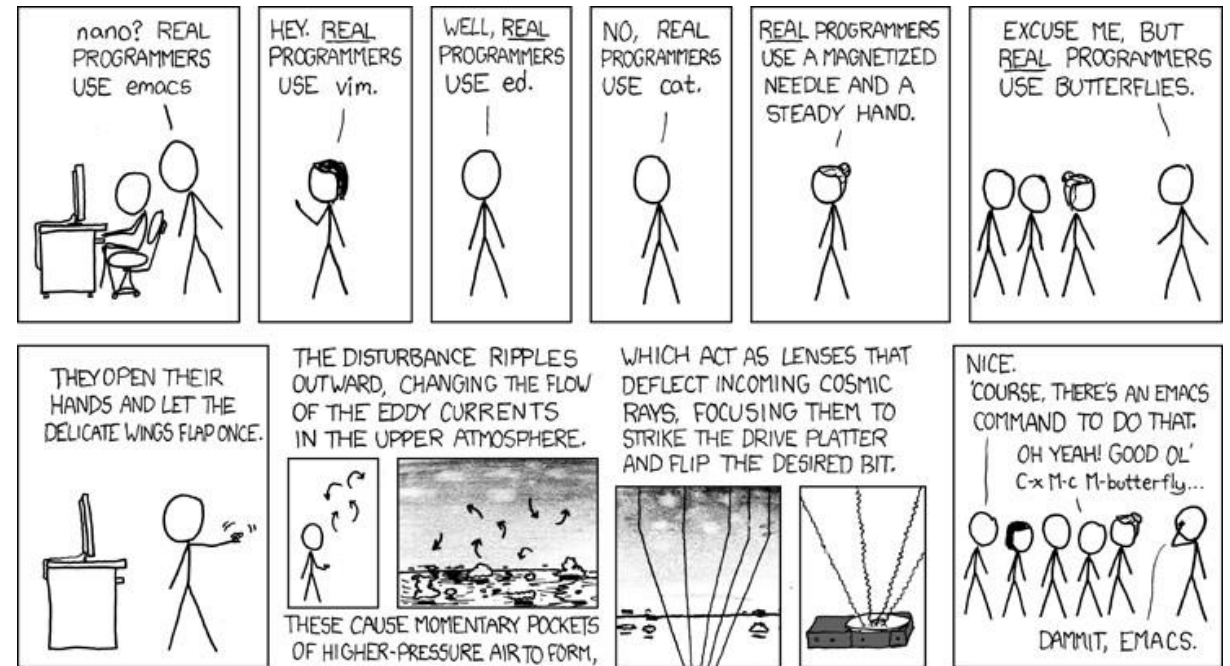
gedit text editor

Don't worry, things may be simpler in most cases.

For example, if you are working in a machine with a graphical user interface and a windows manager, you may use ***gedit***.

\$gedit my_file.txt

Gedit is the default text editor of the GNOME Desktop environment. It includes syntax highlighting for various program code and text markup formats



Commands *move and copy*

The command **mv** in Linux stands for **move**, so it's quite self-explanatory. It allows you to move files or folders from one location to another. For instance, let's try the following commands. First off, let's make sure we are on the directory of the file/folder we want to move.

```
$mv my_file.txt /home/my_ros/my_file.txt
```

In other words, the structure is:

```
mv <file/folder we want to move> <destination>
```

Commands *move and copy*

In Linux systems, the command **cp** stands for **copy**. So basically, it allows you to copy a file or a folder (or multiples) from one location to another one.

```
$cp my_file.txt /root/my_folder/my_file.txt
```

In other words, the structure is:

```
cp <file/folder we want to move> <destination>
```

However, it is NOT possible to copy a folder using the regular **cp** command. In order to copy a folder, you will need to use the **-r** argument.

```
cp -r <folder we want to move> <destination>
```

Remember that you can always run **\$man cp** or **\$cp --help** to list all the available options.

Commands *remove*

In Linux, the **rm** command stands for **remove**. So, you can use it to remove some of the files and folders that you have created:

```
$rm my_file.txt
```

Of course make sure we are in the correct folder.

For removing folders, it works exactly the same as with the **copy** command: you need to add a **-r** flag to the command.

```
$rm -r <folder to remove>
```

Explore permissions

We are going now to explore some advanced utilities that will allow you to interact deeply with a Linux system.

One of these features is related to assigning the correct permissions to file.

Let's go in one folder, and execute the command:

\$ls -la

The **-l** flag allows us to see some basic data related to the files or folders, like the permissions of the files/folders, their creation date/time, etc.

For now, let's just focus on the first part (on the left), These are the **PERMISSIONS** of the file.

```
root@193a91ce3203:/home/my_ros/build# ls -la
total 156
drwxr-xr-x 12 root root 4096 Sep 28 10:13 .
drwxr-xr-x  5 root root 4096 Sep 28 16:44 ..
-rw-r--r--  1 root root   11 Sep 28 10:13 .built_by
-rw-r--r--  1 root root    0 Sep 28 10:13 CATKIN_IGNORE
-rw-r--r--  1 root root 21217 Sep 28 10:13 CMakeCache.txt
drwxr-xr-x  9 root root 4096 Sep 28 10:13 CMakeFiles
-rw-r--r--  1 root root 2317 Sep 24 19:15 CTestConfiguration.ini
-rw-r--r--  1 root root  104 Sep 28 10:13 CTestCustom.cmake
-rw-r--r--  1 root root  397 Sep 28 10:13 CTestTestfile.cmake
drwxr-xr-x  3 root root 4096 Sep 26 00:23 ExpRobolutions
-rw-r--r--  1 root root 55348 Sep 28 10:13 Makefile
drwxr-xr-x  2 root root 4096 Sep 28 10:13 atomic_configure
drwxr-xr-x  3 root root 4096 Sep 24 19:15 catkin
drwxr-xr-x  4 root root 4096 Sep 28 10:13 catkin_generated
-rw-r--r--  1 root root  201 Sep 28 10:13 catkin_make.cache
-rw-r--r--  1 root root 6417 Sep 28 10:13 cmake_install.cmake
drwxr-xr-x  5 root root 4096 Sep 28 10:13 custom_messages
drwxr-xr-x  5 root root 4096 Sep 26 02:09 custom_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 first_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 gtest
drwxr-xr-x  2 root root 4096 Sep 24 19:15 test_results
root@193a91ce3203:/home/my_ros/build#
```

Explore permissions

Each file or directory has 3 permissions types:

read: The Read permission refers to a user's ability to read the contents of the file. It is stated with the character **r**.

write: The Write permission refers to a user's ability to write or modify a file or directory. It is stated with the character **w**.

execute: The Execute permission affects a user's ability to execute a file or view the contents of a directory. It is stated with the character **x**.

On the other hand, each file or directory has three user-based permission groups:

owner: The Owner permissions apply only to the owner of the file or directory, and will not impact the actions of other users. They are represented in the first 3 characters.

group: The Group permissions apply only to the group that has been assigned to the file or directory, and will not affect the actions of other users. They are represented in the middle 3 characters.

all users: The All Users permissions apply to all other users on the system, and this is the permission group that you want to watch the most. They are represented in the last 3 characters.

Explore permissions

So, what does **-rw-r--r--** mean?

The **owner** of the file (in this case, it's us) has **read (r*) and *write (w*) permissions, and the *group** and the **rest of users** have only **read (r)** permissions.

So, if permissions appear with a - symbol, it means that the permissions are not applied.

In this case, this file has no **execution** permissions. And this, if we want to actually execute the file, could be quite a problem. Then... how can we change this?

```
root@193a91ce3203:/home/my_ros/build# ls -la
total 156
drwxr-xr-x 12 root root 4096 Sep 28 10:13 .
drwxr-xr-x  5 root root 4096 Sep 28 16:44 ..
-rw-r--r--  1 root root   11 Sep 28 10:13 .built_by
-rw-r--r--  1 root root    0 Sep 28 10:13 CATKIN_IGNORE
-rw-r--r--  1 root root 21217 Sep 28 10:13 CMakeCache.txt
drwxr-xr-x  9 root root 4096 Sep 28 10:13 CMakeFiles
-rw-r--r--  1 root root 2317 Sep 24 19:15 CTestConfiguration.ini
-rw-r--r--  1 root root  104 Sep 28 10:13 CTestCustom.cmake
-rw-r--r--  1 root root   397 Sep 28 10:13 CTestTestfile.cmake
drwxr-xr-x  3 root root 4096 Sep 26 00:23 ExpRobolutions
-rw-r--r--  1 root root 55348 Sep 28 10:13 Makefile
drwxr-xr-x  2 root root 4096 Sep 28 10:13 atomic_configure
drwxr-xr-x  3 root root 4096 Sep 24 19:15 catkin
drwxr-xr-x  4 root root 4096 Sep 28 10:13 catkin_generated
-rw-r--r--  1 root root  201 Sep 28 10:13 catkin_make.cache
-rw-r--r--  1 root root 6417 Sep 28 10:13 cmake_install.cmake
drwxr-xr-x  5 root root 4096 Sep 28 10:13 custom_messages
drwxr-xr-x  5 root root 4096 Sep 26 02:09 custom_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 first_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 gtest
drwxr-xr-x  2 root root 4096 Sep 24 19:15 test_results
root@193a91ce3203:/home/my_ros/build#
```

Command *chmod*

In Linux, the **chmod** command is used to modify the permissions of a given file or directory (or many of them). There are a couple of ways to use this command, though, so let's go by parts.
Let's first modify **my_file.txt** by writing "echo hello".

Now, let's try the next command:

\$chmod +x my_file.txt

Let's now execute the **ls** command again and see how the permissions have changed.

```
root@193a91ce3203:/home/my_ros# ls -la
total 32
drwxr-xr-x  5 root root 4096 Sep 28 17:49 .
drwxr-xr-x  1 root root 4096 Sep 28 17:26 ..
-rw-r--r--  1 root root   98 Sep 24 19:15 .catkin_workspace
drwxr-xr-x 14 root root 4096 Sep 28 17:46 build
drwxr-xr-x  5 root root 4096 Sep 26 00:23 devel
-rwxr-xr-x  1 root root   13 Sep 28 17:49 my_file.txt
drwxr-xr-x  6 root root 4096 Sep 28 17:42 src
```

Command *chmod*

As you can see, the **execution** permissions, which are represented by the character **x**, have been added to all the permission groups.

What if I only want to modify the permissions for 1 of the groups? Or what if I want to remove a permission instead of assigning it?

chmod <groups to assign the permissions><permissions to assign/remove> <file/folder names>

As for the groups, you can specify them using the following flags:

u: Owner

g: Group

o: Others

a: All users. For all users, you can also leave it blank, as we did in the example command you executed before.

Permissions to assign/remove may be: (+/-) (r/w/x).

Command *chmod*

So, for example:

```
$chmod go-x my_file.txt
```

will remove the permission x (execution), only for the users Groups and Others, of the file my_file.txt.

The same thing can be done using **Binary References**.

Basically, the whole string stating the permissions (*rwxrwxrwx*) is substituted by 3 numbers. The first number represents the Owner permission; the second represents the Group permissions; and the last number represents the permissions for all other users.

Each permission has a number assigned: r = 4, w = 2, x = 1.

Then, you add the numbers to get the integer/number representing the permissions you wish to set.

```
$chmod 740 my_file.txt -> rwxr-----
```

Bash scripts

In one of the previous example, we have executed the file *my_file.txt*

What we have done, it is basically a script: a series of commands within a file capable of being executed

In Linux we have a specific way of writing scripts: **bash scripts**.

A **bash script** is a regular text file that contains a series of commands. These commands are a mixture of commands you would normally type yourself on the command line (such as the ones we have been reviewing, **cd**, **ls**, or **cp**) and also commands we could type on the command line. **Anything you can run normally on the command line can be put into a script and it will do exactly the same thing.** Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

Let's create a *bash scripts*.

Bash scripts

\$touch bash_scripts.sh

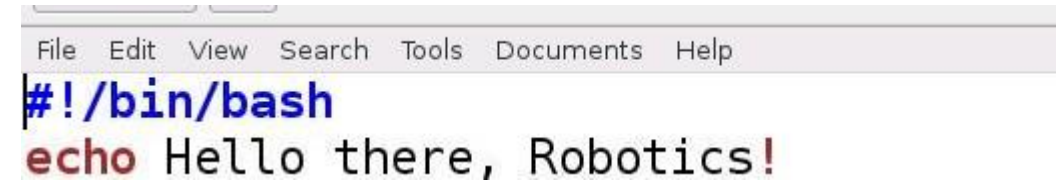
Inside the file, let's place the following contents:

```
#!/bin/bash  
echo Hello there, Robotics!
```

As you can see, the file extension is **.sh**. Usually, this is the extension you will always use when you create a new bash script. Note also that the script starts with the line **#!/bin/bash**. All bash scripts will start with this special line. Basically, it let's the Linux system know that this file is a bash script.

What if I run

\$/bash_scripts.sh?

A screenshot of a terminal window with a menu bar at the top containing 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. The terminal displays two lines of text: the first line is '#!/bin/bash' in blue, and the second line is 'echo Hello there, Robotics!' in red. The text is displayed in a monospaced font with a light gray background.

```
File Edit View Search Tools Documents Help  
#!/bin/bash  
echo Hello there, Robotics!
```

Bash scripts

Yes, remember to fix the permissions!

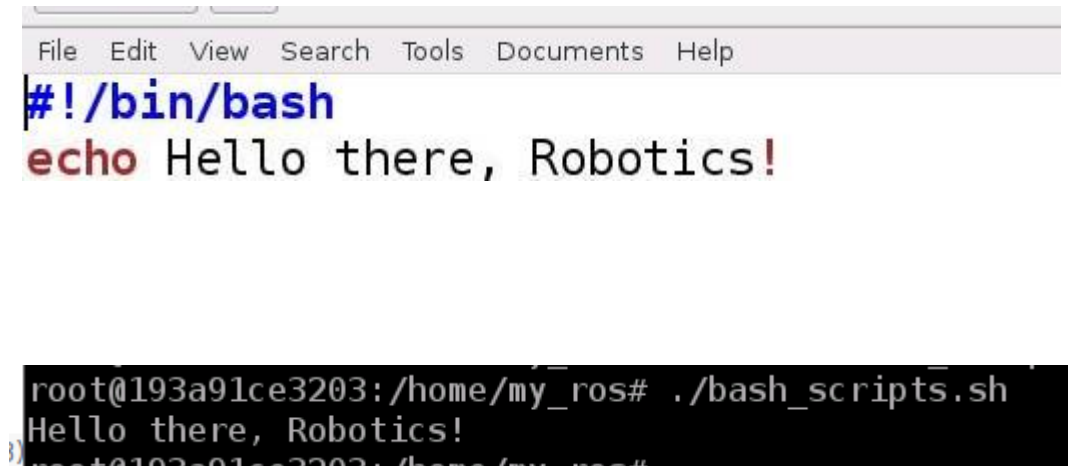
```
$ chmod +x bash_scripts.sh
```

Now you can execute the script!

In a bash script, the command **echo** is used in order to print something on the Shell.

You can also pass parameters to a bash script. This is used when you want your script to perform in a different way, depending on the values of the input parameters (also called arguments).

How?

A screenshot of a terminal window. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. The terminal content shows a blue prompt character followed by '#!/bin/bash' and a red 'echo' command followed by the text 'Hello there, Robotics!'. Below this, a black terminal window shows the command './bash_scripts.sh' being executed, resulting in the output 'Hello there, Robotics!'.

Bash scripts

In fact, it's pretty simple. You can access an argument inside a script using the variables **\$1**, **\$2**, **\$3**, and so on. The variable **\$1** refers to the first argument, **\$2** to the second argument, and **\$3** to the third argument.

So, let's change our script in this way:

```
#!/bin/bash
```

Echo Hello there, \$1!

And run the script:

```
$./bash_scripts.sh students
```

```
INSTALL.SH  
#!/bin/bash  
  
pip install "$1" &  
easy_install "$1" &  
brew install "$1" &  
npm install "$1" &  
yum install "$1" & dnf install "$1" &  
docker run "$1" &  
pkg install "$1" &  
apt-get install "$1" &  
sudo apt-get install "$1" &  
steamcmd +app_update "$1" validate &  
git clone https://github.com/"$1"/"$1" &  
cd "$1";./configure;make;make install &  
curl "$1" | bash &
```

The *.bashrc* script

The **.bashrc** file is a special bash script, which Linux executes whenever a new Shell session is initialized. It contains an assortment of commands, aliases, variables, configuration, settings, and useful functions.

As you may have noticed, the **.bashrc** file is a hidden file. It is automatically generated by the Linux system and it is always placed in the HOME folder (in our case, this is **/home/user/**). However, you can still modify it in order to customize your Shell session.

The **.bashrc** script runs automatically any time you open up a new terminal, window or pane in Linux. However, if you have a terminal window open and want to rerun the **.bashrc** script, you have to use the following command:

\$ source .bashrc

Why is it important for Robotics?

It may be used to set-up the ROS environment, to set ROS workspaces, or to set ***environmental variables***.

Environmental variables

An environment variable is a named value that can affect the way running processes will behave on a computer. For example, a running process can query the value of the *HOME* variable to find the directory structure owned by the user running the process.

Environmental variables may be set by using the command ***export***

We can also run the export command by itself in order to see all the environment variables running.

\$ export

But, there are a lot of variables...

In Linux systems, the **grep** command is used in order to filter elements. For, instance, execute the following command.

\$ export | grep ros

You can always use **grep** to filter results.

Understanding processes

A process refers to a program in execution. Basically, it's a running instance of a program. It is made up of the program instructions, data read from files, other programs or input from a system user.

Basically, there are 2 types of processes in Linux:

Foreground processes : These are initialized and controlled through a terminal session. In other words, there has to be a user connected to the system to start such processes; they haven't started automatically as part of the system functions/services.

Background processes: These are processes not connected to a terminal. This means that they don't expect any user input. Process may be started in background, by adding **&** after the command.

There exist several different commands that will allow you to visualize the running processes on the system. In this course, though, we are going to focus on 2 of them: **top** and **ps**.

Understanding processes

top provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of **processes** or threads which are currently managed by the **Linux Kernel**

It also shows some additional information, such as cpu usage and memory occupation

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26	root	20	0	89100	20120	12964	S	6.3	0.3	6:37.74	x11vnc
22	root	20	0	271368	78640	43028	S	5.0	1.2	4:54.90	Xvfb
12	root	20	0	56184	18436	6904	S	0.3	0.3	0:01.60	python
15	www-data	20	0	125460	3216	1600	S	0.3	0.1	0:01.09	nginx
24	root	20	0	414252	24812	19680	S	0.3	0.4	0:06.81	lxpanel
3705	root	20	0	302220	43052	24804	S	0.3	0.7	0:01.84	terminator
4422	root	20	0	89724	19896	6220	S	0.3	0.3	0:02.04	python
1	root	20	0	4364	632	564	S	0.0	0.0	0:00.34	tini
10	root	20	0	11288	2188	1984	S	0.0	0.0	0:00.00	startup.sh
13	root	20	0	125120	1448	56	S	0.0	0.0	0:00.00	nginx
14	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:13.20	nginx
16	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:00.71	nginx
17	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:00.64	nginx
18	root	20	0	56712	18952	7092	S	0.0	0.3	0:02.88	supervisord
23	root	20	0	624828	26784	20044	S	0.0	0.4	0:02.93	pcmanfm
25	root	20	0	186404	17244	12444	S	0.0	0.3	0:02.76	openbox
27	root	20	0	87432	23324	9816	S	0.0	0.4	0:01.20	python
53	root	20	0	120352	6628	6032	S	0.0	0.1	0:00.00	menu-cached
130	root	20	0	43600	372	12	S	0.0	0.0	0:00.00	dbus-launch
131	root	20	0	42864	3236	2816	S	0.0	0.1	0:00.11	dbus-daemon
141	root	20	0	178532	4612	4156	S	0.0	0.1	0:00.05	dconf-service
3504	root	20	0	301596	43044	24936	S	0.0	0.7	0:02.40	terminator
3508	root	20	0	14872	1704	1556	S	0.0	0.0	0:00.45	gnome-pty-helpe
3513	root	20	0	20792	4508	3188	S	0.0	0.1	0:00.10	bash
3709	root	20	0	14872	1720	1572	S	0.0	0.0	0:00.44	gnome-pty-helpe
3714	root	20	0	20788	4460	3140	S	0.0	0.1	0:00.10	bash
3777	root	20	0	1956436	240696	126012	S	0.0	3.8	0:30.07	firefox
3849	root	20	0	1541092	131240	106472	S	0.0	2.1	0:03.13	Web Content
3918	root	20	0	1498148	104180	81356	S	0.0	1.6	0:07.25	WebExtensions
3961	root	20	0	1473524	73948	59784	S	0.0	1.2	0:00.15	Web Content
4293	root	20	0	640568	39668	29384	S	0.0	0.6	0:08.44	gedit
4429	root	20	0	399776	66192	18708	T	0.0	1.0	0:00.96	python
4445	root	20	0	40460	3520	3072	R	0.0	0.1	0:00.00	top

Understanding processes

ps stands for process status. However, if you execute it, you will only see the processes of the user, which are attached to a terminal.

If you want to see **all** active processes, use the command:

\$ps aux

Consider that you can use the **grep** command to filter results:

Es:

\$ps aux | grep python

```
root@193a91ce3203:~/Downloads/robot-sim# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   4364    632 pts/0    Ss   16:23   0:00 /bin/tini -- /usr/bin/supervisord -n
root        10  0.0  0.0  11288   2188 pts/0    Ss   16:23   0:00 /bin/bash /startup.sh
root        12  0.0  0.2   56184  18436 pts/0    Ss   16:23   0:01 python ./run.py
root        13  0.0  0.0   125120   1448 ?        Ss   16:23   0:00 nginx: master process nginx -c /etc/nginx/nginx.conf
www-data    14  0.0  0.0   125460   3216 ?        Ss   16:23   0:13 nginx: worker process
www-data    15  0.0  0.0   125460   3216 ?        Ss   16:23   0:01 nginx: worker process
www-data    16  0.0  0.0   125460   3216 ?        Ss   16:23   0:00 nginx: worker process
www-data    17  0.0  0.0   125460   3216 ?        Ss   16:23   0:00 nginx: worker process
root        18  0.0  0.2   56712  18952 pts/0    S+   16:23   0:02 /usr/bin/python /usr/bin/supervisord -n
root        22  2.1  1.2  271368  78640 pts/0    Sl   16:23   5:10 /usr/bin/Xvfb :1 -screen 0 1920x969x16
root        23  0.0  0.4   624828  26784 pts/0    Sl   16:23   0:02 /usr/bin/pcmanfm --desktop --profile LXDE
root        24  0.0  0.3   414252  24812 pts/0    Sl   16:23   0:07 /usr/bin/lxpanel --profile LXDE
root        25  0.0  0.2   186404  17244 pts/0    Ss   16:23   0:02 /usr/bin/openbox
root        26  2.9  0.3   89100  20120 pts/0    Ss   16:23   6:58 x11vnc -display :1 -xkb -forever -shared -repeat
root        27  0.0  0.3   87432  23324 pts/0    Ss   16:23   0:01 python /usr/lib/noVNC/utils/websockify/run --web /usr/lib/noVNC 6081 localhost:5900
root        53  0.0  0.1   120352   6628 ?        Sl   16:23   0:00 /usr/lib/menu-cache/menu-cached /root/.cache/menu-cached-:1
root       130  0.0  0.0   43600    372 ?        Ss   16:39   0:00 dbus-launch --autolaunch=40dc81ed31bcebe8a699e4e15ad90669 --binary-syntax --close-std
root       131  0.0  0.0   42864   3236 ?        Ss   16:39   0:00 /usr/bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
root       141  0.0  0.0   178532   4612 ?        Sl   16:39   0:00 /usr/lib/dconf/dconf-service
root      3504  0.0  0.6   301596  43044 pts/0    Sl+  17:48   0:02 /usr/bin/python /usr/bin/terminator
root      3508  0.0  0.0   14872   1704 pts/0    S+   17:48   0:00 gnome-pty-helper
root      3513  0.0  0.0   20792   4508 pts/1    Ss+  17:48   0:00 /bin/bash
root      3705  0.0  0.6   302220  43052 pts/0    Sl+  18:55   0:02 /usr/bin/python /usr/bin/terminator
root      3709  0.0  0.0   14872   1720 pts/0    S+   18:55   0:00 gnome-pty-helper
root      3714  0.0  0.0   20788   4460 pts/2    Ss   18:55   0:00 /bin/bash
root      3777  0.6  3.8  1958484  242724 pts/0    Sl+  18:58   0:30 /usr/lib/firefox/firefox
root      3849  0.0  2.0  1541092  131240 pts/0    Sl+  18:58   0:03 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -prefsLen 1 -prefMapSi
root      3918  0.1  1.6  1498148  104180 pts/0    Sl+  18:58   0:08 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -prefsLen 5866 -prefMap
root      3961  0.0  1.1  1473524  73948 pts/0    Sl+  18:58   0:00 /usr/lib/firefox/firefox -contentproc -childID 3 -isForBrowser -prefsLen 6588 -prefMap
root      4293  0.2  0.6   640568  39668 pts/0    Sl+  19:19   0:08 gedit file:///root/Downloads/robot-sim/test.py
root      4422  0.2  0.3   89724  19896 pts/0    Ss   19:58   0:03 python /usr/lib/noVNC/utils/websockify/run --web /usr/lib/noVNC 6081 localhost:5900
root      4429  0.1  1.0   399776  66192 pts/2    Tl   20:11   0:00 python run.py -c games/abc.yaml test.py
root      4456  0.0  0.0   36128   3196 pts/2    R+   20:20   0:00 ps aux
```

Killing processes

Ctrl + C is used to kill a process with the signal **SIGINT**, and can be intercepted by a program executed in a shell, so that it can clean itself up before exiting (in our case, stop the robot), or not exit at all. It depends on how the application is built.

Ctrl + Z is used for suspending a process by sending it the signal **SIGSTOP**, which cannot be intercepted by the program. Basically, it sends **SIGTSTOP** to a **foreground application**, effectively putting it in the background. Thus, the process is still there, running in the background.

How can I stop a process that is running in the background? Well, for this case, Linux provides the **kill** command. It's very easy to use, but you need to know the Process ID (**PID**) of the process you want to terminate. If you check the previous slide, when running **ps**, you get also the list of all PIDs associated to process.

Just execute:

\$ kill <PID_number> to kill the process associated to that PID number.

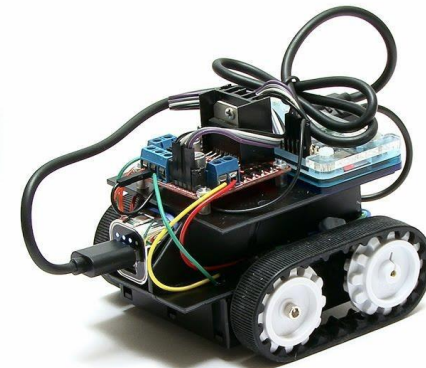
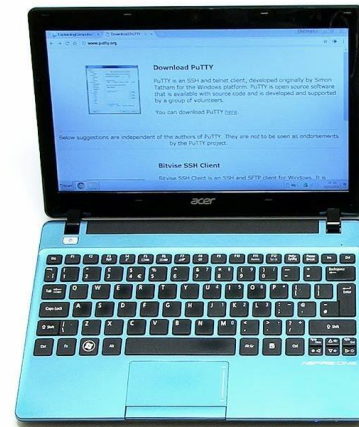
SSH protocol

Secure Shell, most commonly known as **ssh**, is a protocol that allows users to connect to a remote machine in a secure way. It is based on a Client-Server architecture. So, from your local machine (Client), you can log into the remote machine (Server) in order to transfer files between the two machines, execute commands on the remote machine, etc...

In robotics, it is mostly used to access the remote machine that runs in a real (physical) robot from any computer in order to control it and send commands to it.

`$ssh <user>@<host>`

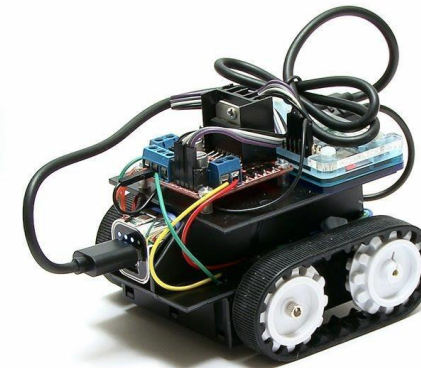
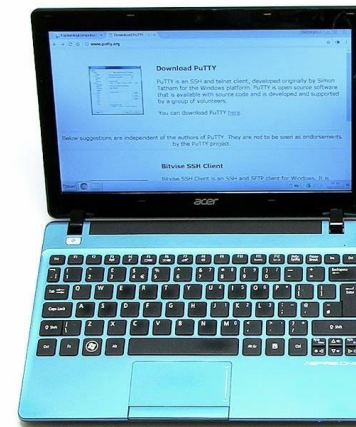
<host> makes reference to the remote machine you want to access (where the SSH Server is running), and **<user>** makes reference to the account in which you want to login from the remote machine.



SSH protocol

Please consider that the host and the client should be in the same LAN, or the host should be in some way accessible from the outside.

Finally, in order to close an ssh session and go back to your local machine, all you have to do is to execute the **exit** command on Shell.



That's all

Of course there are many other commands and things to know about Linux, but these info should be sufficient to start!

Next steps: Python and CPP!



Python (for Robotics)

Carmine Tommaso Recchiuto, Phd

Python

C++ and Python are the most popular languages in robotics. In this course, and probably in most of the course that you will have, you will use these two programming languages. Indeed, ROS can be programmed using Python or C++

If you want to quickly prototype something and don't want the headaches of having errors fire all over your program because you forgot a semicolon, you should use Python.

That's one of the reasons why Python is winning every year more space in the robotics sector. Especially in robotics research, Python is the number one preferred language

Many of the artificial intelligence libraries used for robotics are written in Python: the reinforcement learning algorithms of OpenAI, the learning libraries of Scikit-learn, the OpenCV libraries for image processing, the deep learning of Tensorflow, and many others.

Python

To install python in Ubuntu Linux, just type:

\$sudo apt-get install python python3

You may find out the default version of Python in your system by typing:

\$python (then you should write **exit()** to close the python shell)

```
root@193a91ce3203:~# python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You may have been noticed that there are two python versions: python2 and python3. Although Python 2.7 is still widely used, Python 3 adoption is growing quickly. They have different (sometimes incompatible) libraries, and other differences. Older ROS versions are based on Python2, while ROS Noetic and ROS2 work with Python3. In the following, we will mainly refer to Python3.

Install Pip

Pip is a tool that will help us manage software packages for Python.

Software packages are bundles of code written by someone else that are designed to solve a specific problem. Why write code to solve a specific problem from scratch, when someone else has already written code to solve that exact same problem?

\$sudo apt-get install python3-pip (installing pip for python2 in Ubuntu 20 is a bit more complex).

If at any point in the future you want to install a Python-related package using pip, you can use the following command:

\$pip3 install <package_name>

For example, to install the scientific computing package called [Numpy](#), you would type:

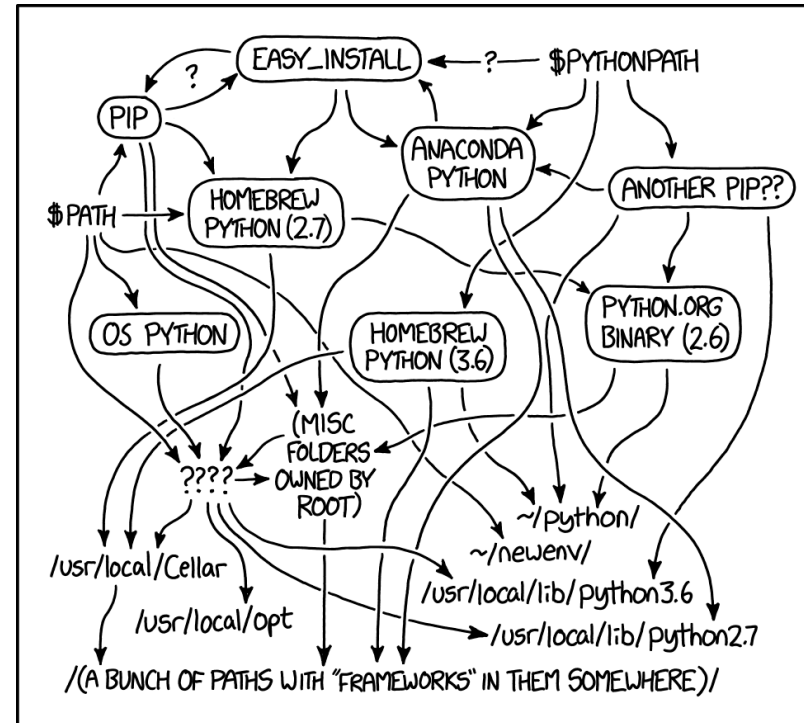
\$pip3 install numpy

Pip

Very important, in some context it can be needed to install a specific version of a package, for compatibility problems. You can still do that with PIP, by running:

\$pip3 install <package_name>==<version_number>

There are actually other package management systems which work well with python, such as conda, easy_install,...



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Create your first program

Let's create our first program. Programs in Python are often called **scripts**. It may happen that sometimes I will say **scripts** and sometimes I will say **programs**. Both words mean the same thing.

Create a new directory named **python_for_robotics**.

```
$mkdir python_for_robotics
```

Move to that directory

```
$cd python_for_robotics
```

Let's create the file **hello_world.py**, and we modify it with gedit.

```
$gedit hello_world.py
```

Create your first program

Here we write the following code:

```
#!/usr/bin/env python
```

```
print("Hello, World!")
```

The first line is the “shebang” line. the program loader takes the presence of the first two characters (**#!**) as an indication that the file is a script, and tries to execute that script using the interpreter specified by the rest of the first line in the file.

The second line is a print statement (please notice that in python2 it will work also without the parentheses).

You can execute the python script by typing **\$python hello_world.py**. Another way to run a program is to make a Python script **executable**.

Open a new terminal window, move to where your hello_world.py program is located, and type:

```
$chmod +x hello_world.py
```

```
$/hello_world.py
```

Create your first program

The same thing can be done with a python shell:

```
$python
```

```
>>> print ("Hello, World")
```

So, a program may be written directly **inside the Python interpreter**: you write a line of code, and the interpreter executes that code, one line at a time.

However, the more common (and comfortable) way to write Python programs is to create Python files (called scripts) containing (lots of) lines of code.

Let's see an example

More complex programs

```
Open ▾   
#!/usr/bin/env python  
  
# Below are the docstring comments that  
# describes what the program is about  
  
'''  
Welcome to the Hello World Program!  
  
This program prints "Hello World".  
  
Usage:  
python hello_world.py  
'''  
  
def main():  
    """  
    Define the main method of the program.  
    """  
    print("Hello World")  
  
# First print the docstring comments  
print(__doc__)  
  
# Run the main method  
main()
```

- In the first line I inform the program about where to find the Python interpreter (**#!/usr/bin/env python**) (shebang)
- Then I wrote some comments to tell what the program does and what command to type in the terminal window to run the code (**python hello_world.py**). Please refer to the slides related to the Software Documentation for more details
- Then I defined a method named **main**. This is where we implement our code.
- The final block of code prints the comments and runs the main method.

Variables

Python is completely **object oriented**, and not "**statically typed**". You do not need to declare variables before using them, or declare their type. Every variable in Python is an **object**.

What does **object-oriented** means? OO is a programming paradigm based on the concept of objects, which can contain data and code: data in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*).

So, if you want to declare a integer variable and assigning the value 5, you just need to type:

```
x = 5
```

Python also allows you to change the **type** of the variable during the program. For instance, you could do something like this:

```
x=5
```

```
x="five"
```


Data Types

In the example before, we have seen numbers and strings. Numbers may be integer or float, and strings are sequence of characters with quotes or double quotes.

We may have also **Lists**, an ordered sequence of items, where all the items of the list do not need to be of the same type. To declare a Python List, you just have to put all the items inside brackets [], and separate each item by commas.

```
l = [1, 2, "hello", 6]
```

At any time, you can access a specific item of the list.

```
print (l[2])  
print (l[0:1])
```

Tuples are similar to lists, but they cannot be updated. Values separated by commas, are enclosed within parentheses ().

```
t = (1, 2, "hello", 6)
```

Data Types

Dictionaries are also similar to lists, in the sense that they contain a list of values and that they can be updated. But the main difference is that items in dictionaries are accessed via keys and not via their position. So basically, dictionaries are a list of items, with each item being a pair made up of a key and a value.

```
dict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
```

```
print (dict["Jon"])
```

```
print (dict["Night King"])
```

```
>>> 25
```

```
>>> 35
```

If-Else statements

When programming robots, we often want a robot to take some action or return some data based on some condition. For example, imagine a security guard robot whose job is to check the age for each person wanting to enter a nightclub. Suppose this robot is naive. Instead of checking the person's ID, the robot asks for the person's age.

We want this robot to allow anybody inside the club who is 18 years old or older but turn away anyone who is under the age of 18. In other words, **if the person is greater than or equal to 18 years old, let them in, otherwise, keep them out.**

Let's try the following code:

If-Else statements

```
#!/usr/bin/env python

# Robot security guard for a nightclub

# Ask for the person's age
age = input("How old are you?: ")

# Make a decision based on the person's age
if int(age) < 18:
    print("You may NOT enter")
    print("Goodbye")
else:
    print("You may enter")
```

Remember that the hash **#** symbol is how we write comments in Python. Everything after the **#** symbol is a comment. When we run the program on our computer, the computer will ignore this line.

The **input** command is Python's way of accepting input from the user (in this case, the person trying to enter the nightclub). We store the user's input inside the variable named **age**.

An if-else statement consists of the **if** keyword, followed by a condition (i.e. age greater than or equal to 18 years old) and a colon

TWO IMPORTANT REMARKS

```
#!/usr/bin/env python

# Robot security guard for a nightclub

# Ask for the person's age
age = input("How old are you?: ")

# Make a decision based on the person's age
if int(age) < 18:
    print("You may NOT enter")
    print("Goodbye")
else:
    print("You may enter")
```

Indentation!

Indentation is very important: In Python, indenting (i.e. leaving whitespace using either the Tab key on your computer or the Space bar) has meaning. Code that is at the same location in the program and is indented the same number of spaces from the left margin will run together.

TWO IMPORTANT REMARKS

```
#!/usr/bin/env python

# Robot security guard for a nightclub

# Ask for the person's age
age = input("How old are you?: ")

# Make a decision based on the person's age
if age < 18:
    print("You may NOT enter")
    print("Goodbye")
else:
    print("You may enter")
```

Be careful with data types!

Try removing the int() in if int(age) < 18

You will get errors if you are comparing integers with strings

If-elif-else statements

```
#!/usr/bin/env python

# Robot security guard for a nightclub

# Ask for the person's age
age = input("How old are you?: ")

# Make a decision based on the person's age
if int(age) < 18:
    print("You may NOT enter")
    print("Goodbye")
elif int(age) >= 18 and int(age) < 21:
    print("You may enter, but you CANNOT drink")
else:
    print("You may enter")
```

If we have more than one condition, you can use the statement **elif**

These statements are useful for enabling a robot to make decisions based on a set of conditions.

FOR Loops

For loops allow the code inside them to be executed repeatedly for a **known** number of times. Let's look at an example.

Imagine we want our security guard robot to walk up and down the hall 5 times. Each time the robot moves, it prints its motion to the screen. One way to do this is by writing the following Python code:

```
print("robot moving right")
print("robot moving left")
print("robot moving right")
print("robot moving left")
print("robot moving right")
print("robot moving left")
print("robot moving right")
print("robot moving left")
print("robot moving right")
print("robot moving left")
```


FOR Loops

... but we can do the same by writing

```
# From 0 to 5 (i.e. run the code 5 times)
```

```
for i in range (0, 5):  
    print("robot moving right")  
    print("robot moving left")
```

The variable *i* will start at 0 and it will be incremented each time the code inside the loop executes.

When *i* will be equal to 5, the loop stops.

You can also use a for loop to move through items in a list:

```
# Create the list
```

```
colors = ["red", "orange", "green", "blue"]
```

```
# Here we are using x instead of i. We can  
# use any character for this variable.
```

```
for x in colors:  
    print(x)
```

WHILE Loops

While loops execute a set of statements until a condition is FALSE

For example, let's say we have a robot that operates in a three-dimensional space. The robot moves around the space and prints its x, y, and z coordinates. **Once the robot gets more than 5 units from the origin (where x=0, y=0, and z=0), in either the x, y, or z direction, the robot stops printing its position.** Let's use a while loop to make this happen.

```
#!/usr/bin/env python
```

```
# The robot begins at the origin (x=0, y=0, and z=0)
```

```
robot_x = 0
```

```
robot_y = 0
```

```
robot_z = 0
```

```
# Print the robot's starting position
```

```
print("Current Position: x=" + str(robot_x) + " y=" +  
      str(robot_y) + " z=" + str(robot_z))
```

```
# Stop executing code once either x, y, or z exceeds 5
```

```
while (robot_x < 5 or robot_y < 5 or robot_z < 5):
```

```
    # Increment the robot's x position by 1
```

```
    robot_x += 1
```

```
    # Increment the robot's y position by 1
```

```
    robot_y += 1
```

```
    # Increment the robot's z position by 1
```

```
    robot_z += 1
```

```
    # Print the robot's current position
```

```
    print("Current Position: x=" + str(robot_x) + " y=" +  
          str(robot_y) + " z=" + str(robot_z))
```

```
print("Mission complete.")
```

Using functions

A **function** in Python is a piece of code designed to perform a specific task. When you are writing big programs with hundreds of lines of code, programs without functions can be really hard to read and debug (i.e. debug means “to find errors”). In this case, you can use functions to group lines of code into separate bundles based on the specific task they are designed to perform.

Functions are really helpful because they allow you to reuse pieces of code rather than typing identical lines of code again and again.

For example, imagine if you had to add multiple pairs of numbers together and print out the sum each time. Let's do this now.

```
#!/usr/bin/env python

# Add 3 and 4 and print the result
a = 3 + 4
print("3 + 4 = " + str(a))

# Add 5 and 7 and print the result
b = 5 + 7
print("5 + 7 = " + str(b))

# Add 2 and 9 and print the result
c = 2 + 9
print("2 + 9 = " + str(c))
```

Using functions

Annoying. What about:

```
def add(number1, number2):  
    sum = number1 + number2  
    print(str(number1) + " + " + str(  
        number2) + " = " + str(sum))  
  
# Define the main function.  
# This method calls the add function above.  
def main():  
    add(3,4)  
    add(5,7)  
    add(2,9)  
    add(25,30)  
  
# When we run the program, this is where  
# the program will start executing code.  
main()
```

This is how usually code is structured in Python. You have the **main** function (remember the first example?), which is executed in the script.

When a function (e.g. add) is called by the **main** function, the code inside that function executes (e.g. numbers are added, and the sum is printed). Once the block of code inside the function executes, program control then goes back to the main program and continues where it left off.

Handling exceptions

You know that sometimes things do not go as planned when you write a piece of software. Your program might stop all of a sudden due to some unforeseen error.

You should anticipate errors that might appear when running a program and then find a way to handle them. Errors in Python are referred to as **exceptions**. Let's see an example related to handling exceptions in Python

```
#!/usr/bin/env python
```

```
# Here we will print the variable named researchtrack
```

```
print(researchtrack)
```

If you run this script, you will get an error, because the variable has never been defined

Handling exceptions

How to handle this situation? **Try-except** statement

```
#!/usr/bin/env python
```

```
try:  
    print(researchtrack)
```

```
except:  
    print("An exception occurred.\nVariable research track is not defined.")
```

When you run the script, the **try** block is the first piece of code that gets executed. If there is an error inside the try block, the **except** (short for “exception”) block of code gets executed. If there is NOT an error inside the try block, program control skips the **except** block of code and moves on to whatever code follows that.

So the two keywords here are **try** and **except**.

- **try** contains the code you want to execute
- **except** contains the code for handling any exception thrown during the execution of the try block.

Classes and Objects

Python uses classes and objects as a way to model things that exist in the real world. We already mentioned that objects in Python are made up of data and functions. Let's be more precise.

An object is actually an instance of a class. Imagine that you want to control a mobile robot. The specific mobile robot that you want to control will be an object of the class **robot**.

So, what is a class?

A class is a template for an object, which enable reuse of code in your Python programs.

Let's make an example. Imagine that you have a number of mobile robots and you want to control the speed of these robots. It would be really tedious and inefficient to create a brand new template for each robot you want to control.

The smarter thing to do is to create a single class (i.e. template or blueprint) called the **robot class**. **Then, when you want to control a specific robot (i.e. object), all you need to do is reuse the class.**

Classes and Objects

```
#!/usr/bin/env python
```

```
# Here is where we define the class named Robot
```

```
class Robot:
```

```
#### Data ####
```

```
# All classes have a function called __init__(), which executes each time you create a new object
```

```
def __init__(self, name, max_speed):
```

```
    self.max_speed=max_speed
```

```
    self.speed=0
```

```
    self.name=name
```

```
#### Functions ####
```

```
# Function to speed up the robot by a certain amount
```

```
def speed_up(self, speed_increase):
```

```
    self.speed += speed_increase
```

```
    if self.speed > self.max_speed:
```

```
        self.speed = self.max_speed
```

```
    print(self.name + " speed is now " + str(self.speed) + " m/s")
```

```
# Function to slow down the car by a certain amount.
```

```
def slow_down(self, speed_decrease):
```

```
    self.speed -= speed_decrease
```

```
    if self.speed < -self.max_speed:
```

```
        self.speed = -self.max_speed
```

```
    print(self.name + " speed is now " + str(self.speed) + " m/s")
```

```
# Here is where we define the main function of the program.
```

```
def main():
```

```
    # The syntax below shows how to create an object of a class.
```

```
    robot_1 = Robot("robot_1", 10)
```

```
    robot_2 = Robot("robot_2", 20)
```

```
    robot_3 = Robot("robot_3", 5)
```

```
    # The syntax below shows how to call a function from an object.
```

```
    robot_1.speed_up(20)
```

```
    robot_2.speed_up(20)
```

```
    robot_3.speed_up(2)
```

```
    # Slow down robot 3 by 10 m/s
```

```
    robot_3.slow_down(10)
```

```
    # Run the main function.
```

```
    # Code execution starts (and ends) here.
```

```
    main()
```


Classes and Objects

```
#!/usr/bin/env python
```

```
# Here is where we define the class named Robot
```

```
class Robot:
```

```
#### Data ####
```

```
# All classes have a function called __init__(), which executes each time you create a new object
```

```
def __init__(self, name, max_speed):
    self.max_speed=max_speed
    self.speed=0
    self.name=name
```

```
#### Functions ####
```

```
# Function to speed up the robot by a certain amount
```

```
def speed_up(self, speed_increase):
    self.speed += speed_increase
    if self.speed > self.max_speed:
        self.speed = self.max_speed
    print(self.name + " speed is now " + str(self.speed) + " m/s")
```

```
# Function to slow down the car by a certain amount.
```

```
def slow_down(self, speed_decrease):
    self.speed -= speed_decrease
    if self.speed < -self.max_speed:
        self.speed = -self.max_speed
    print(self.name + " speed is now " + str(self.speed) + " m/s")
```

The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

The class **Robot** has three **attributes** (data), among which **max_speed** and **name** are set in the init function and may be used for having different instance of the class.

The class has also two methods (functions): **speed_up** and **slow_down**, which acts on the attribute speed.

Classes and Objects

We have then the main function of the program, which create three objects of the class Robot, and then modifies the speed of the three robots.

The code execution starts and ends with the **main()** function

```
# Here is where we define the main function of the program.
```

```
def main():
```

```
# The syntax below shows how to create an object of a class.
```

```
robot_1 = Robot("robot_1", 10)
```

```
robot_2 = Robot("robot_2", 20)
```

```
robot_3 = Robot("robot_3", 5)
```

```
# The syntax below shows how to call a function from an object.
```

```
robot_1.speed_up(20)
```

```
robot_2.speed_up(20)
```

```
robot_3.speed_up(2)
```

```
# Slow down robot 3 by 10 m/s
```

```
robot_3.slow_down(10)
```

```
# Run the main function.
```

```
# Code execution starts (and ends) here.
```

```
main()
```

Working with Files

When you build a robotics project, you will often have to work with files. For example, imagine an underwater robot that needs to take periodic measurements of the water temperature. You might want to record all the information from the temperature sensor inside a text file.

Open(), write(), read(), close()

When we open a file, the newly created file is stored inside a variable, which is a file object. We can then use the methods `write()`, `read()`, `close()`.

```
f = open("myfile.txt", "w")

# We write this text to the text file
f.write("Hello!")

# Close the text file
f.close()

# Append (i.e. 'a') text to the existing text file
f = open("myfile.txt", "a")

# Here is the text we are appending.
# The '\n' symbol means to put the text on a new line.
f.write("\nNow we have some more text in this file!")
f.close()

# Open and read (i.e. 'r') the file
f = open("myfile.txt", "r")
print(f.read())
```

Import

Imports are very useful and common in Python. **Imports** allow you to include in your program the code created in other Python modules. This means that the Python code in your program can execute the Python code defined in another program file, without having to rewrite it.

But what is a module? **A module is simply a file that contains functions that you would like to include in your application.**

Example: I may write another script, which imports the module robot where I have defined the class Robot.

```
#!/usr/bin/env python
```

```
import robot
```

```
def main():
```

```
    robot_1 = robot.Robot("robot_1", 10)  
    robot_2 = robot.Robot("robot_2", 20)  
    robot_3 = robot.Robot("robot_3", 5)
```

```
    robot_1.speed_up(20)  
    robot_2.speed_up(20)  
    robot_3.speed_up(2)
```

```
    robot_3.slow_down(10)
```

```
main()
```

Import

If needed, I can also import only a class of a module, or a function (method)

```
#!/usr/bin/env python

from robot import Robot

def main():
    robot_1 = Robot("robot_1", 10)
    robot_2 = Robot("robot_2", 20)
    robot_3 = Robot("robot_3", 5)

    robot_1.speed_up(20)
    robot_2.speed_up(20)
    robot_3.speed_up(2)

    robot_3.slow_down(10)

main()
```

Import

You may have noticed that the `main()` is executed twice. This happens because when we import the module, the whole script is executed. Is there a way to have a different behavior if the script is imported, or if it is directly executed as a stand-alone piece of code?

Yes, there is...

We can execute the code **only** if the script is directly launched, but we do not execute it if the script is imported, by doing this:

```
if __name__ == '__main__':  
    main()
```

That's all!

Ok, hope you had fun. In the remaining part of the class we are going to practice some of our recently acquired Linux – Python skills!

ONLY IF YOU ARE NOT USING THE DOCKER IMAGE PROVIDED:

Open a shell and execute the following command:

```
$ sudo apt-get update
```

```
$ sudo apt-get install git
```

```
$ git clone https://github.com/CarmineD8/python\_simulator
```

You have downloaded a simple robotic simulator. In order to run it, you should first install:

```
$ sudo apt-get install python-dev python-pip python-pygame python-yaml
```

```
$ sudo pip install pypybox2d
```

IF YOU ARE USING THE DOCKER IMAGE PROVIDED:

Just go to `/Desktop/ResearchTrack1/python_simulator/robot-sim`

Exercise

Indeed, the simulator requires three libraries: pygame, PyPyBox2D, and PyYAML. The easiest way to install these is through your distribution's package manager (but PyPyBox2D is only available through pip).

Now open a terminal shell, move to the *robot-sim* directory and run:

```
$python2 run.py test.py
```

If everything works, you should see a mobile robot and some boxes.

Please follow the instructions at https://github.com/CarmineD8/python_simulator , and try to do the exercises 1 and 2.