



PYCHARM / GIT

Thomas Binetruy



Table of Contents

- ① Abstract
- ② Introduction
- ③ Getting accustomed to PyCharm and Git
 - Setting up the project
 - Our REPL program
 - Gitting our code
 - Making our program interesting
 - Gitting our new modifications
- ④ Making our program non trivial
 - Branching
 - Vector dot product
- ⑤ Conclusion



Abstract

This document demonstrates how to setup a Conda project with PyCharm Community Edition (*free for everybody*). We develop a program sufficiently simple as to not distract us from learning PyCharm but sufficiently complex that using Git will be helpful. We use PyCharm's Git integration to add changes to the index, commit them, see *diffs* between commits, create and checkout branches, and have a visual commit tree among other features. We show the equivalent CLI commands along with their results to convince us that PyCharm's Git integration is working as intended.

Introduction

This document provides an introduction to using PyCharm Community Edition, *which is free*, along with its Git integration. We will create a small command line utility in python that will:

- 1 Return the current time when the user enters `time`;
- 2 Returns vector dot product when user enters two vectors, for example when he enters `dotproduct [1, 2, 3] [4, 5, 6]`.

We'll need to parse our user's input, and decide whether to call the python `time` function, which will require an `import datetime`, or the `numpy.dot` method.

Here's what the program looks like in action:

```
>> help
Here are the available commands:
  help
  time
  dotproduct
>> time
2018-09-23
>> dotproduct [1,2,3] [4,5,6]
32
```

This is a basic **REPL** (Read, Eval, Print, Loop): the program takes a user input, processes it, prints the evaluation's output, and loops so that the user can enter further commands.

Python has a REPL of its own: it's a command line interface (CLI) that waits for Python expressions and prints their outputs before asking for another input.

Your terminal emulator is also a REP: it reads an input, evaluates it, prints its output, and loops.

Getting accustomed to PyCharm and Git

● Setting up the project

- ① **Download** PyCharm Community Edition for your platform, it's *free* for everybody;
- ② Open PyCharm;
- ③ Click on Create New Project and call it `tuto-1` if it's the first time opening PyCharm. Otherwise, `File -> New Project` in the toolbar the the top of your window;
- ④ Make sure you pick the proper interpreter, choosing Conda will configure everything so that scientific packages such as *Numpy* will be preinstalled;
- ⑤ `File -> New -> Python File`, name it `repl.py`.

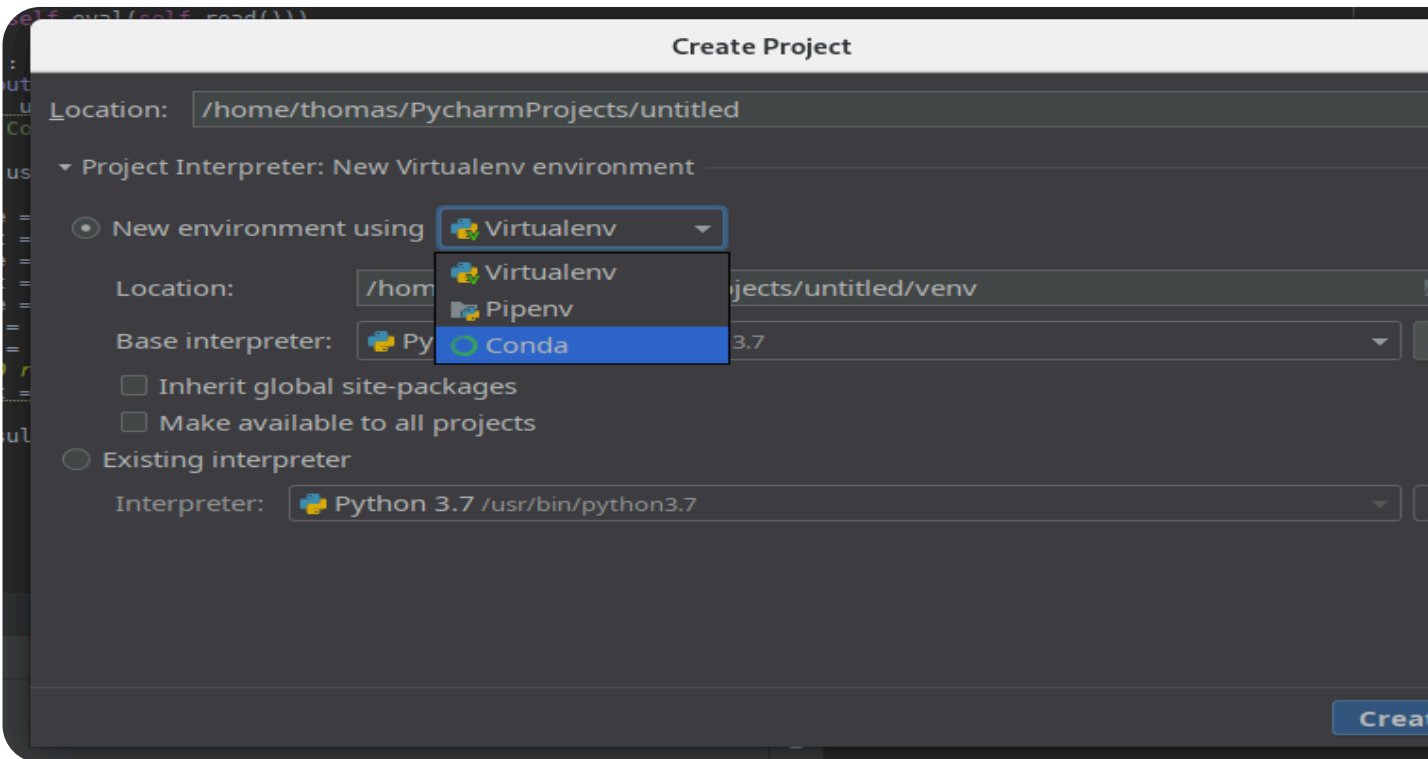


Figure 1: PyCharm's create project popup

Our REPL program

Copy this code into `repl.py` to get us started:

```
import datetime

class REPL:
    def __init__(self):
        print('Press Ctrl-C Ctrl-C or Ctrl-D to quit')

        while 1:
            print(self.eval(self.read()))

    def read(self):
        return input('>> ')
    def eval(self, user_input):
        return user_input
```

```
REPL()
```

● Executing the code

We'll see two ways to execute your code from PyCharm. At this point you might not get the difference between both, but you will by the end of this document ;) You'll see that the second way to run our code is actually an incredibly powerful tool for developing in python.

From the terminal

Let's execute this code using the command line to see what it does:

- ① Click on the terminal tab at the bottom right of the PyCharm window;
- ② type `python repl.py` to run the script (`cd` into your project directory if you're not already in it. Use `pwd` to see your current path);
- ③ Type something into the prompt;
- ④ Our REPL should spit it back at user;
- ⑤ Got to stop 3. if you're bored and step 6 otherwise;
- ⑥ Press Ctrl-C to quit.

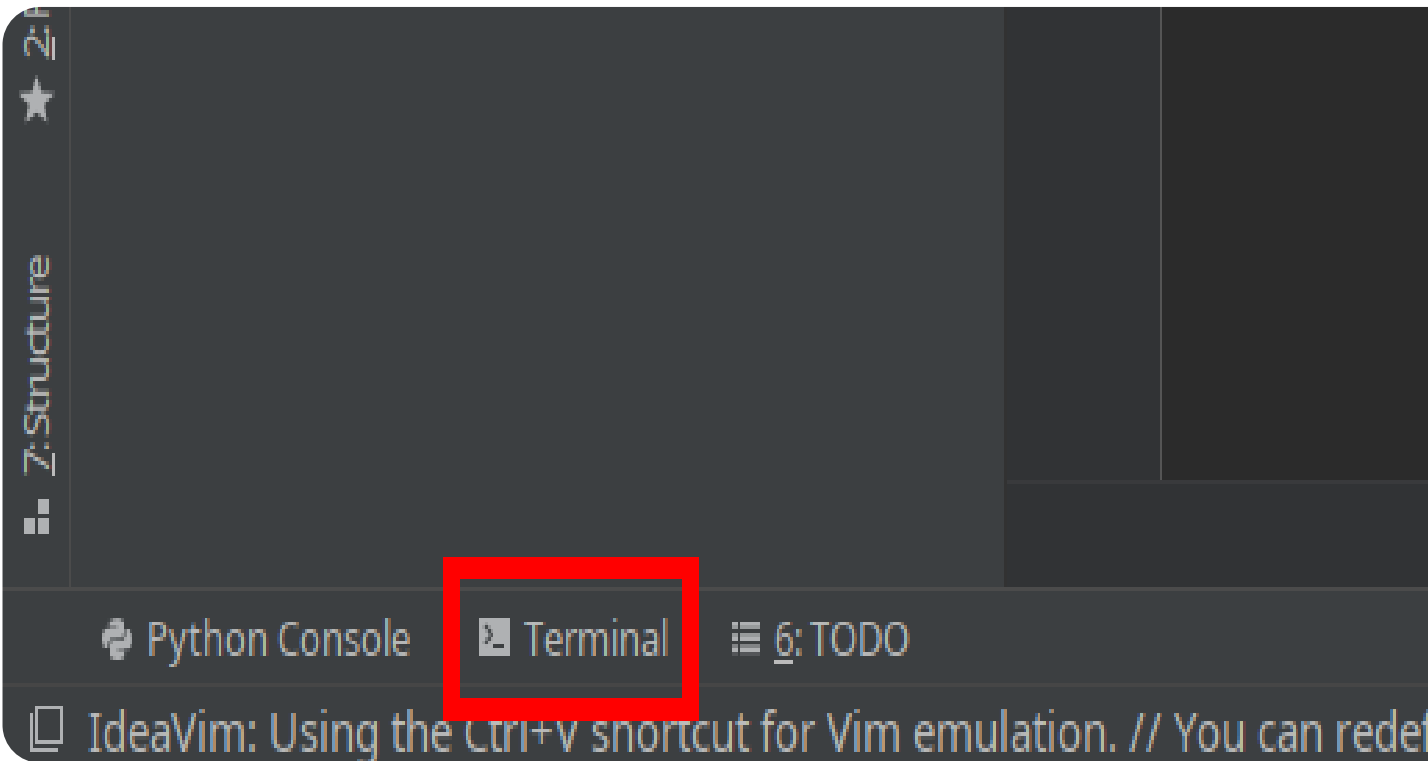


Figure 2: PyCharm's console tab

From the python REPL

We're now going to run our code slightly differently, we're going to send our script to the Python REPL (which is what you get when you type `python` in your terminal). To do this:

- ① Right click on your code and click on `Run 'repl'` ;
- ② It should pup up a REPL tab under your code showing our intro message followed by a prompt.

We'll find out that running our code directly in the REPL makes iterating on Python code very natural, because it allows us quickly inspect variables after our script has ran (since it'll be loaded in memory).

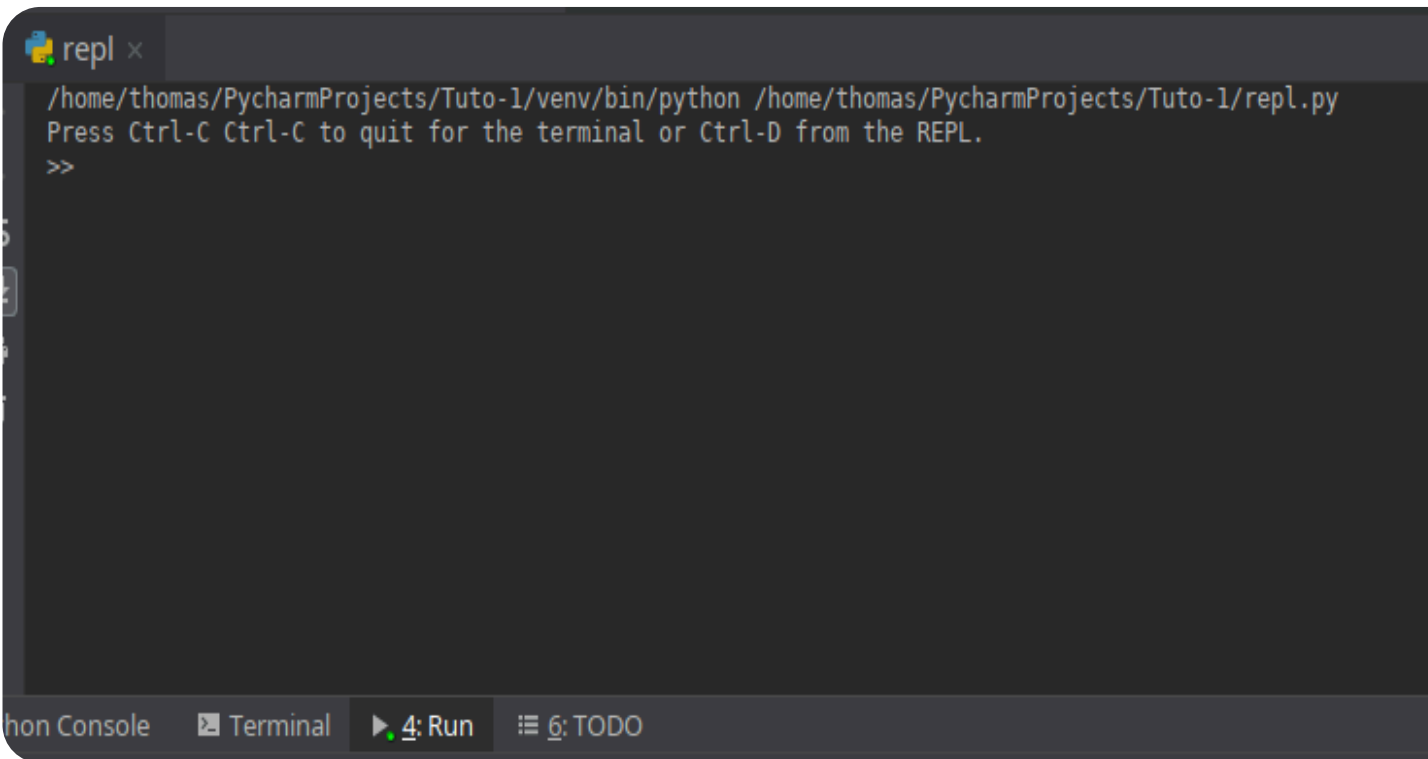


Figure 3: REPL first run

● Gitting our code

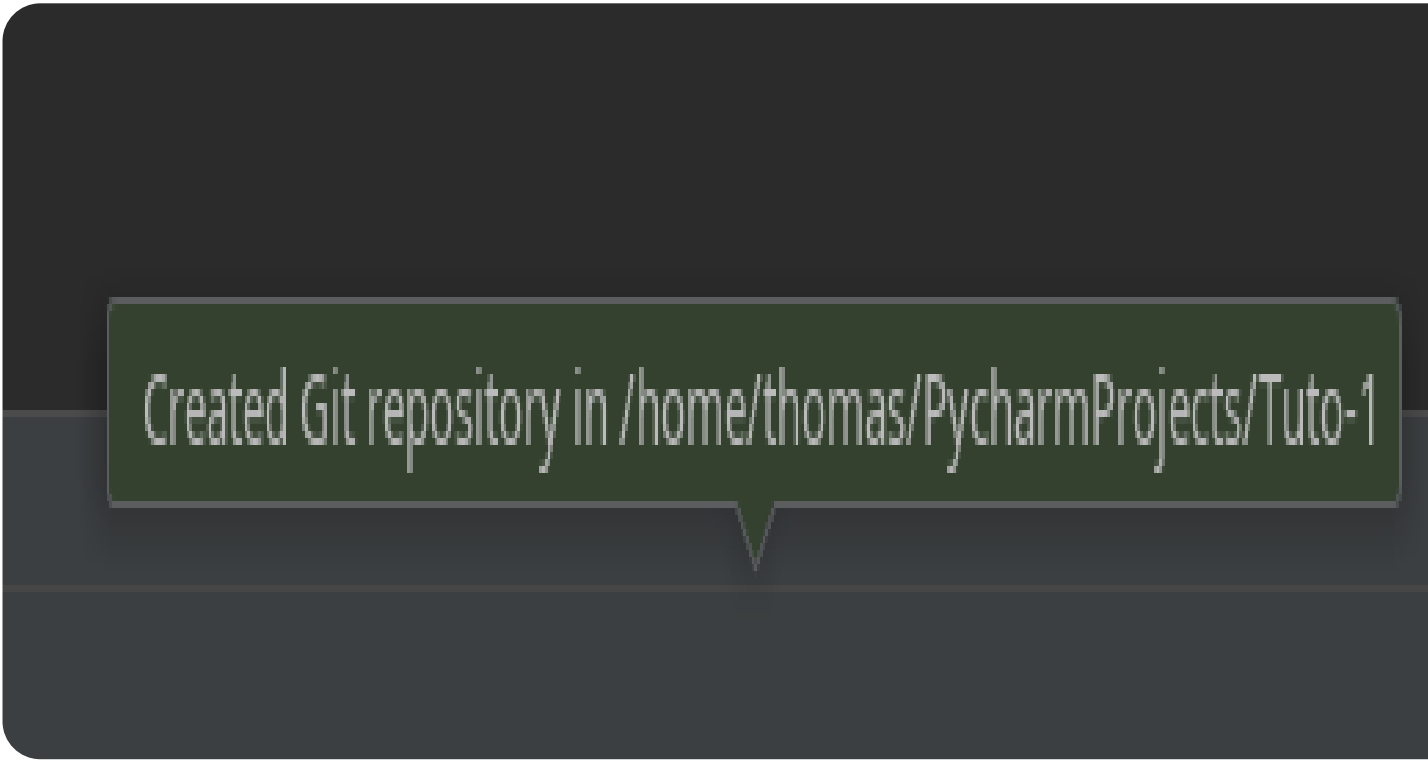
Okay, so now that we have setup our boilerplate code for our script, let's Git it using the PyCharm integration. Make sure you have Git installed on your platform, it's an independent program from PyCharm.

- ① In the top toolbar, click on `VCS -> Enable version control integration -> 'Git' in the dropdown -> OK`. You should see a green message saying "Create Git repository in [project path]";
- ② Now, your file is added to the index to Git status (check by clicking on the terminal tab and typing in `git status`) but we still need to commit our changes;
- ③ On the top right of the PyCharm window should be a box saying "Git" with a blue arrow pointing down and a green checkbox on its right. Click on the checkbox to prepare our commit;
- ④ You should see a popup with the added code in green at the bottom of the popup and an empty commit message textbox. Enter a commit message and click on the commit button



at the bottom of the popup.

You can now open your terminal (just use the one integrated to PyCharm like we've been doing) and type `git status` followed by `git log` to show that our file changes have indeed been added to our Git history!

A dark-themed terminal window with a green speech bubble overlay. The speech bubble contains the text "Created Git repository in /home/thomas/PycharmProjects/Tuto-1".

```
Created Git repository in /home/thomas/PycharmProjects/Tuto-1
```

Figure 4: *Git repository created message*

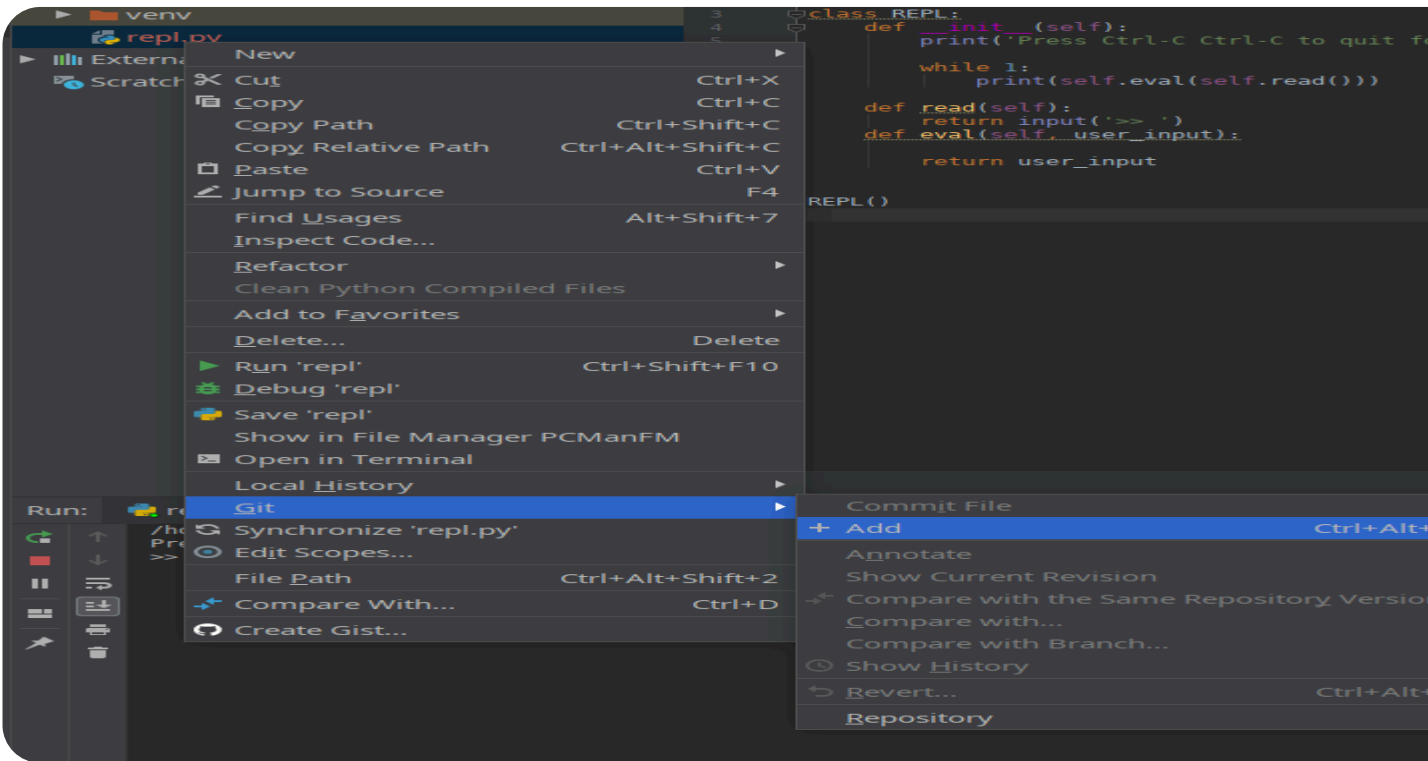


Figure 5: Adding a file to Git in PyCharm



```
Terminal
+ No commits yet
x Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   repl.py

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

      .idea/
      venv/

(venv) [thomas@thomas-linux Tuto-1]$
```

Python Console Terminal 4: Run 6: TODO

1 file committed: Boilerplate code for our script :) (moments ago)

Figure 6: `git status` showing file added in console

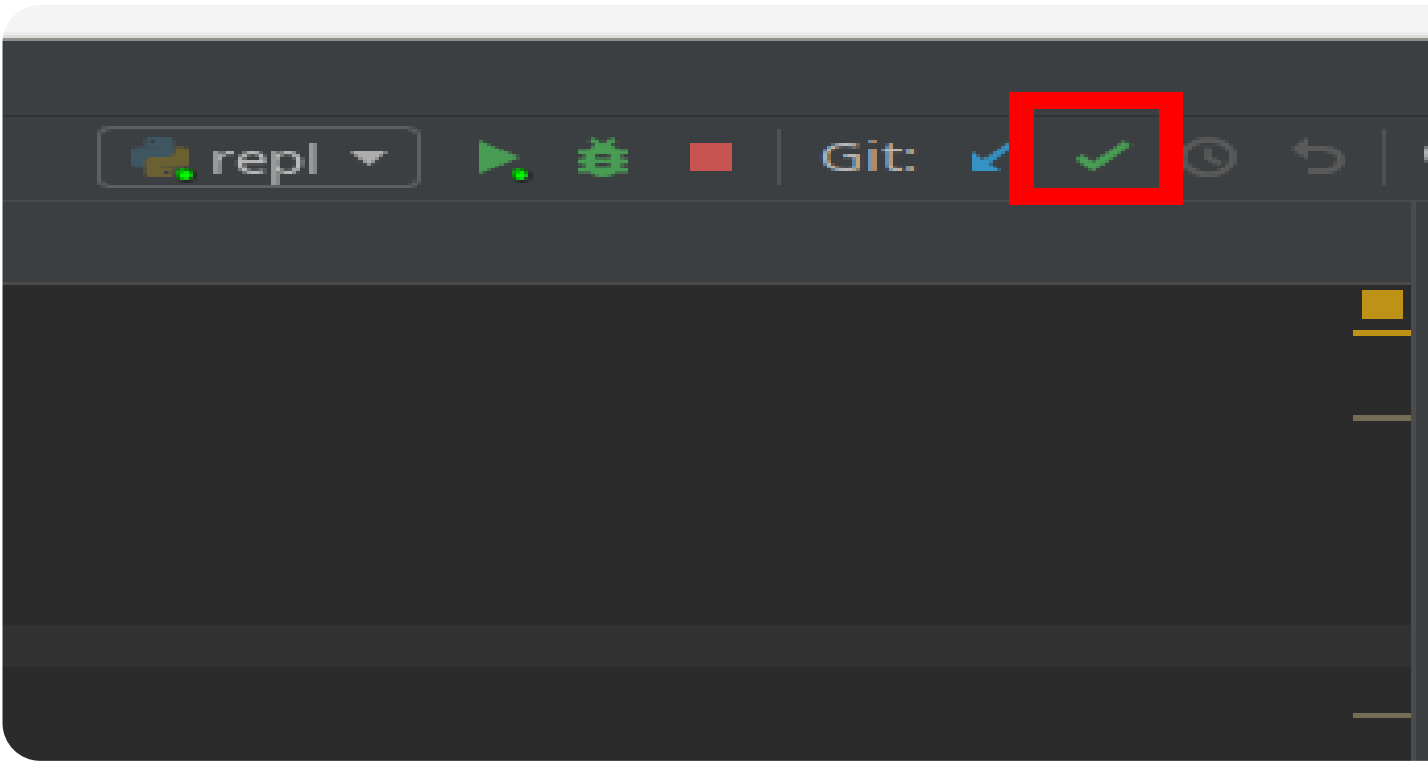


Figure 7: Commit checkbox

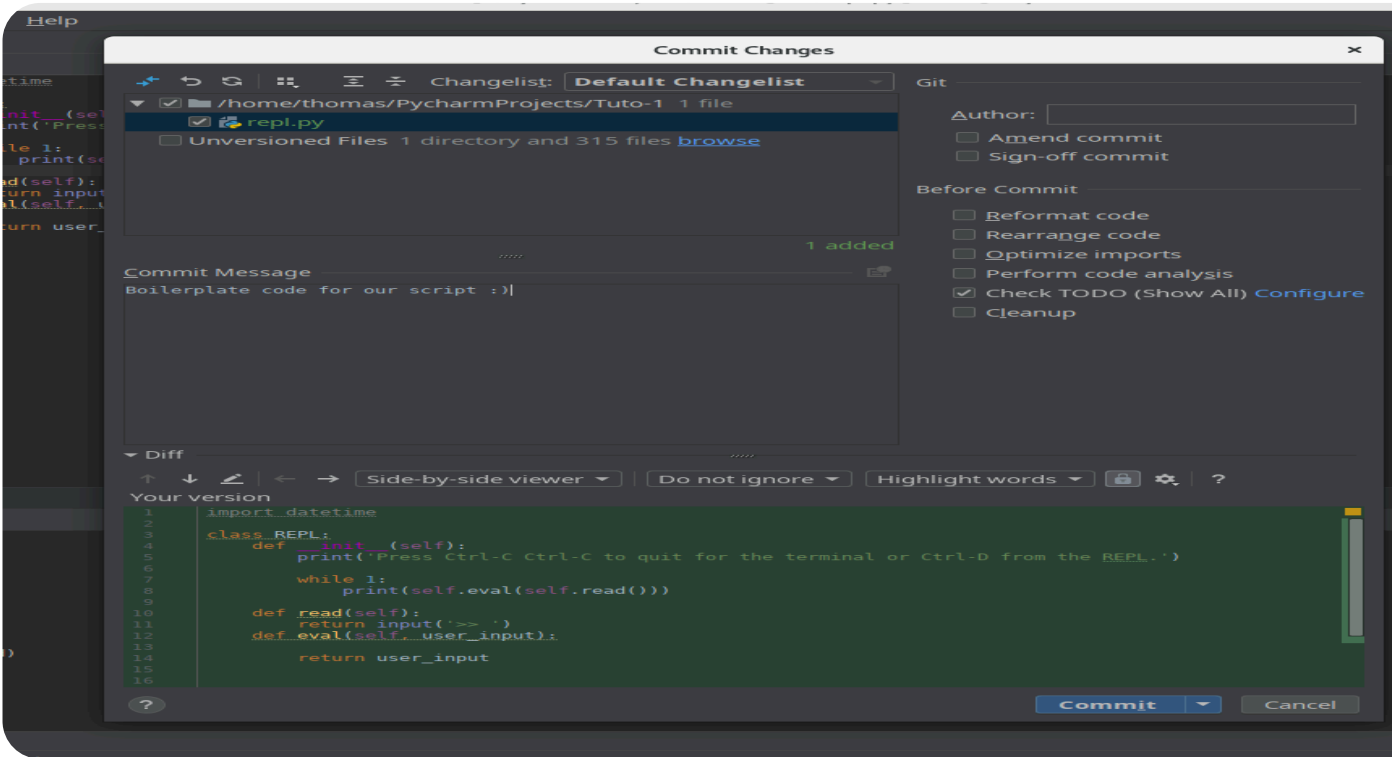


Figure 8: Commit popup for our first commit

```
(venv) [thomas@thomas-linux Tuto-1]$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .idea/
        venv/

nothing added to commit but untracked files present (use "git add" to track)
(venv) [thomas@thomas-linux Tuto-1]$
```

Figure 9: Double checking our commit via the CLI with `git status`

● Making our program interesting

Now that we have all of our boilerplate, let's make our REPL a bit more useful. Let's make it spit the current time when the user enter `time`:

Let's modify our `eval` method so that it looks like that:

```
def eval(self, user_input):
    help_message = 'Here are the available commands: \n help \n time \n dotproduct'
    result = 'Command not recognized. ' + help_message

    if user_input == 'time':
        # Using our datetime import
        result = datetime.date.today()
    if user_input == 'help':
        result = help_message
```



```
return result
```

Now run our file in the repl (remember: right click on the file -> run 'repl') and try to enter `time`, `help`, or anything else.

```
Run: repl x
/home/thomas/PycharmProjects/Tuto-1/venv/bin/python /home/thomas/PycharmProjects/Tuto-1/repl.py
Press Ctrl-C Ctrl-C to quit for the terminal or Ctrl-D from the REPL.
>> time
2018-09-22
>> help
Here are the available commands:
help
time
dotproduct
>> foobar
Command not recognized. Here are the available commands:
help
time
dotproduct
>>
```

Figure 10: REPL second run

● Gitting our new modifications

- ① Right click on our script: `-> Git -> Add`;
- ② Click on green checkbox at the top right of the PyCharm window;
- ③ Enter a descriptive commit message such as "Implemented eval method with time and help commands";

4

Click on commit.

You should see a green message at the bottom of the PyCharm window that showed up.

If you type `git log` in the terminal tab, you'll see that we now have 2 commits!

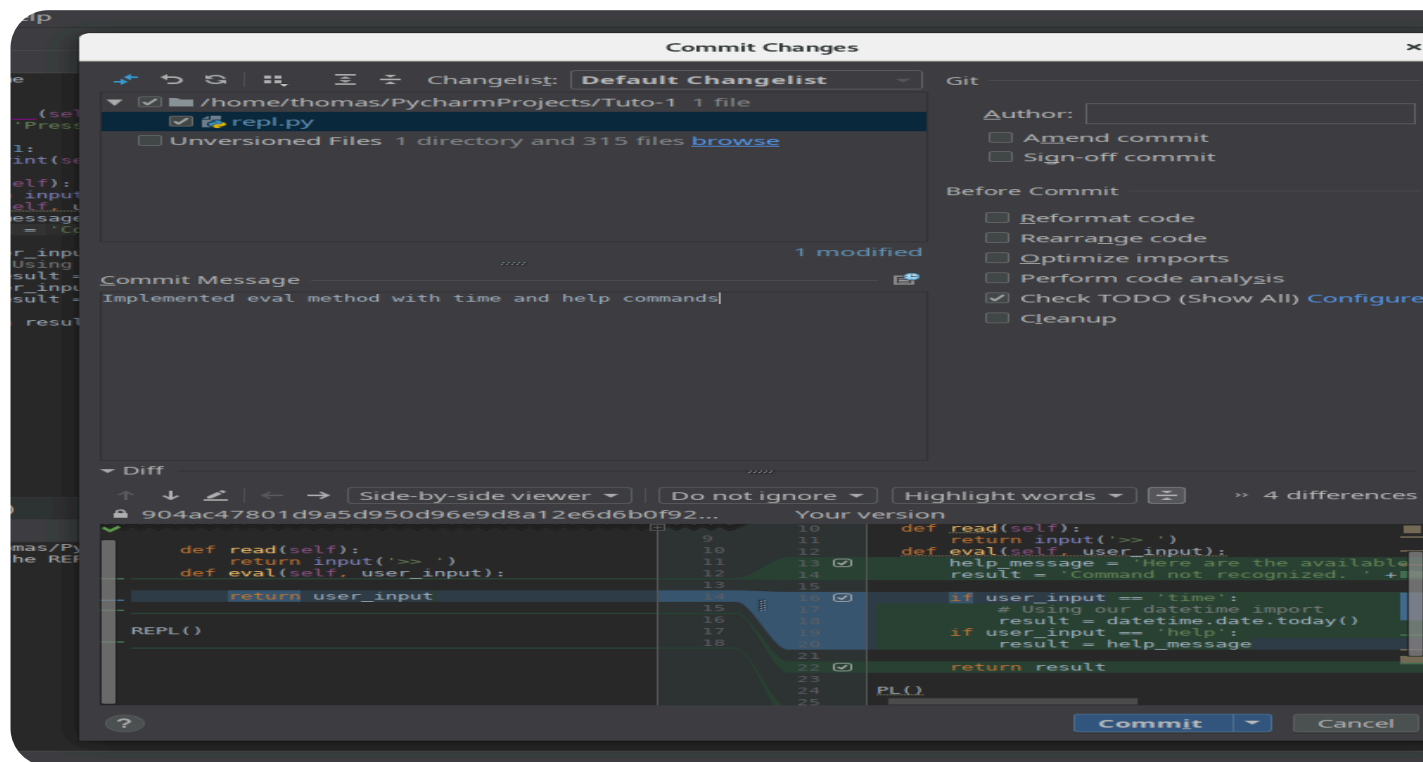


Figure 11: Commit popup for our second commit

Making our program non trivial

With the help of Git, we'll be able to start making our program more complete while also being able to version it properly. We're going to implement the `numpy.dot(vect1, vect2)` method that'll allow to use our REPL to compute the vector dot product between to vectors.



Branching

Say that our crazy REPL program is used by thousands of users on our production system and therefore we absolutely do not want to commit on our *production branch*, which is the one we've been working off of currently. But we still want to be able to use git to track our development work. Thankfully, PyCharm allows us to interface through it to create a new git branch and switch to it - *checking out a branch* in git language: `git checkout [branch_name]`.

- 1 In the top toolbar: `VCS -> VCS Operations Popup... -> Branches -> New Branch`
- 2 Type `dev` in the textbox, *acknowledge the checkbox indicating that you will be switched to this new branch when creating it*;
- 3 Click okay to create our new dev branch and checking out to it.

Now if you open the PyCharm terminal via its tab at the bottom, you can enter `git branch` to see that we have indeed switched to the dev branch (see the little star next to the `dev` branch?).

```
(venv) [thomas@thomas-linux Tuto-1]$ git branch
* dev
  master
(venv) [thomas@thomas-linux Tuto-1]$
```




Python Console  Terminal  4: Run  6: TOD

Figure 12: Git branches through the CLI

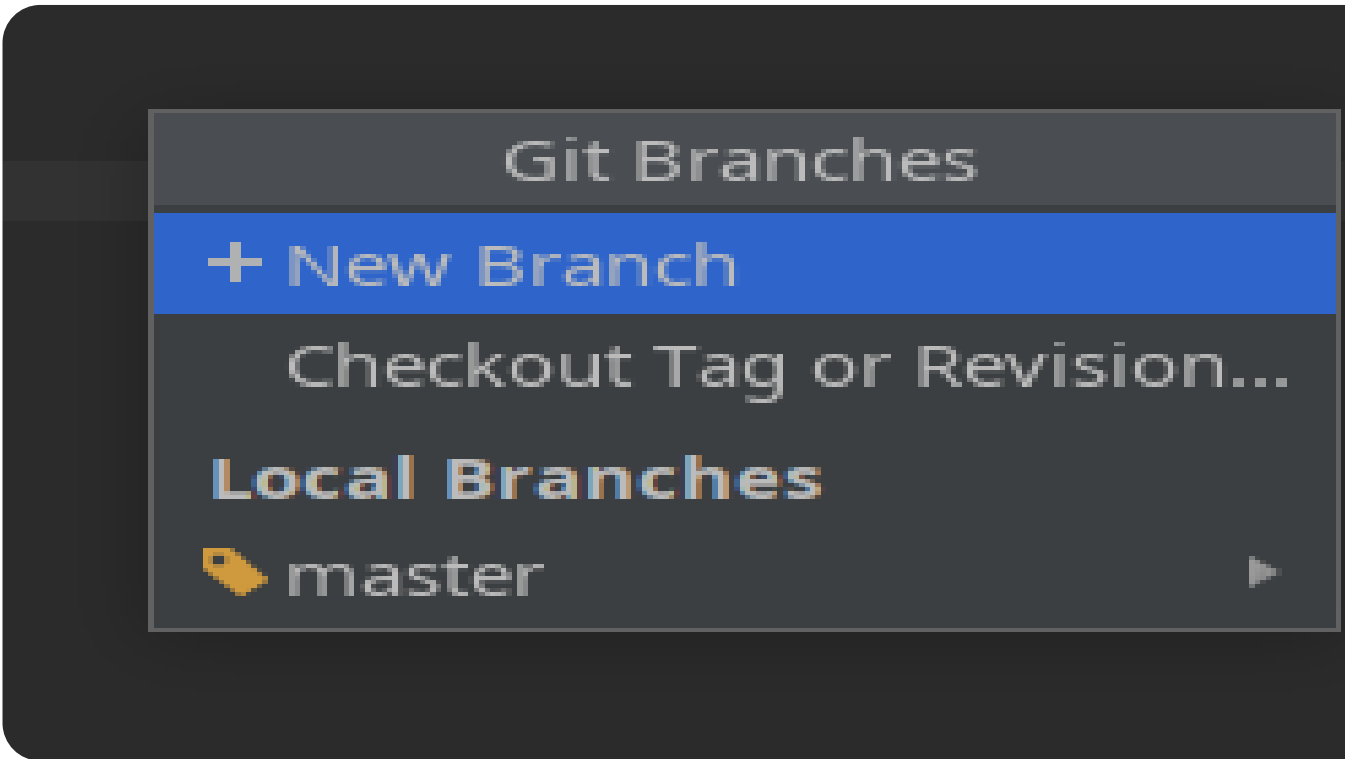


Figure 13: Git branches Pycharm popup

Now that we're on our production branch anymore, we can start messing up the code without having to worry!

• Vector dot product

Let's add our program the ability to do the vector dot product using a special syntax: `dotproduct [vector_1] [vector_2]`, and we'll define our vectors to be python arrays of numbers that **cannot** contain spaces (for now and simplicity's sake, this rabbit hole is for another day), for instance `[1,2,3,100]`.

The reason for this *and quite frankly, very poor* design decision is that we'll be able to split our input string by spaces and then construct a string that we'll feed to python's `eval` function. The idea is that we get our program behave as such:

```
>> dotproduct [1,2,3,4] [1,2,3,4]
```

```
# calls np.dot([1,2,3,4],[1,2,3,4])
30
```

We'll need to update our `eval` function to split our string by spaces and update the rest of our method to reflect this change. Then we'll need to create a `dotproduct` method for our class and call it when needed:

```
import datetime
import numpy as np

class REPL:

    # ...

    def eval(self, user_input):
        help_message = 'Here are the available commands: \n help \n time \n dotproduct'
        result = 'Command not recognized. ' + help_message

        fn_name = user_input.split(' ')[0]

        if fn_name == 'time':
            result = datetime.date.today()
        if fn_name == 'help':
            result = help_message
        if fn_name == 'dotproduct':
            vect1 = user_input.split(' ')[1]
            vect2 = user_input.split(' ')[2]
            result = eval("np.dot(" + vect1 + "," + vect2 + ")")

        return result

#end_src python
```

Run it in the REPL and you should get:

```
#+begin_src bash
Press Ctrl-C Ctrl-C to quit
>> time
2018-09-22
>> help
Here are the available commands:
help
time
dotproduct
```



```
>> dotproduct [1,2,3,4] [1,2,3,4]
30
>>
```

Let's commit this really quick:

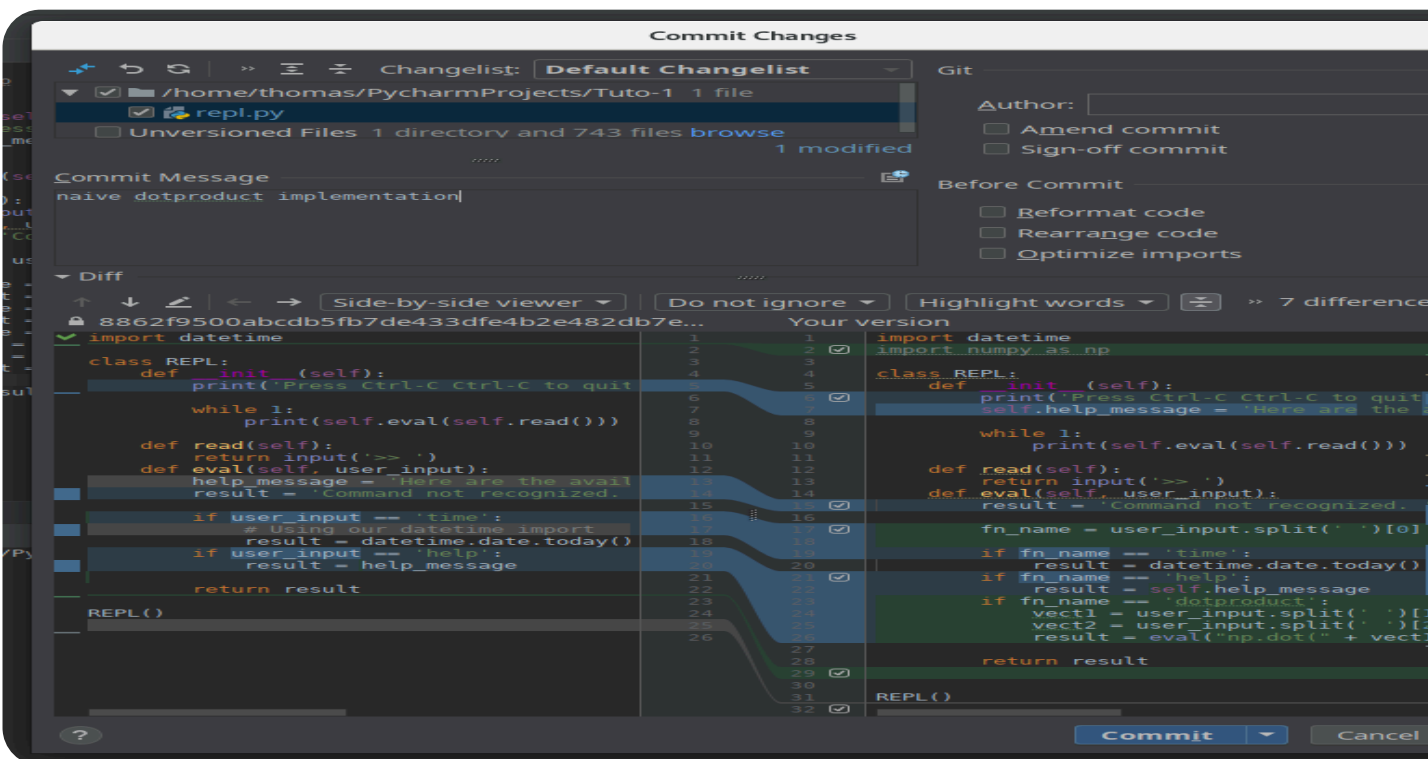


Figure 14: Commit popup

Let's see how this handles typos, try to run `dotproduct [` and you should get something like:

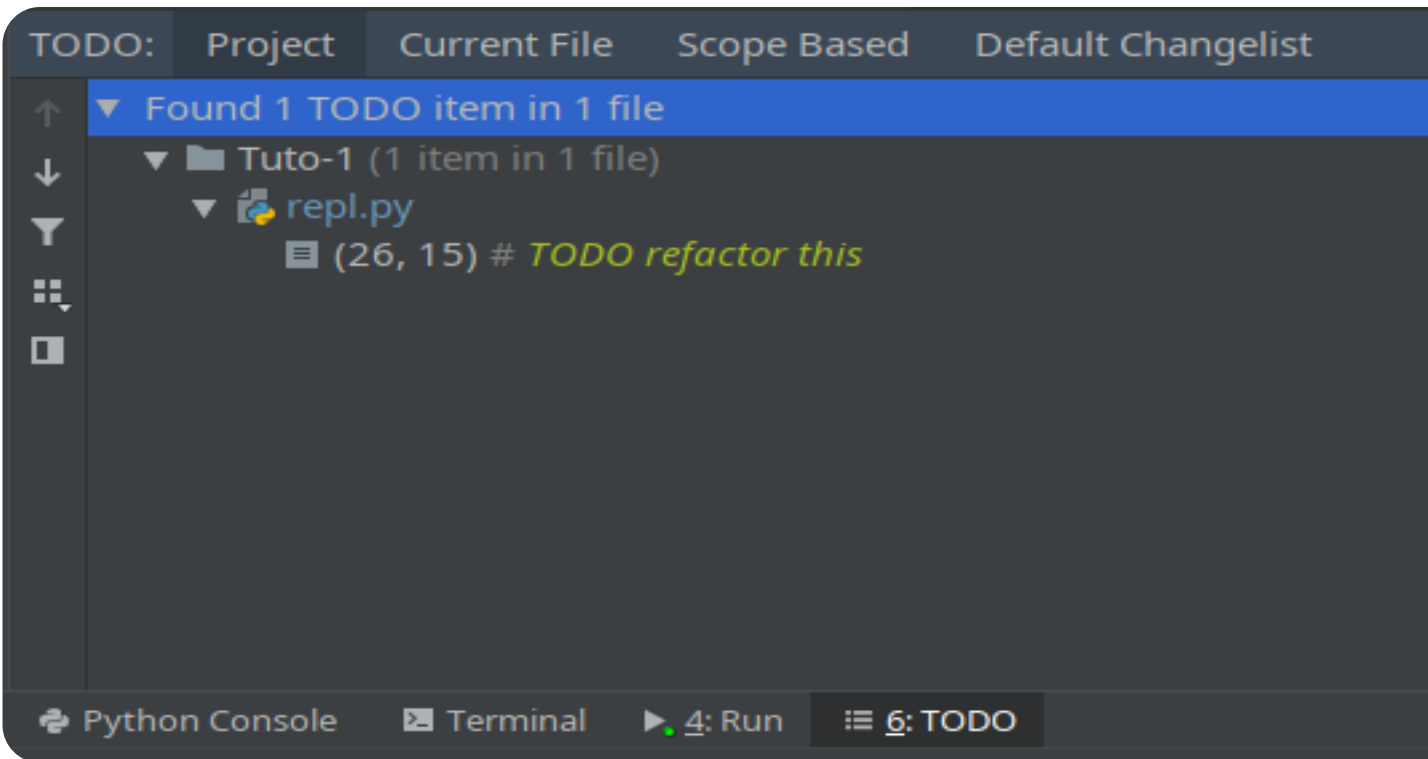
```
Traceback (most recent call last):
  File "/home/thomas/PycharmProjects/Tuto-1/repl.py", line 31, in <module>
    REPL()
  File "/home/thomas/PycharmProjects/Tuto-1/repl.py", line 10, in __init__
```

```
print(self.eval(self.read()))
File "/home/thomas/PycharmProjects/Tuto-1/repl.py", line 25, in eval
    vect2 = user_input.split(' ')[2]
IndexError: list index out of range
```

And our program freezes... This is why in practice **it's considered pure evil to use eval**. We did it here because our goal isn't to make a useful program, but rather to learn about PyCharm by making a program that's kinda fun. That being said, let's make a note in our code that we need to refactor this part of the code in order to make it secure (aka, parse the user entry, construct the array from it and call `numpy.dot` while catching the appropriate exceptions).

```
vect1 = user_input.split(' ')[1]
vect2 = user_input.split(' ')[2]
# TODO refactor this
result = print(eval("np.dot(" + vect1 + "," + vect2 + ")"))
```

Now if you click on the TODO tab at the bottom of your PyCharm window, you'll see your message appear from there. Really useful to track todos !



Looking at our git tree visually

Click here on the clock icon located at the top right of your PyCharm window to pop the Git log tree.

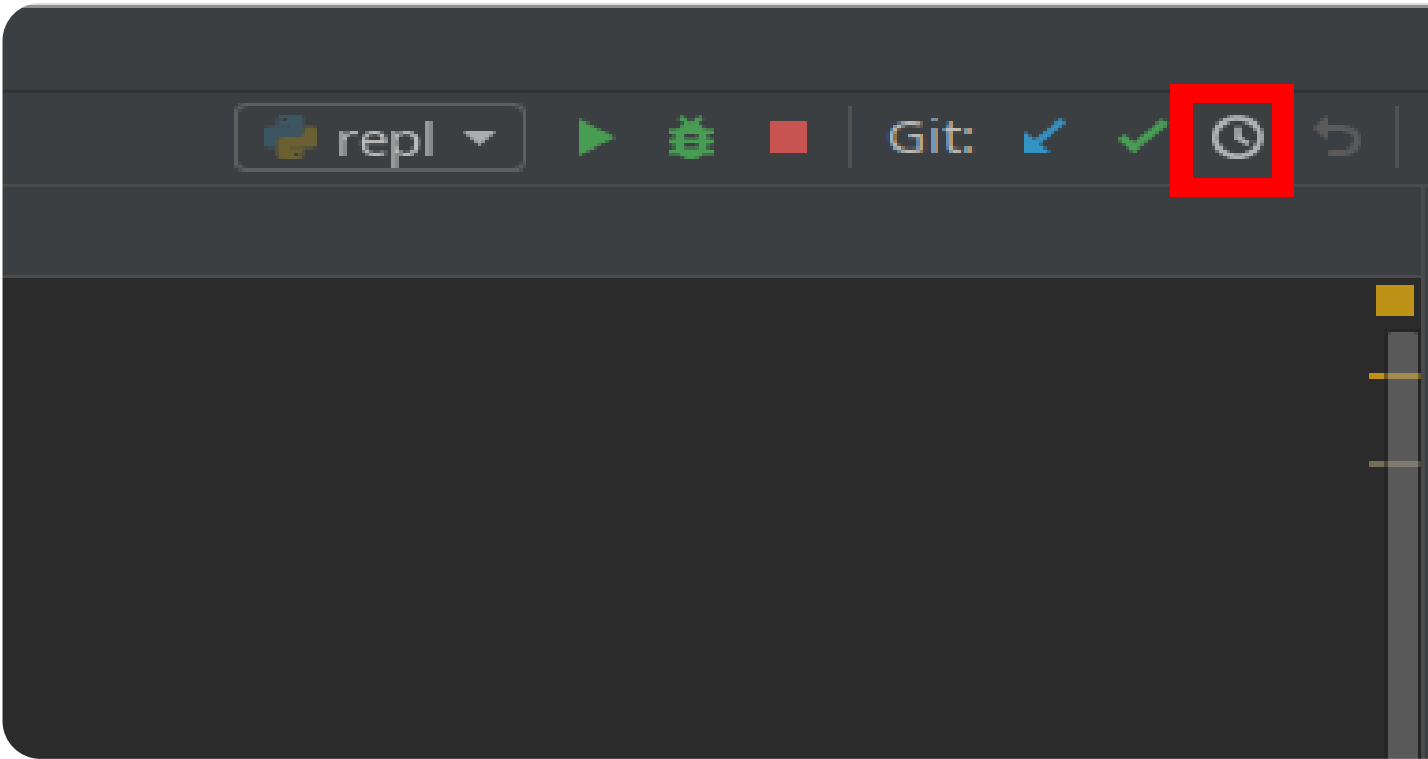


Figure 16: *Git log icon*

This shows you a visual commit tree in a tray that should have appeared. Double clicking on a commit message will show you the commit's diff.

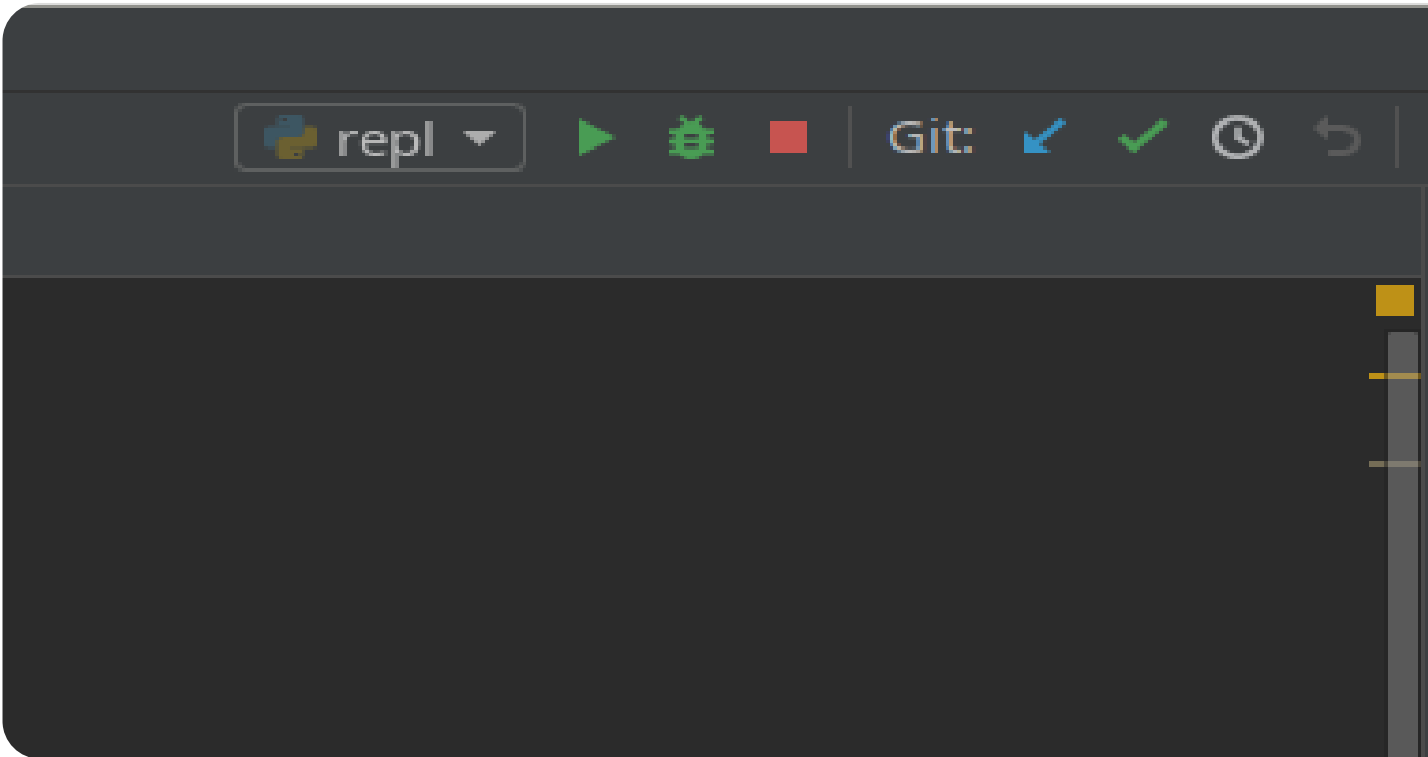


Figure 17: PyCharm visual branch log

You can also right click on the commits to have more options. Of course, there's also a command line way to see your tree: `git log --graph --all`.

So far, you'll notice that our tree is completely linear. Our `master` branch is 2 commits from the root, and our `dev` branch is 4 commits away. However, our branches haven't diverged yet. Merging `master` with `dev` would bring our `master` branch up to date with our `dev` branch. This is called a **fast-forward merge** in Git jargon. They, by definition, *cannot create conflicts* and are therefore very easy to handle.

Switching branch

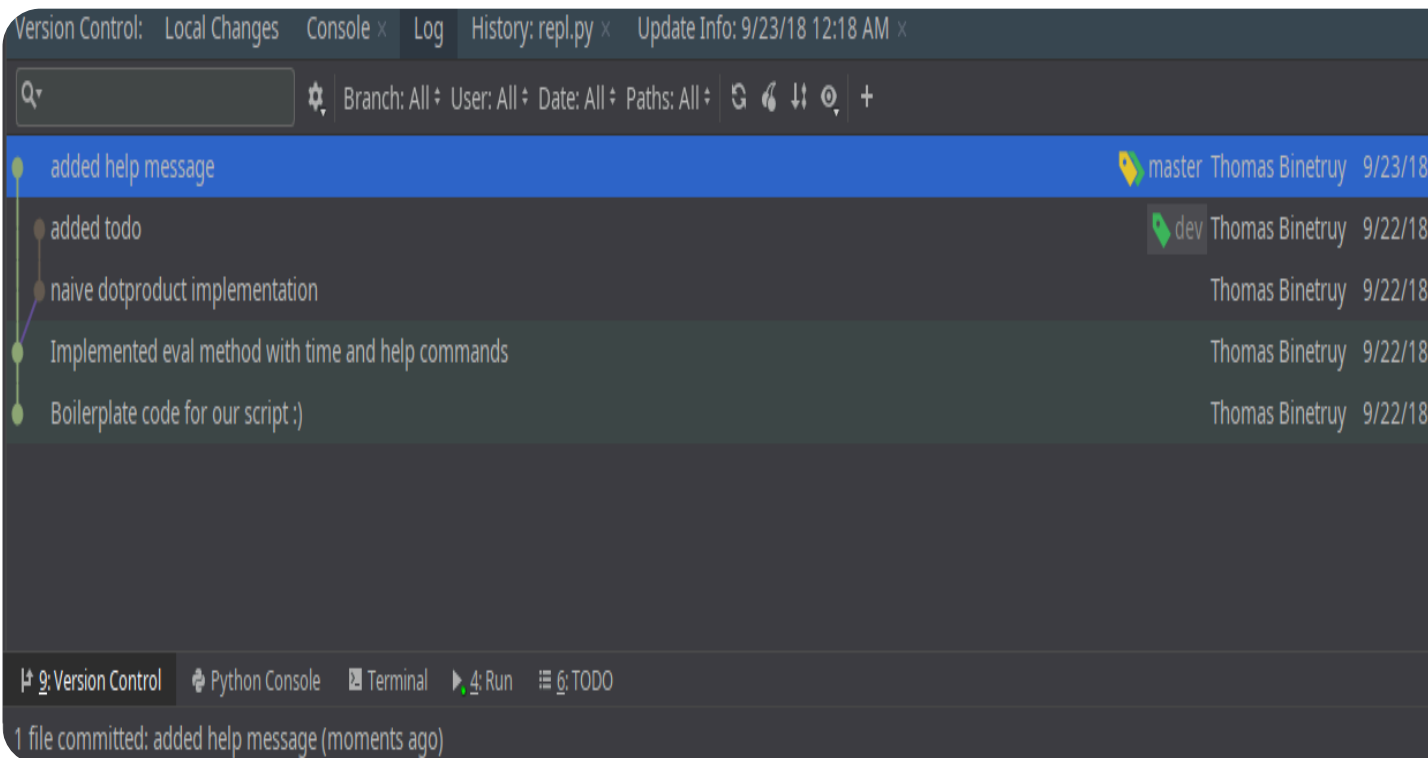
Let's switch back to our `master`. In the top toolbar:
`VCS -> VCS Operations Popup -> Branches... -> Master -> Checkout`

The code should have updated before your eyes :)

Let's introduce a commit in order to diverge from our `dev` branch. Indeed, some of our users don't understand what to do when using our program, so we'd like to introduce them with a help message at the when they launch our program. Let's modify our class' constructor:

```
def __init__(self):  
    print('Press Ctrl-C Ctrl-C to quit for the terminal or Ctrl-D from the REPL.')  
    print('Here are the available commands: \n help \n time \n dotproduct')
```

Let's commit our changes and see what our git tree looks like before merging our two branches.





• Merging with the dev branch

- 1 Right click on your code and `Git -> Repository -> Merge Changes...`
- 2 Check the `dev` checkbox
- 3 Click on `Merge`

If you've encountered a conflict, congratulations - you're now introduced to the Merge Conflict popup! Just click on `Accept Theirs` for now. If you're still having problems, type `git commit -m "merging"` in the terminal to finish the merge.

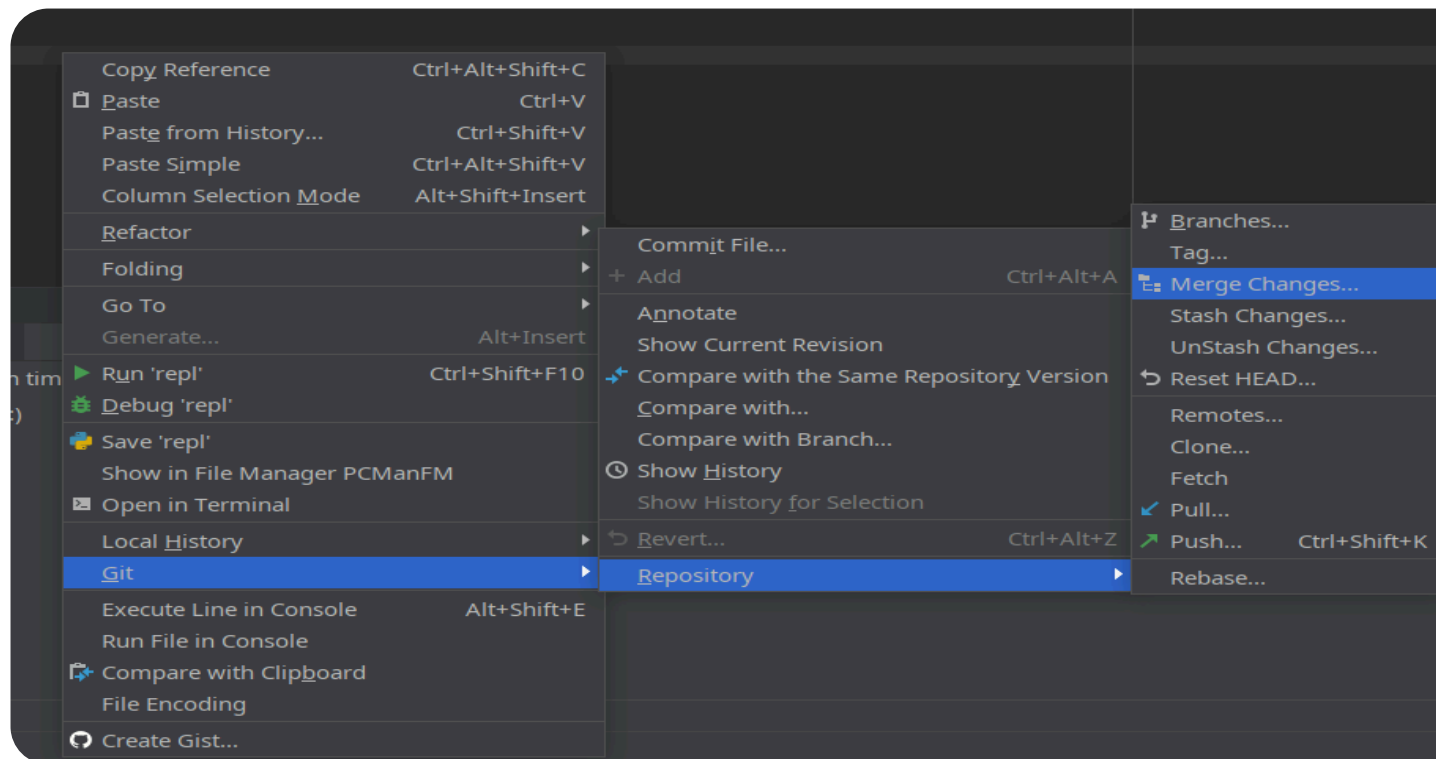


Figure 19: PyCharm merge popup location

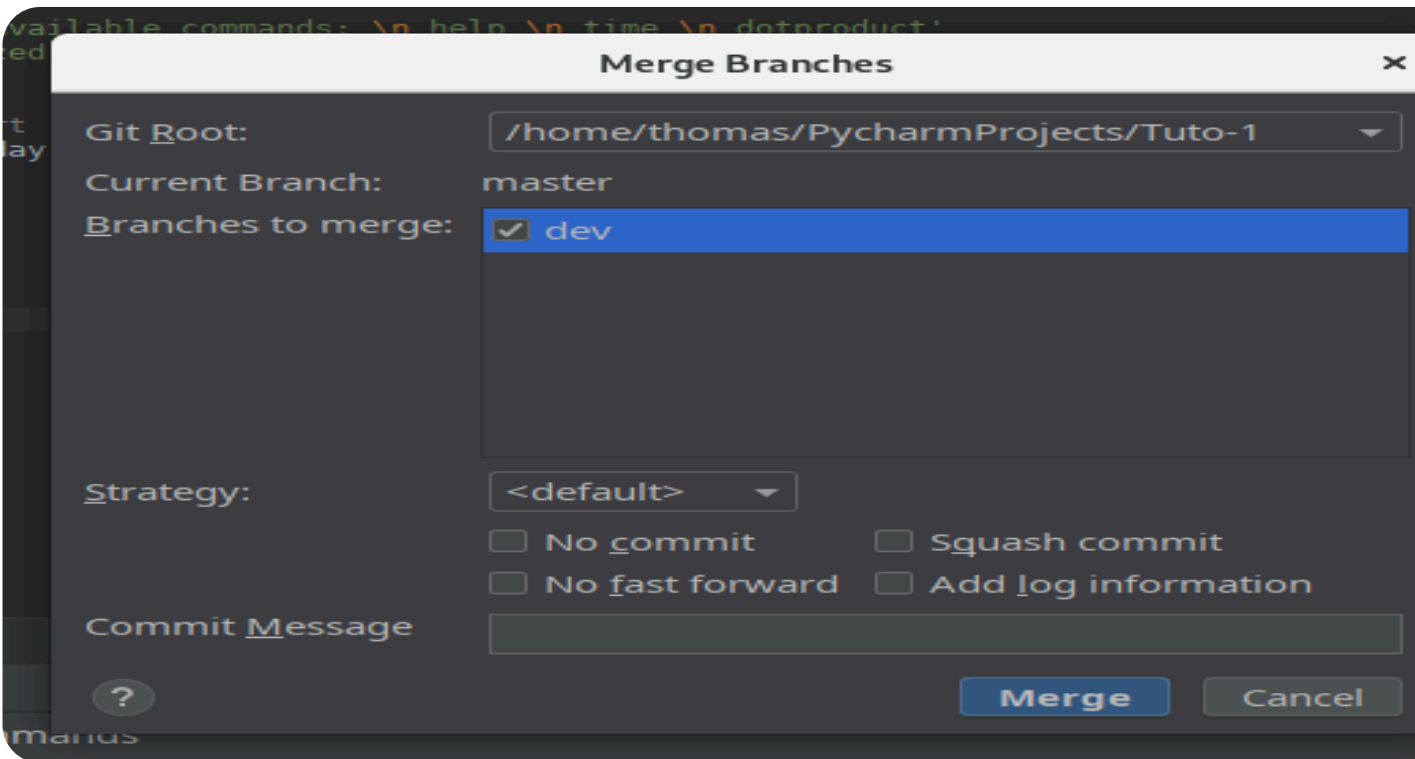


Figure 20: PyCharm merge popup

Our code on our master branch has now integrated those in the dev branch. Check out the tree on PyCharm's interface (top right clock if it's disappeared):

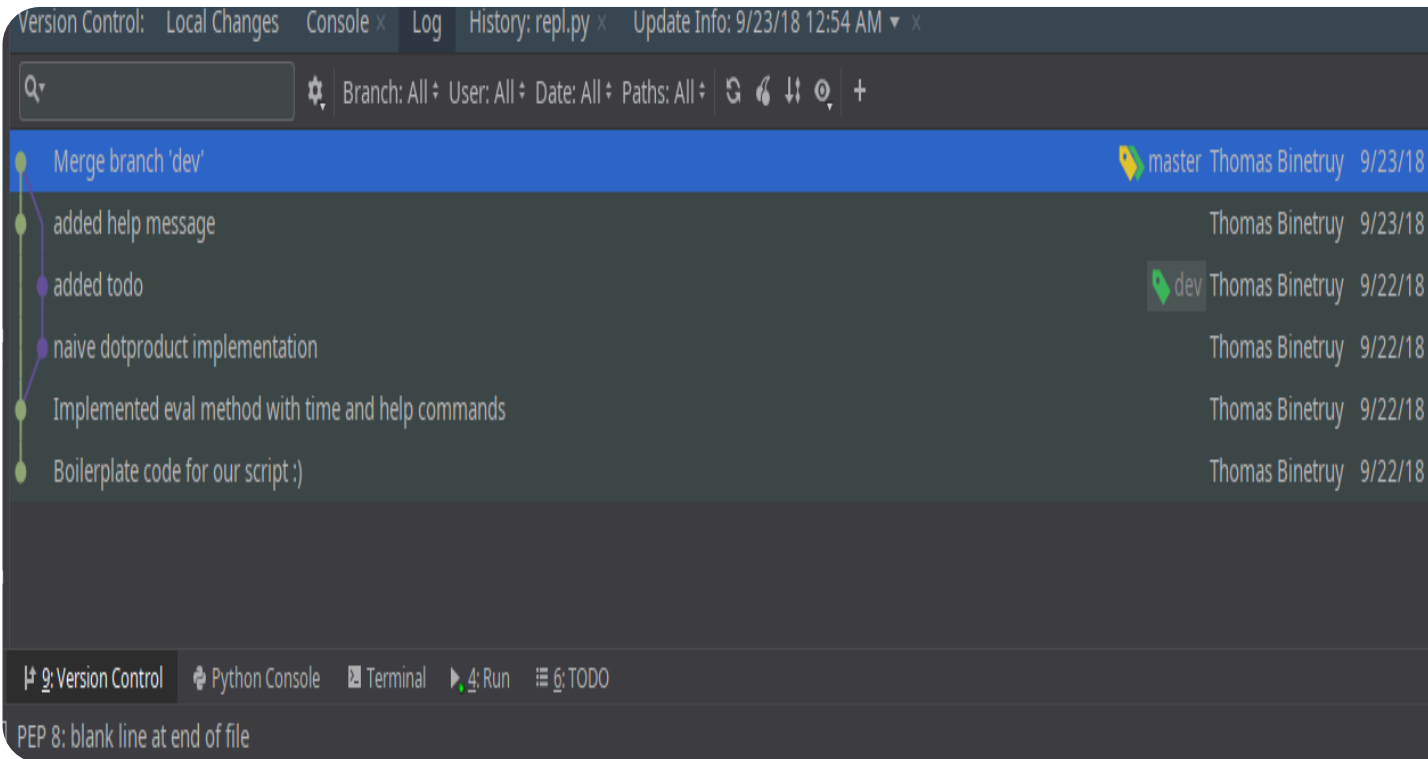


Figure 21: PyCharm Git log after merge. Both the master and dev branches have the same commits

```
* commit 24a33a7c7f15a356daf62a579387f2bae471136d
| Author: Thomas Binetruy <tbinetruy@gmail.com>
| Date:   Sun Sep 23 01:38:41 2018 +0200
|
|     added readme
|
| * commit d9ac2298682db2d6fa0b3e58c5a25f7d232deb48
| \ Merge: a35739e 2889b50
|   Author: Thomas Binetruy <tbinetruy@gmail.com>
|   Date:   Sun Sep 23 01:35:20 2018 +0200
|
|       Merge branch 'dev'
|
| * commit 2889b50cf1429692984dc06343f7bc741bee1716 (dev)
| | Author: Thomas Binetruy <tbinetruy@gmail.com>
| | Date:   Sat Sep 22 23:41:52 2018 +0200
| |
| |     added todo
| |
| | * commit d9840b954ce7e910917b2bba8fe9e9cdf929029c
| | | Author: Thomas Binetruy <tbinetruy@gmail.com>
| | | Date:   Sat Sep 22 23:11:55 2018 +0200
| | |
| | |     naive dotproduct implementation
| | |
| | | * commit a35739ebdd24bfe51a794c6508cfaae1199b8bbb
| | | \ Author: Thomas Binetruy <tbinetruy@gmail.com>
| | |   Date:   Sun Sep 23 01:31:23 2018 +0200
| | |
| | |     added intro message
| | |
| | | * commit 8862f9500abdcdb5fb7de433dfe4b2e482db7e282
| | | | Author: Thomas Binetruy <tbinetruy@gmail.com>
| | | | Date:   Sat Sep 22 19:08:10 2018 +0200
| | | |
| | | |     Implemented eval method with time and help commands
| | | |
| | | | * commit 904ac47801d9a5d950d96e9d8a12e6d6b0f9261a
| | | | | Author: Thomas Binetruy <tbinetruy@gmail.com>
| | | | | Date:   Sat Sep 22 18:52:51 2018 +0200
| | | | |
| | | | |     Boilerplate code for our script :)
| | | | |
| | | | | (END)
|
| 6: TODO      2: Version Control      Terminal      Python Console
```

Figure 22: Git commit tree on the CLI after merge

N.B. If you're wondering why my CLI tree doesn't look exactly the same as my PyCharm on, it's because I had extra commits and rewrote my history so that it wouldn't diverge too much from this tutorial (I tried to keep the same tree shape, but obviously my commit times now differ). Anywho...

Conclusion

Well here you go, this was a quick look into how to use PyCharm to create a fun little Python script, our crazy REPL that give you the current date and can do the vector dot product in the least secure way possible :D

So far we've only used Git to version our own work. This is already really great, but the true power of Git is you can merge your branched with other people's, that you can easily download other git repositories from the web (using `git clone`), contribute to it locally,



and upload your contributions back (with `git push`) for them to be investigating.

However, I believe this document does a *decent* job at giving an overview of the very basics of Git and how they can be benefit *short term*. Similarly, PyCharm has its very own Python REPL that we've started using but that is much more powerful. We it allows use to "send" our file to it and be able to investigate what variables are equal to. Better, it allows us to evaluate specific **parts** of a script on demand. But don't worry about this for now, just know that it exists because maybe you'll need it someday!

Don't forget to have a look at the tree first chapters from [Pro Git](#) which will teach your about branching and what **Git** as actually doing under to hood. You'll learn about how git moves you from a branch to another when you perform a `checkout`, or what a branch actually is, and many other *crucial* concepts in Git. Don't be scared, it's a very *visual* book, and is therefore very helpful in allowing one to construct a mental image of what Git operations *concretely* do.

Let me know if you have any questions about this document: thomas.binetruy@telecom-paristech.fr, and thanks for having spent the time to read this.

The source code for this document is available here: <https://github.com/tbinetruy/pycharm-tuto>. Feel free to create an issue if something doesn't work, or a pull request for those that know what it is and want to contribute.