Git Workshop Testing

Release 0.0.1

Christoph Lange

Feb 18, 2021

BASIC GIT CONCEPTS

Here are the basic git concepts that we covered in the last workshop

CHAPTER

ONE

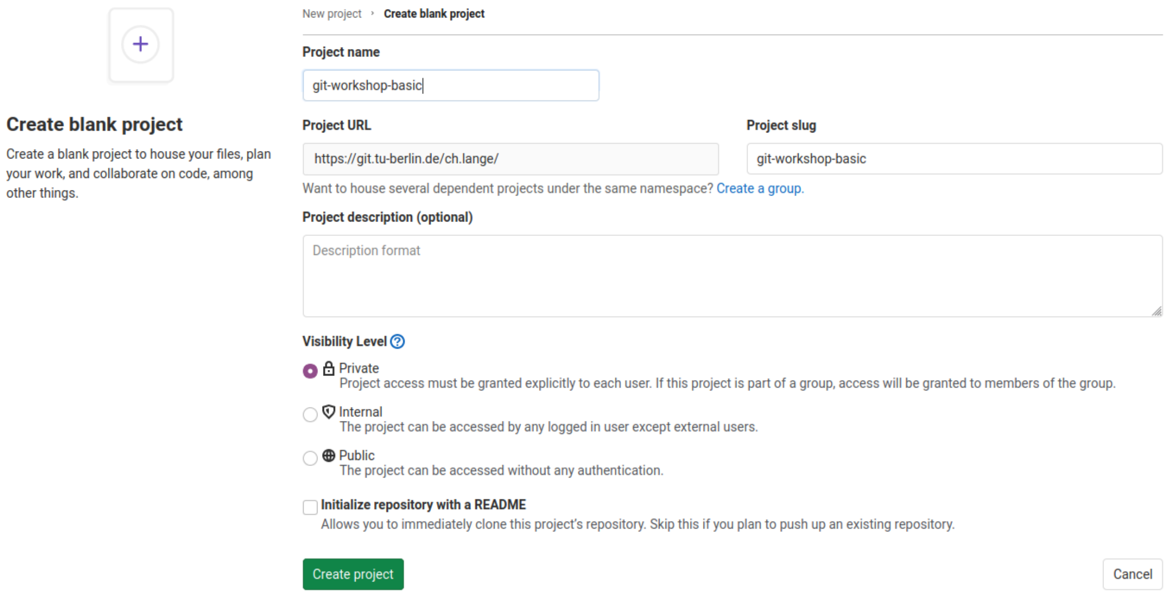
CREATING A REPOSITORY

Steps

The basic idea is to create a repo on the remote server. Then we create some content for the repository locally and finally we want to sync this content to the remote server.

1.1 Create Project on GitLab

First of all you want to create a repository on GitLab/GitHub. Therefore, go to the URL of your GitLab Server, i.e. https://git.tu-berlin.de/kiwi-git-workshops. Then you click on **New Project** and select **Create blank project**. Afterwards you may choose a name for your repository



Create a blank project to house your files, plan your work, and collaborate on code, among other things.

Project name: git-workshop-basic

Project URL: https://git.tu-berlin.de/ch.lange/

Project slug: git-workshop-basic

Want to house several dependent projects under the same namespace? Create a group.

Project description (optional): Description format

Visibility Level

[✓ Private] Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

Internal The project can be accessed by any logged in user except external users.

Public The project can be accessed without any authentication.

☐ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

and click **Create project**. Now we created an empty project on the remote server.

1.2 Use Project Template

Now we create a folder with some code on our local machine. Therefore we use a template via the following steps:

1. Open a terminal
2. Install the python package cookiecutter

   ```
   pip3 install cookiecutter
   ```

3. Use *cd* to navigate to the directory that you want to start a repository.

   ```
   cd path/to/your/git-projects
   ```

4. Create your python package with

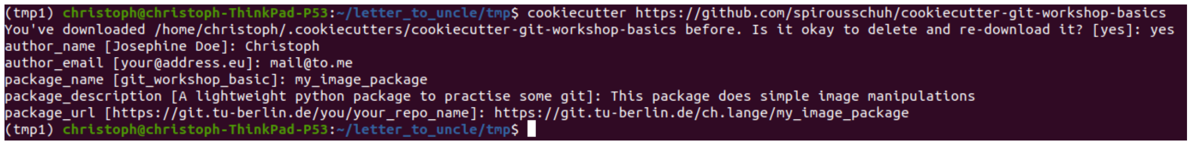
   ```
   cookiecutter https://github.com/spirousschuh/cookiecutter-git-workshop-testing
   ```

5. Specify the template parameter. Now you will see

   ```
   author_name [Josephine Doe]:
   ```

   This is a question. "What should be the name of the author?" and requires your input. You can either press *Enter*, then the author_name is set to the default option Josephine Doe. Or you can enter another name.

6. Answer the questions that will be prompted to you or press *Enter* to choose the default value. You do not need to reveal your real data, as it is a toy project anyway. But you could choose answers like these:

   ```
   (tmp1) christoph@christoph-ThinkPad-P53:~/letter_to_uncle/tmp$ cookiecutter https://github.com/spirousschuh/cookiecutter-git-workshop-basics
   You've downloaded /home/christoph/.cookiecutters/cookiecutter-git-workshop-basics before. Is it okay to delete and re-download it? [yes]: yes
   author_name [Josephine Doe]: Christoph
   author_email [your@address.eu]: mail@to.me
   package_name [git_workshop_basic]: my_image_package
   package_description [A lightweight python package to practise some git]: This package does simple image manipulations
   package_url [https://git.tu-berlin.de/you/your_repo_name]: https://git.tu-berlin.de/ch.lange/my_image_package
   (tmp1) christoph@christoph-ThinkPad-P53:~/letter_to_uncle/tmp$
   ```

   Pay attention at the third question. The answer to that question will be the name of the folder where you can find your package later.

   Now we created a folder of code locally.

1.3 Sync Local and Remote Repository

In this section we will syncronize our local folder with the remote git server. Right know they do not know about each other.

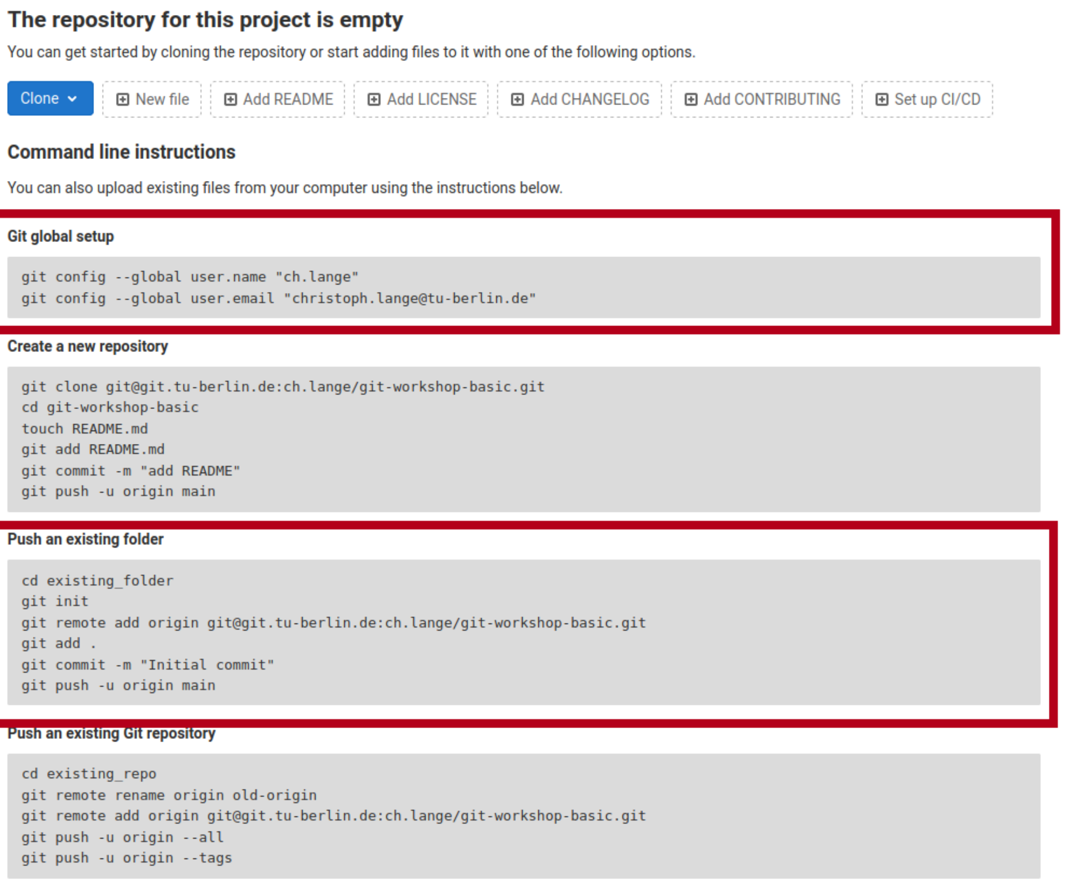
1. Go the folder that you just created in the last step

   ```
   cd my_image_package
   ```

   The name of the folder corresponds to your answer to the question

   ```
   package_name [git_workshop_testing]: my_image_package
   ```

2. Go back to your browser and open the remote server url (https://git.tu-berlin.de). Then go to the project that you just created in the section *Create Project on GitLab*. As it is an empty project the landing page should look like this:



The repository for this project is empty

You can get started by cloning the repository or start adding files to it with one of the following options.

Command line instructions

You can also upload existing files from your computer using the instructions below.

Git global setup

```
git config --global user.name "ch.lange"
git config --global user.email "christoph.lange@tu-berlin.de"
```

Create a new repository

```
git clone git@git.tu-berlin.de:ch.lange/git-workshop-basic.git
cd git-workshop-basic
touch README.md
git add README.md
git commit -m "add README"
git push -u origin main
```

Push an existing folder

```
cd existing_folder
git init
git remote add origin git@git.tu-berlin.de:ch.lange/git-workshop-basic.git
git add .
git commit -m "Initial commit"
git push -u origin main
```

Push an existing Git repository

```
cd existing_repo
git remote rename origin old-origin
git remote add origin git@git.tu-berlin.de:ch.lange/git-workshop-basic.git
git push -u origin --all
git push -u origin --tags
```

3. Follow the step that are displayed under **Git global setup** (first red box) one by one, i.e. you copy each line to your terminal and press *Enter*.

4. Follow the steps you find in the section **Push an existing folder** (second red box). You need to replace *cd existing_folder* with the *project-name* you chose in step 6. In case you forgot the package name you can check it with *ls -l* which displays the content of the current directory. (if you get an error like *error: src refspec main does not match any* you need to replace main with master)

5. Install your new package in editable mode

```
pip install -e .
```

6. Go to your project webpage *https://git.tu-berlin.de/your_name/your_project*. When you see a basic README.md file you succeeded.

CHAPTER

TWO

GIT WORKFLOW

2.1 Idea

This is a concise manual to a basic Git workflow. You can find more details here. For each step you can find instructions how to follow that workflow using PyCharm. There is different ways to achieve the same goal without PyCharm. Once you are familiar with the basic concepts you can use any tool you like.

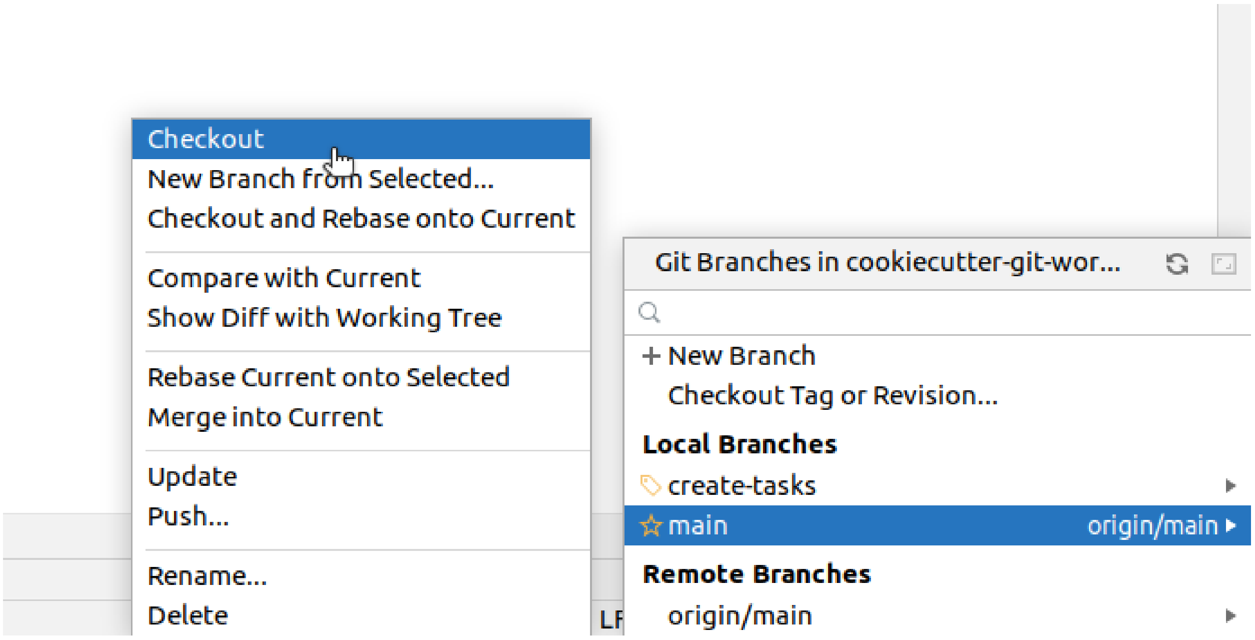
2.2 Instructions

Once you have an idea what you want to achieve the following steps will help you to get there.

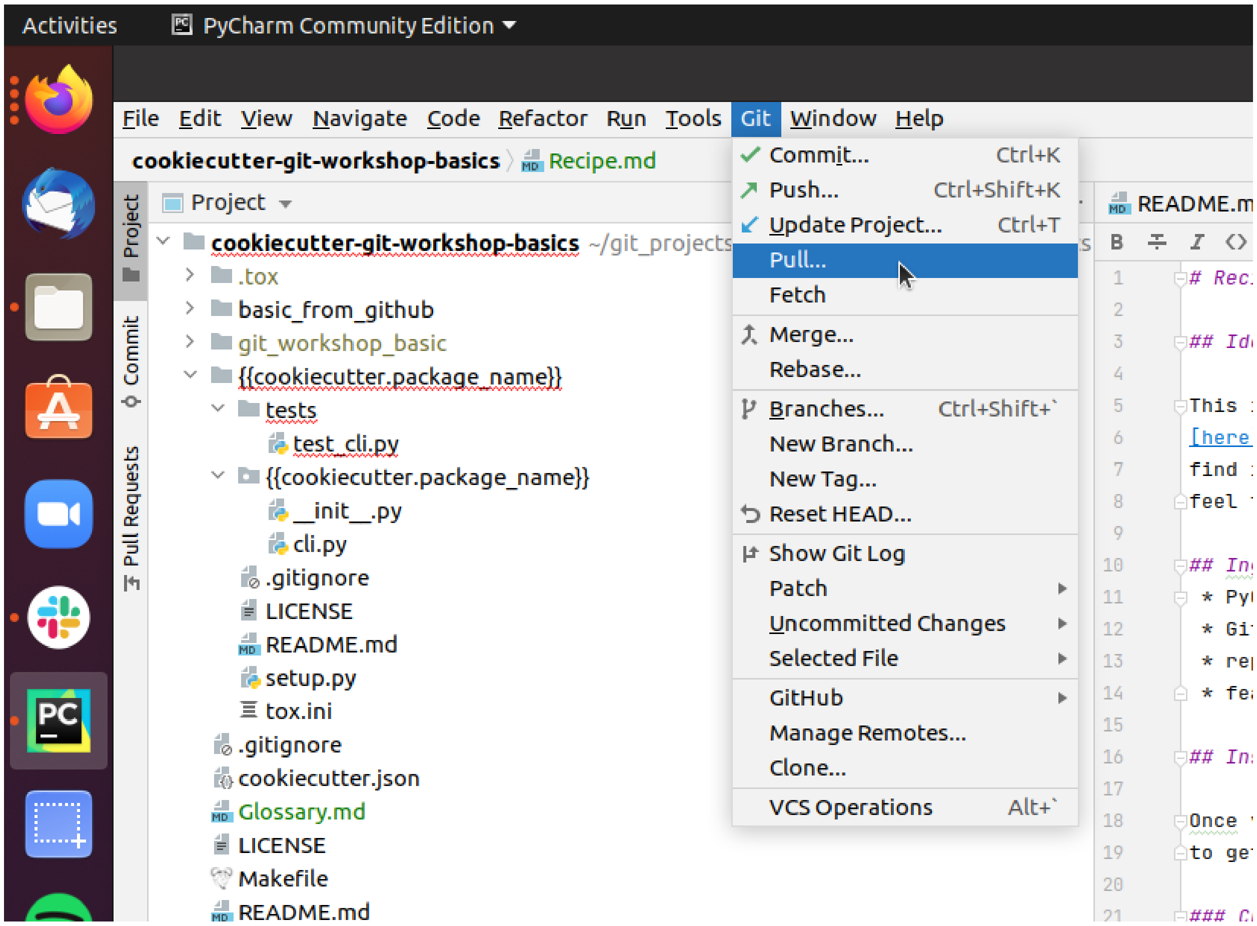
Steps

2.2.1 Update Local

First we want to make sure to use the newest version of the repositories main branch. Therefore we click on the button in the bottom right corner next to the patlock. Then we see a context menue like this that displays all the local branches.

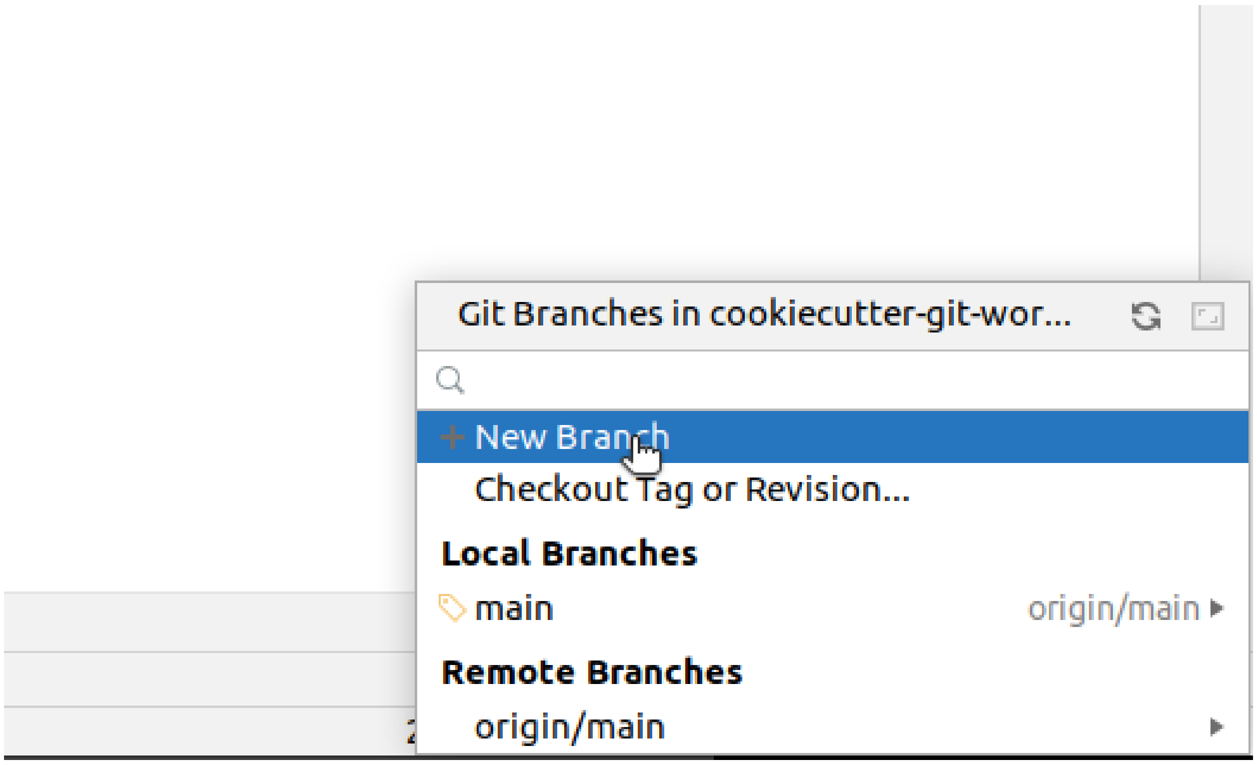

Checkout
New Branch from Selected...
Checkout and Rebase onto Current
Compare with Current
Show Diff with Working Tree
Rebase Current onto Selected
Merge into Current
Update
Push...
Rename...
Delete
Git Branches in cookiecutter-git-wor...
New Branch
Checkout Tag or Revision...
Local Branches
create-tasks
main
origin/main
Remote Branches
origin/main

Click on the main/master branch and choose "Checkout" in the second context menue to switch to the main/master branch. Now we need to make sure that your local main/master branch is up to date with the upstream main/master. Therefore we pull the newest state from upstream. In the upper left corner we can find the menue bar, click on "Git" and choose pull in the pull down menue.



2.2.2 Create Branch

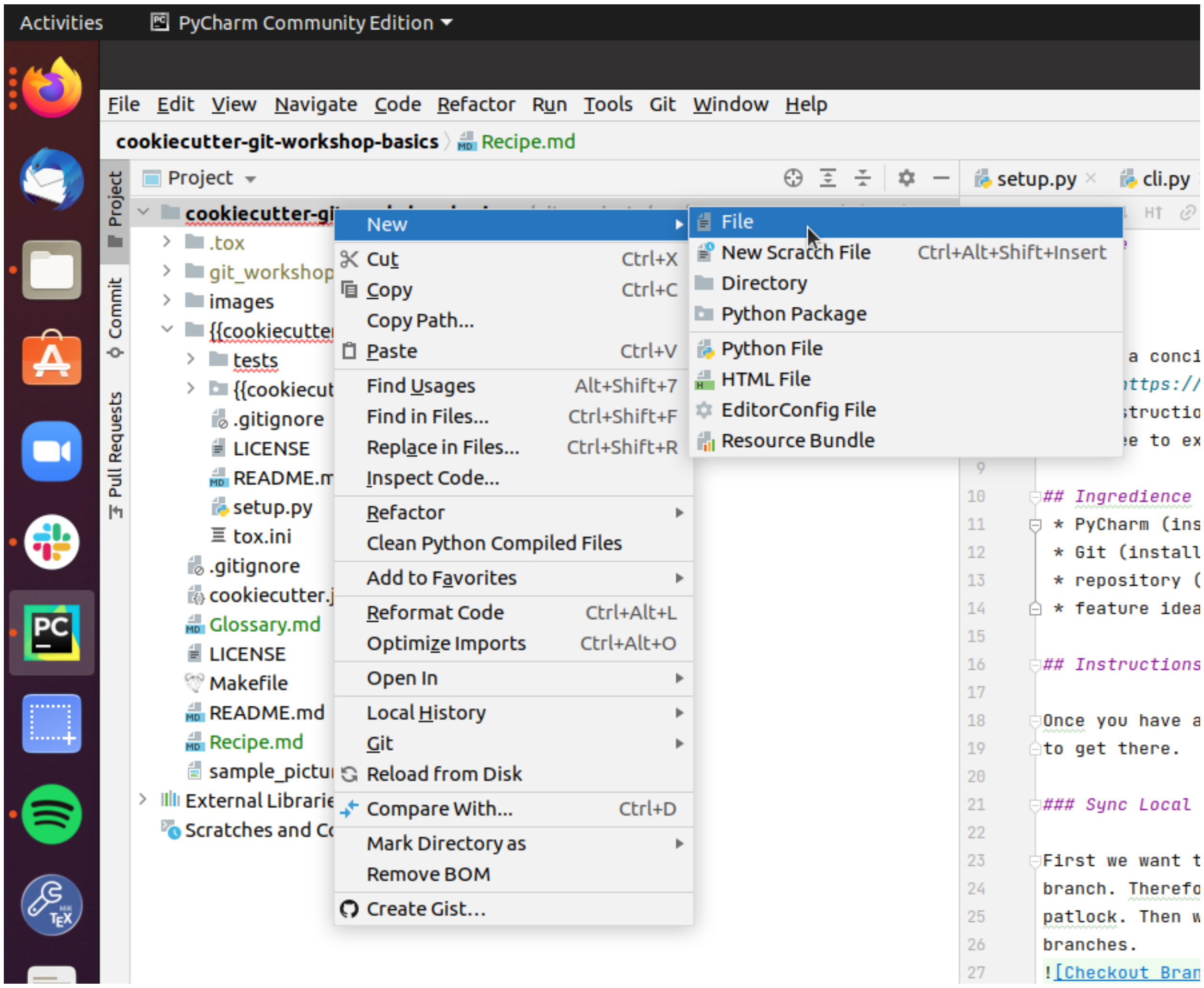
Now we create a branch to implement our feature. In order to do so move your cursor to the buttom right corner and click on your current branch name, which should be main/master, next to the patlock.



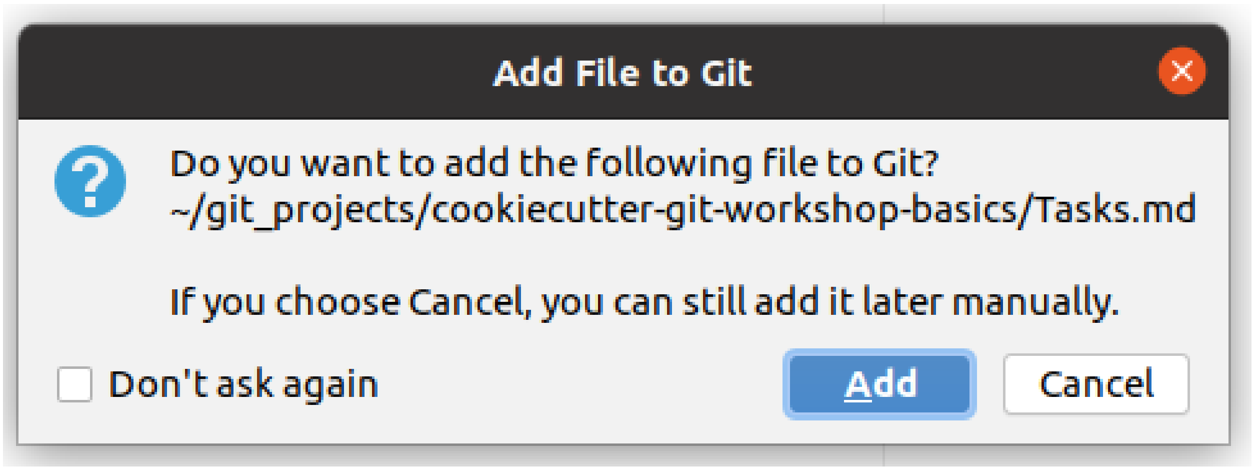
Within the context menue click on "New Branch" and enter a branch name that relates to your feature idea.

2.2.3 Add Commits

Now you need to add, change or delete some content in the repository to achieve your goal. For instance you want to add a new file "Tasks.md". Then you make a right click onto the folder that should contain your new file.



In the context menue select "new" and "File" and enter the filename in the consecutive prompt. Then PyCharm wants to know if Git should look after your new file.


Add File to Git

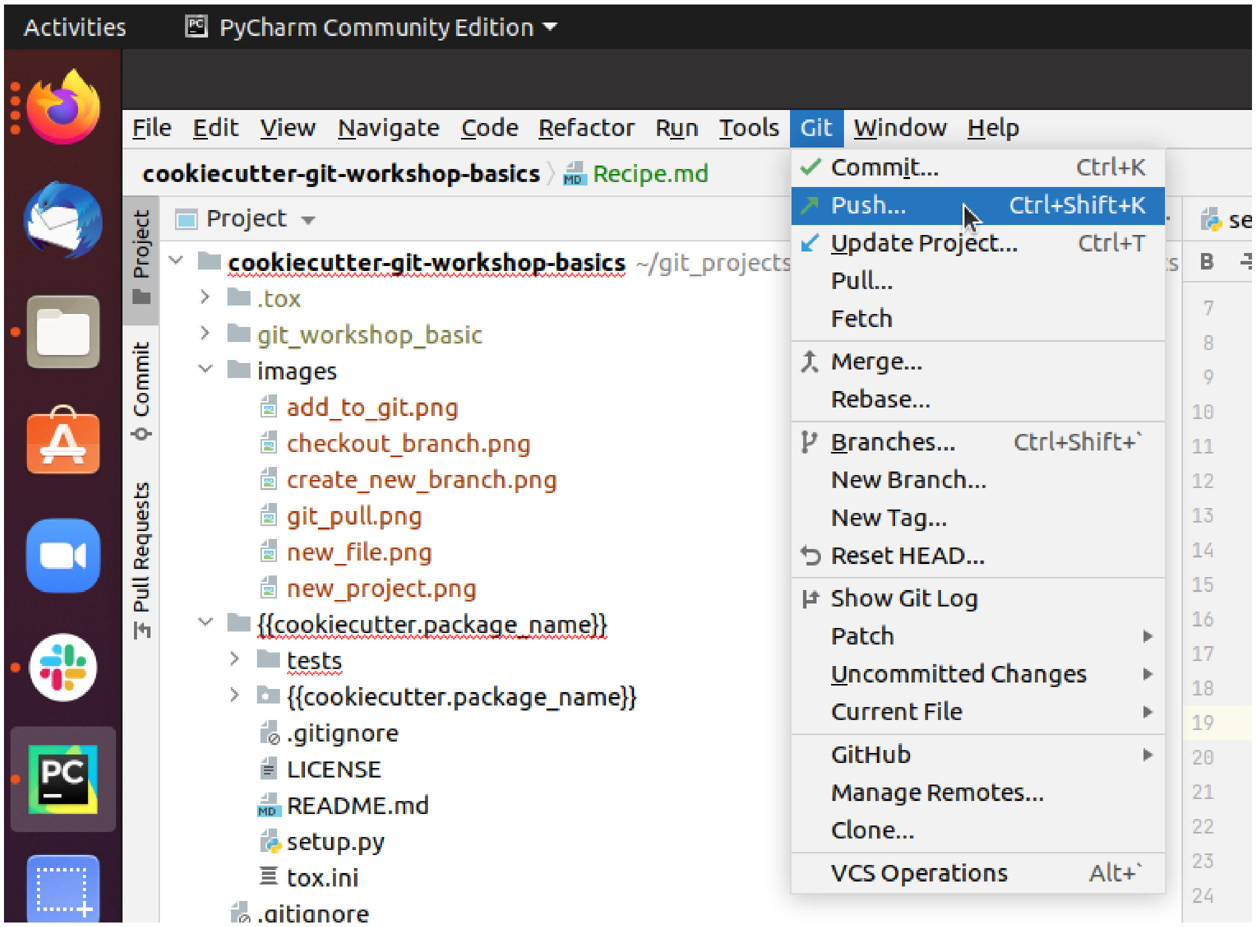
Do you want to add the following file to Git?
~/git_projects/cookiecutter-git-workshop-basics/Tasks.md

If you choose Cancel, you can still add it later manually.

Normally that is a good idea and you shall choose "Add".

2.2.4 Push Branch

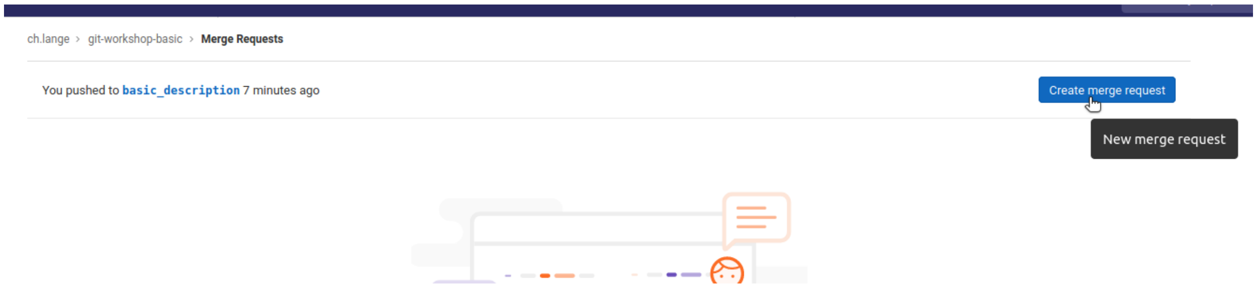
Now you want to push the branch with your changes to the upstream server. This way you create an identical copy of your local branch on the server. To do so



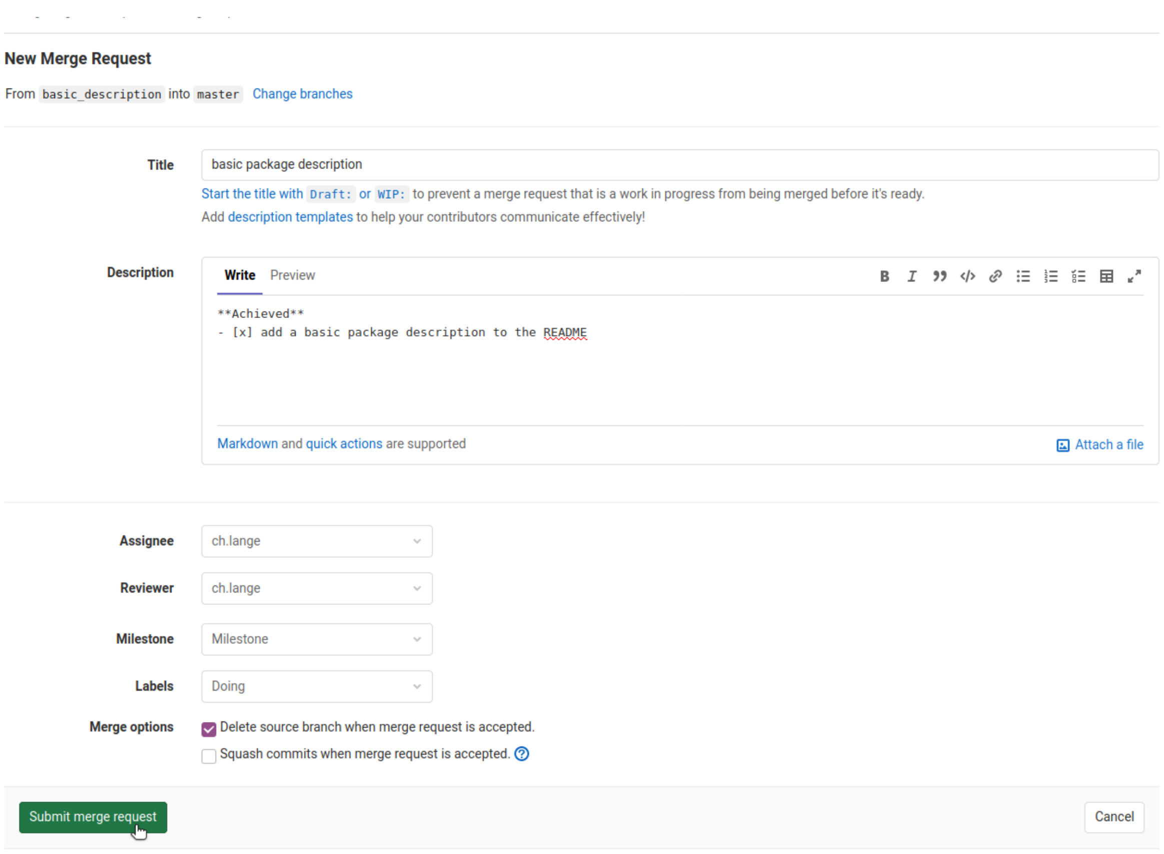
go to the upper left corner where you can find the menue bar and click on *Git* and choose *push* in the pull down menue.

2.2.5 Merge Request

Now that you pushed your local branch to the upstream server, you want to create a merge request on the server. Therefore open your browser and go to *https://git.tu-berlin.de/your_name/your_project/*. On the left hand side you click on *Merge Requests*. Then you get to a page that looks like this:


ch.lange › git-workshop-basic › Merge Requests
You pushed to basic_description 7 minutes ago
Create merge request
New merge request

Here click on *Create merge request* to create a request to merge *your_branch* (here: "basic_description") into main/master. Then you can add a description


New Merge Request

From `basic_description` into `master` Change branches

Title: basic package description

Start the title with `Draft:` or `WIP:` to prevent a merge request that is a work in progress from being merged before it's ready.

Add description templates to help your contributors communicate effectively!

Description:

```
**Achieved**
- [x] add a basic package description to the README
```

Markdown and quick actions are supported

Assignee	ch.lange
Reviewer	ch.lange
Milestone	Milestone
Labels	Doing

Merge options: [✓] Delete source branch when merge request is accepted. [ ] Squash commits when merge request is accepted.

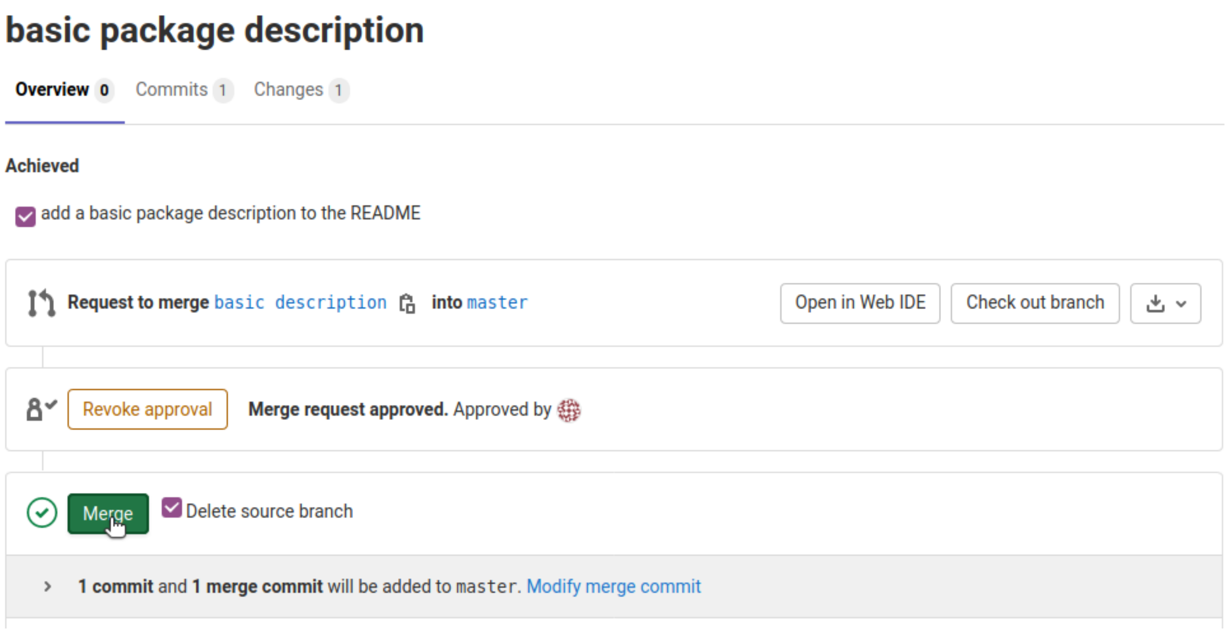
and assign a reviewer. Finally submit you merge request.

2.2.6 Discussion

Now the reviewer of the merge request checks your changes and gives you feedback. After some discussion you might want to go back to step 3 and add additional commits to change the current state. For the sake of practising some interations in the workshop, you can just approve your own merge requests and continue.

2.2.7 Merge Branch

When all discussions are done and you are sure that your changes improve the main/master branch, it is time to merge your branch by


basic package description
Overview 0
Commits 1
Changes 1
Achieved
add a basic package description to the README
Request to merge basic description into master
Open in Web IDE
Check out branch
Revoke approval
Merge request approved. Approved by
Merge
Delete source branch
1 commit and 1 merge commit will be added to master. Modify merge commit

clicking on **Merge**.

Now master on the upstream server is newer than your local branch and its time to start all over again (*Update Local*).

Today we will learn how to write unit tests.

CHAPTER

THREE

UNIT TESTS

3.1 Idea

We want to test a unit of code to make sure it does what we expect, i.e.

```python
def test_addition():
    # given
    summands = [3, 2]

    # when
    the_sum = sum(summands)

    # then
    assert the_sum == 5
```

Main ingredience of a unit test

- test data
- the functionality you want to test
- meaningful assert statements

3.2 Code of Conduct

Some things to keep in mind when writing your unit tests.

Does	Do nots
bring your own test data	API calls
temporary files	read test data from files
negative control	mocks
self-sufficiency	
focus on your own package	

CHAPTER

FOUR

THE CURSE OF UNIT TESTS

4.1 Reasons against Testing

There is a couple of “reasons” out there why people do not want to test their code

- my project is late
- do not touch my code it works
- my code is un-testable
- I just work on the project myself

No doubt testing your code is hard now, but working on non-tested code is much harder in the long term. Its kind of an investment that will pay off in the future.

4.2 Benefits of Testing

How tested code makes your life easier:

- more certainty on how it works
- increases trustworthiness
- reduces technical depth
- documents the usage of your code
- puts yourself into your users’ shoes
- fosters good coding practices
- focus on your current task and forget about the rest of the code
- Be lazy! Let the computer do the checking.

Moreover we will deal with the most important concepts to use unit test for the sake of improve your code quality.

CHAPTER

FIVE

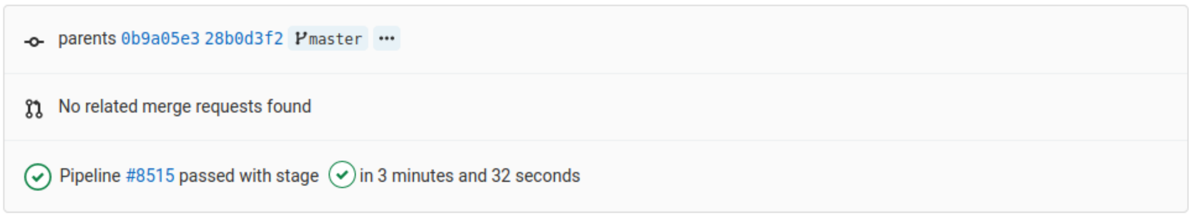
CONTINUOUS INTEGRATION

5.1 Idea

Continuous Integration is the traffic lights of software development. It checks the traffic for you, to see if you are good to go. Therefore it automatically checks things like

- running all unit tests
- tries different system configurations
- checks coding conventions

You can extent the checks to whatever you like. When all checks ran successfully you get a green light.



parents 0b9a05e3 28b0d3f2 master ···

No related merge requests found

Pipeline #8515 passed with stage in 3 minutes and 32 seconds

5.2 Rules

For continuous integration to work properly, it is crucial to enforce the following rules

- the main / master branch is protected
- the only way to alter the main / master branch is a Merge Request
- Just merge when lights are green

Seting up CI for your repository is not straight forward. Therefore for today's workshop it is enabled automatically, as long as you created your repo in the right group (https://git.tu-berlin.de/kiwi-git-workshops).

For more information on how to setup CI, please take a look here https://docs.gitlab.com/ee/ci/introduction/.

CHAPTER

SIX

REFACTORING

6.1 Idea

What is refactoring?

- rearranging your code
- does not change the functionality

Why shall we do it then?

- structure
- improves readability
- makes it easier to maintain
- might ease testing

So how does testing come into the picture? We want to be sure that the functionality does not change.

CHAPTER

SEVEN

TEST DRIVEN DEVELOPMENT

7.1 Rules

The idea behind test driven development is to start writing a test, before you do any coding. If you want to be strict about it the following rules lead you to glory and honor.

- just write code to pass a failing unit test
- once the unit test passes you are not allowed to continue on the coding side

7.2 Benefits

- start coding from general apects to specifics
  - What do I need?
  - How does the interface look like?
  - What are the parameters?
- focus on a small pieces (an iterative approach)
- issues do not accumulate
- ensures 100 % code coverage

7.3 Disadvantages

- works well on greenfield projects
- counter intuitive

Therefore we have a collection of tasks to practise the methodologies introduced above

CHAPTER

EIGHT

TASK 0: CREATE A NEW REPOSITORY

Similar to last time we want to create a new repository that we use for this workshop. Please note that we want to create a repository in the group kiwi-git-workshops.

For step by step instructions on how to create a repository, you can take a look at the creating a *Creating a Repository* page.

CHAPTER NINE

TASK 1: UNIT TESTS

Please create a seperate branch for each of the sub-tasks and create a Merge Request every time. You can find detailed instructions on the *Git Workflow* page.

9.1 Understand the Function

The repository that you created in task 0 mostly consists of two python files *cli.py* and *processing.py*. The file *processing.py* contains a function called *invert_image* which is not properly tested.

Try to understand what it does.

If you want more clarity on how it works, it might be a good idea to take a look on a unit test. Try to find the unit test that corresponds the *invert_image* function.

9.2 Extend a Unit Test

Now it is your job to change that. In the file *tests/test_processing.py* there is a function called *test_invert_image_one_pixel*. This is a first basic unit test you can start with.

Do not forget to create a Merge Request and merge it.

9.3 Add a Second Unit Test

Now that we have one unit test for the function *invert_image* we want to check if it really works properly. Therefore we want to write a another unit test.

In the file *tests/test_processing.py* there is a function called *test_invert_image_two_pixel*. Please use this function to write a second unit test.

CHAPTER TEN

TASK 2: REFACTORING

Please create a seperate branch for each of the sub-tasks and create a Merge Request every time. You can find detailed instructions on the *Git Workflow* page.

10.1 Rewrite function

Now that we tested our function *invert_image* properly its time to refactor it.

Why do we want to do that? Well the package Pillow already has this functionality build-in. So there is no need to implement it ourself.

Now it is your turn. Please try to find the *invert* functionality of Pillow and change the function *invert_image* such that it uses the Pillow version of inverting an image.

CHAPTER

ELEVEN

TASK 3: TDD

Now we would like to implement something from scratch. Therefore it is the perfect opportunity to try some test driven development.

11.1 Crop an Image

Your task is to implement a functionality that crops an image.

11.2 Hints

As we are doing TDD you need to start with the unit test. To keep things simple in the first place create a unit test in *test_processing.py*.

Try to think of a very simple image and a very simple crop. You might want to copy some parts of the other unit test.

Once you have a first unit test for the pure cropping function, you can start with a unit test for the command line interface. The command line interface can be found in *cli.py*. The unit test for it can be found in *test_cli.py*.

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- search