Developers guide Documentation

Release 0.1

DFTB+ developers

Feb 15, 2024

CONTENTS

CHAPTER

ONE

GIT WORKFLOW FOR CONTRIBUTORS

1.1 General workflow

For developing DFTB+ we use a feature-branching workflow described, for example, in Understanding the GitHub Flow. The main points for most developers are:

- Development happens based on the *main* branch, which always contains a clean release-ready code.
- Every feature is developed in a separate feature branch, which is derived from the *main* branch. If the feature is mature enough (it works correctly, its code is clean, it is well documented, thoroughly tested, etc.), the feature branch is merged into the *main* branch.
- In order to ease integration, feature branches should be **short living** and pull requests should only contain a **reasonably small amount** of changes. Try to chop your implementation into small self containing changes and issue those in separate consecutive pull requests. (No one likes or is efficiently able to review a code change containing thousands of lines.)

In order to add a feature, you have to do the following steps:

1. Fork the official (upstream) repository on GitHub and make a local clone of your fork (origin).
2. Synchronize your *main* branch (`origin/main`) to match upstream *main* (`upstream main`).
3. Derive a feature branch from the *main* branch of your forked project.
4. Develop and finish your feature in your feature branch.
5. Integrate eventual changes from the upstream main branch.
6. Issue a *pull request* for your feature branch.
7. Wait for feedback from the maintainers and then apply any suggestions or required changes to your feature branch.
8. When you obtain the notification that your feature branch has been merged to the upstream *main* branch, delete the feature branch in your personal repository.
9. In order to develop the next feature, execute the above steps again, *starting from step 2*.

Below you find a detailed description of each step, using the DFTB+ main repository as an example. If you work on an other DFTB+ related project, replace the repository name *dftbplus* with the actual repository name.

1.2 Fork the project

1.2.1 Fork the repository

1. Fork the desired repository (e.g. *dftbplus*) owned by the user *dftbplus* to *your personal* GitHub account. You will find the *Fork* button in the upper right corner on the project page.
2. Check out your personal fork to your local machine:

   ```
   git clone git@github.com:YOUR_USER_NAME/dftbplus.git
   ```

3. Register the official repository as *upstream* in your git clone:

   ```
   git remote add upstream git@github.com:dftbplus/dftbplus.git
   ```

1.2.2 Set up your own repository

Set up your identity

When you contribute to our project it is important that the author information of your commits contain your full name and a valid (preferably your official) email address. Set up those for your repository (or globally by adding the `--global` option) by

```
git config user.name 'FULL_NAME'
git config user.email 'EMAIL_ADDRESS'
```

Add check on commit message formatting

We use the commonly adopted git commit message format containing a short imperative subject line and an optional detailed description which is separated by an empty line (see for example How to Write a Git Commit Message). Using a simple commit message hook, git can check that your commit messages follow this format. Please copy our special git commit hook as *.git/hooks/commit-msg* into your repository and make it executable (`chmod +x .git/hooks/commit-msg`).

You may wish to make this a global hook for all of your git repositories by adding it to an init.templatedir directory. This can be added for *all* repositories with

```
git config --global init.templatedir '~/.git-templates'
mkdir -p ~/.git-templates/hooks
```

The commit-msg file can then be placed in *~/.git-templates/hooks/commit-msg*. We would then also suggest setting the permission to be user writable only

```
chmod -R 700 ~/.git-templates
```

Any new local repositories will settings specified from this directory, unless overridden by a local *.git/* directory within the repository itself. Existing repositories need to be reinitialised in their top directory to use the init.templatedir

```
git init
```

Again, any local *.git/* directory overrides settings in *~/.git-templates*

1.3 Synchronising to the upstream main branch

Before you start developing a feature, you should make sure that you implement your feature in the most recent version of the code. This minimises the chances of conflicts (and additional work needed from you) when your feature is later merged into the upstream repository:

1. Pull the recent changes from the upstream *main* branch into your local *main* branch:

   ```
   git checkout main
   git pull --ff-only upstream main
   ```

   Upload the changes in your local *main* branch to GitHub by issuing:

   ```
   git push origin main
   ```

   Note: if the `git pull --ff-only upstream main` command fails, you have probably polluted your personal main branch, and it can no longer be made to exactly match the upstream one. In that case, you may revert it via a hard reset and then pull the current upstream/main again:

   ```
   git reset --hard upstream/main
   git pull --ff-only upstream master
   ```

1.4 Developing your feature

1. Check out your *main* branch, which you should have synchronised to upstream *main* as described in the previous section:

   ```
   git checkout main
   ```

2. Create you own feature branch:

   ```
   git checkout -b some-new-feature
   ```

   To develop a new feature you should always create a new branch derived from *main*. You should never work on the *main* branch directly, or merge anything from your feature branches onto it. Its only purpose is to mirror the status of the upstream *main* branch. The feature branch name should be short and descriptive for the feature you are going to implement.

3. Develop your new feature in your local branch. Make sure to add regression testing for your feature in the test directory and update the documentation. You can commit your changes by

   ```
   git commit -m "Add some new feature ..."
   ```

   You may make multiple commits if your development naturally dividides into multiple steps. But please note, that too many commits and especially commits containing broken or non-functional code make finding bugs (e.g. by git-bisection) a real pain. Therefore, try to make sure that your branch only contains *essential commits with working code in each commit*.

   In case, you wish to remove some intermediate commits in your feature branch, you may use the interactive rebasing:

   ```
   git rebase -i HEAD~N
   ```

where `N` should be replaced by the number of commits you would like to rearrange/squash. As interactive rebasing changes the git-history, make sure that

- you only squash commits of your feature branch, no earlier ones,
- you squash your commits before any other branches had been derived from your feature branch and
- you squash your commits before any other branches have been merged into your feature branch.

1.5 Merge the changes back into the upstream repository

First, make sure, that your feature integrates well into the most recent main code version. Be aware that the upstream code may have evolved while you were implementing your feature.

1. Synchronise your *main* branch to the upstream *main*, as written in the section *Synchronising to the upstream main branch*.
2. Integrate any changes that appeared on *main* during your feature development.
   - If your feature branch consists of only one or two commits, it does not contain any merge-commits and no other branches had been derived from it (and you are an experienced git user) you may rebase your branch on current *main*:
     - Check out your feature branch:

       ```
       git checkout some-new-feature
       ```

     - Rebase it on *main*:

       ```
       git rebase main
       ```

       Resolve any conflicts arrising during the rebase process.
   - Otherwise use a normal merge to update your feature branch with the latest development on main:
     - Check out your feature branch:

       ```
       git checkout some-new-feature
       ```

     - Merge the *main* branch into it:

       ```
       git merge main
       ```

     This will result in an extra merge commit.
3. Test whether your updated feature branch still works as expected (having regression tests for your feature can help here).
4. Publish your feature branch to your personal repository on GitHub:

   ```
   git push origin some-new-feature
   ```

5. Issue a pull request on GitHub for your *some-new-feature* branch (look for the upwards arrow in the left menu).
6. Wait for the comments of other developers, apply any fixes you are asked to make, and push the changes to your feature branch on GitHub.
7. Once the discussion on your pull request is finished, one of the developers with write permission to the upstream repository will merge your branch into the upstream *main* branch. Once this has happened, you should see your changes showing up there.

1.6 Delete your feature branch

Once your feature has been merged into the upstream code you should delete your feature branch, both locally and on GitHub as well:

1. In order to delete the feature branch locally, change to the *main* branch (or any branch other than your feature branch) and delete your feature branch:

   ```
   git checkout main
   git branch -d some-new-feature
   ```

2. In order to delete the feature branch on GitHub as well use the command:

   ```
   git push origin --delete some-new-feature
   ```

This closes the development cycle of your feature and opens a new one for the next one you are going to develop. You can then again create a new branch for the new feature and develop your next extension starting with the steps described in section *Synchronising to the upstream main branch*.

1.7 A few notes about Submodules

The DFTB+ program uses several libraries from elsewhere in the project. Some of those libraries (e.g. MpiFx, ScalapackFx, libNEGF, libMBD) are included within the repository via the git *submodule* mechanism.

1.7.1 Checking out submodules

When checking out the code, you should pull the submodules with

```
git submodule update --init --recursive
```

1.7.2 Updating submodules after changing to a branch

If you switch between branches, the branch you change into may reference a different commit of a submodule than the branch you just have left. You can recognise this by looking at the status of the submodules after the branch change, e.g. by issuing

```
git status
```

The directories containing affected submodules will have their status set to be "modified". These submodules must be realigned to the correct commit (to the commit recorded for the current branch) before you do any other work in the branch. You can do this for all submodules by issuing

```
git submodule update --recursive
```

If both the submodule commit id and also the repository URL for the submodule change when switching to a new branch, you will have to synchronise the repository URLs first before doing the update, e.g.

```
git submodule sync --recursive
git submodule update --recursive
```

1.7.3 Changing submodule content

If you need to modify the submodules, you should fork their respective projects and work according their development workflow (remember, that in several cases these are projects developed and maintained by groups not part of the DFTB+ team).

If you want to update DFTB+ to use a new version of a given submodule, do the following steps:

1. Go to the `origin` folder containing the submodule.
2. Fetch the relevant branch from the upstream-project of the submodule.
3. Check out the commit which should be used by DFTB+. (If this commit is on a different branch from the one recorded in the *.gitmodules* file in the DFTB+ source folder, make sure to correct the branch name there.)
4. Update the submodule commit ID's (recorded in *CMakeFiles.txt*) by executing

   ```
   ./utils/test/check_submodule_commits -u
   ```

   from the DFTB+ source folder.
5. Stage the submodule folder and the *CMakeFiles.txt* file for a commit and commit your changes.

1.7.4 Referencing submodules

Since the code should be available for users without accounts on github.com, all submodules are included as web (https) links instead of ssh references.

If you work on the integration of the submodules, you might find it useful to globally configure git to substitute ssh links for the https references by issuing the command

```
git config --global url.ssh://git@github.com/.insteadOf https://github.com/
```

You can alternatively set up this substitution for only your local *dftbplus* repository. You should run this command in the directory containing your copy and leave out the `--global` option.

CHAPTER

TWO

WORKFLOW FOR MAINTAINERS

In order to ease the process of maintaining, the following strategy is applied for the development of DFTB+ and related projects.

2.1 Roles

We have three groups of developers with different access rights to the repositories:

- Coordinating developers (repository administrators): They are responsible for strategic decisions. Some of them also represent the project externally.
- Core developers (repository maintainers): Experienced, skilled developers contributing regularly to the projects and having a good overview of the project. They are often responsible for certain sets of features within the code.
- Contributors: casual contributors.

2.2 Access Rights

The groups as listed above have following rights:

- Repository administrators: They have all possible rights on the repositories, including the possibility to grant rights to other developers.
- Repository maintainers: They have write access to project repositories. They can review and merge pull requests (PR). They can write into the corresponding repository of contributors during the review process. They have access to the test server, so that they can execute extended tests on the code in a pull request.
- Contributors: They have the usual read and fork rights as granted by GitHub and can of course submit pull requests.

2.3 Merging pull requests

The following rules are currently applied when merging pull requests:

- All changes to the master branch should be made via pull requests.
- Pull requests must be approved by a maintainer before they can be merged. Pull requests issued by a maintainer must be approved by another maintainer.
- Trivial pull requests can be merged by any maintainer, provided self-submitted code has been approved by another maintainer.

- Pull requests with non-trivial or substantial changes (major re-structuring, changes in code behavior, large functionality additions, etc.) must be discussed with and approved by an admin before merging.
- Pull requests should pass the extended manually triggered buildbot tests (not just the automatic Travis tests triggered by GitHub) before they can be merged.
- Both, the person making the pull request and the person merging the pull request are responsible for the quality and the impact of the merged changes.

2.4 Merging strategy

Following strategy should be applied when merging the pull request:

- If the PR contains a **single commit** only and is based on current master, it can be merged by fast-forward.
- In all other cases, a merge commit should be created.
- The code in the PR should be brought up to date with master by the contributors before the review. The contributors are of course free to decide, whether they accomplish this via a merge (merging master into the feature branch) or via a rebase (rebasing the feature branch on current master – only for skilled contributors!). However, once the review has started, no further rebases of the feature branch should happen.

2.5 Packaging releases

A DFTB+ release is created by an annotated tag in the git repository either using the online interface provided by GitHub or the command line with

```
git tag -as <version>
```

This task is usually performed by repository administrators.

2.5.1 Updating the conda-forge feedstock

Before a release tag is create in the main git repository the respective commit should be tested with the conda-forge toolchain. To do this without a local conda installation, fork the respective feedstock repositories from the conda-forge organisation (*e.g.* if you plan to make a release in the https://github.com/dftbplus/mpifx, you can find the feedstock at https://github.com/conda-forge/mpifx-feedstock) and update the `recipe/meta.yaml` file to the release commit.

For a detailed workflow visit the conda-forge documentation about updating feedstocks.

To test a potential commit, download the commit using the git archive function and determine its SHA256 checksum with

```
> wget https://github.com/dftbplus/dftbplus/archive/<commit>.tar.gz -O dftbplus.tar.gz
...
> sha256sum dftbplus.tar.gz
```

Update the `url` and `sha256` in the source field of the recipe.

Listing 1: recipe/meta.yaml

```yaml
source:
  url: https://github.com/dftbplus/{{ name }}/archive/<commit>.tar.gz
  sha256: <sha256>
```

Also, you want to reset the build number and bump the version number to the prospective release while already on it. Check the `patches` section and remove any patch already applied in master.

If you have a local conda-smithy toolchain available, rerender the feedstock

```
> conda update conda-build conda-smithy conda-forge-pinning
> conda smithy rerender
```

Read the output of the rerender step carefully, than commit your changes including the maintenance line suggested by conda-smithy and push to your fork of the feedstock. If you do not have a local conda-smithy toolchain available commit anyway and push to your fork, but request the conda-forge webservice to rerender for you when creating the pull request.

Follow the usual GitHub workflow to create pull request against the feedstock repository, *read* the pull request template and tick of all points you have done, use a strikeout to remove irrelevant points. Adding an additional unticked item like *bump to release tag* seems prudent. If you have not yet rerendered the feedstock, add the suggested line to request the conda-forge webservice to rerender for you.

After successfully creating the pull request, wait for the friendly conda-forge webservices to comment into your pull request, usually the linter and, if you requested to rerender, also the webservice bot will comment on your pull request. Now, you can check the continuous integration runs at Azure pipelines, Travis CI and/or Drone CI. For DFTB+ this can take several minutes to build all the possible targets. Everything should pass before you finally create the release tag and insert the correct release URL and SHA256 hash. Adding the automerge label is also a possibility to let the conda-forge webservice handle the feedstock update for you after the CI is passing. In case the build fails inspect the logs and open an issue at the upstream repository, than go fix the issue and repeat from the beginning.

Note, some feedstocks are split into several packages, for example the Python API of DFTB+ is separated from the main DFTB+ package, in this case you have to update several feedstocks at once.

Setting up a local conda-forge toolchain

To bootstrap a new conda environment tailored for conda-forge start by installing miniforge with the correct installer from the latest miniforge release. After installing conda setup a conda-build toolchain by installing

```
> conda install conda-build conda-smithy conda-forge-pinning
```

Enter the feedstock you want to build and start conda-build with

```
> conda build recipe
```

After the build has finished successfully you can install the freshly built package (assuming you used the dftbplus-feedstock) in a new environment to test it

```
> conda install dftbplus --use-local -mn dftbplus
> conda activate dftbplus
> which dftb+
/home/<user>/miniforge3/envs/dftbplus/bin/dftb+
```

Giving the complete spec as `<name>=<version>=<hash>` might be necessary to get the locally built package installed, you can find the complete specs in the conda-build output.

Running conda build with docker

Alternatively you can run the conda-forge builds in docker containers to avoid your local development environments to pollute the conda-build. Check the conda-forge namespace at docker-hub for suitable containers, at the time of writing the `comp7` toolchain is in use for Linux. Pull the container and create an instance, you might also want to enter it interactively to check the conda-build result afterwards:

```
> sudo docker pull conda-forge/linux-anvil-comp7
> sudo docker run -it --rm -v /path/to/feedstock/recipe:/home/conda/recipe conda-forge/
↪linux-anvil-comp7
$ conda build recipe
```

Run the last command inside the container. After the build has finished successfully you can install the freshly built package (assuming you used the dftbplus-feedstock) in the container and test it with

```
$ conda install dftbplus --use-local
$ which dftb+
/opt/conda/bin/dftb+
```

Giving the complete spec as `<name>=<version>=<hash>` might be necessary to get the locally built package installed. Remember that you are probably missing most of your development toolchain inside the container, therefore, add directories with Slater–Koster files or input files to the container by mounting the additional directories with the `-v` option in advance.

CHAPTER

THREE

FORTRAN STYLE GUIDE

The main principle when contributing code to Fortran projects should be readability. If your code can not be easily read and understood by others it will be hard to maintain and extend. It should also fit well with the existing parts of the code (in style as well as in its programming paradigms) maintaining the principle of least surprise.

Below you will find some explicit coding rules we try to follow. The list can not cover all aspects, so also look at the existing source code and try to follow the conventions being used.

If you use Emacs as editor, consider adding appropriate customisation settings to your config file in order to automatically enforce some of the conventions below. You may also wish to use ws-butler.

3.1 Line length and indentation

- Maximal **line length** is **100** characters. For lines longer than that, use continuation lines.
- **Nested blocks** are indented by **2** white spaces:

```
write(*, *) "Nested block follows"
do ii = 1, 100
  write(*, *) "This is the nested block"
  if (ii == 50) then
    write(*, *) "Next nested block"
  end if
end do
```

- **Continuation lines** are indented by **4** white spaces. Make sure to place continuation characters (&) both at the end of the line as well as at the beginning of the continuation line:

```
call someRoutineWithManyParameters(param1, param2, param3, param4,&
    & param5)
```

  Try to break lines at natural places (e.g. at white space characters) and include one white space character after the opening ampersand in the continuation line.

- **Single line preprocessor directives** are indented as normal code:

```
@:ASSERT(someCondition)
call someRoutine(...)
```

- **Preprocessor block directives** (directives with starting and ending constructs) are outdented by **2** characters with respect of the code they enclose. The enclosed code must be aligned as if the preprocessor directives were not present:

```
  call doSomething()
#:if WITH_SCALAPACK
  call someRoutineScalapackVersion(...)
#:else
  call someRoutineSerialVersion(...)
#:endif

do iKS = 1, nKS
#:if WITH_SCALAPACK
  call someRoutineScalapackVersion(iKS, ...)
#:else
  call someRoutineSerialVersion(iKS, ...)
#:endif
end do
```

3.2 Naming

The naming conventions basically follow those in the Google Style Guide for Java naming convention, with minor modifications.

- **Variable** names follow the **lowerCamelCase** convention:

```
logical :: hasComponent
```

- **Constants** (parameters) use the **lowerCamelCase** convention similar to variables

```
integer, parameter :: maxArraySize = 100
```

  with the exception of the constants used to define the kind parameter for intrinsic types, which should be all lowercase (and short):

```
integer, parameter :: dp = kind(1.0d0)
real(dp) :: val
```

- **Subroutine** and **function** names follow also the **lowerCamelCase** notation:

```
subroutine testSomeFunctionality()
myValue = getSomeValue(...)
```

- **Type** (object) names are written **UpperCamelCase**:

```
type :: TRealList
type(TRealList) :: myList
```

  All type names should be prefixed with a capital 'T', in order to clarify the distinction between type names and variable names:

```
type :: TBroydenMixer
:
end type TBroydenMixer
:
type(TBroydenMixer) :: broydenMixer
```

- **Instances** referenced out of type-bound procedures are to be named *this*:

```
subroutine typeBoundProcedure(this, ...)
  class(TType), intent(inout) :: this
  :
end subroutine typeBoundProcedure
```

- **Module** names follow **lower_case_with_underscore** convention:

```
use dftb_common_accuracy
```

  Underscores are used for name-spacing only, so the module above would be typically found at the path *dftb/common/accuracy.f90*. The individual component names (`dftb`, `common`, `accuracy`) may not contain any underscores and must be shorter than 15 characters.

- **Preprocessor** variables and macros follow **UPPER_CASE_WITH_UNDERSCORE** convention:

```
#:if WITH_MPI
  withMpi = ${FORTRAN_LOGICAL(WITH_MPI)}$
#:endif
```

3.3 White spaces

Please use white spaces to make the code readable. In general, you **must use** white spaces in following situations:

- Around arithmetic operators:

```
2 + 2
```

- Around assignment and pointer assignment operators:

```
aa = 3 + 2
pWindow => array(1:3)
```

- Around the `::` separator in declarations:

```
integer :: ind
```

- After commas (`,`) in general and especially in declarations, calls and lists:

```
real(wp), allocatable :: array(:)
type, extends(TBaseType) :: TDerivedType
subroutine myRoutine(par1, par2)
call myRoutine(val1, val2)
print *, 'My value:', val
do ii = 1, 3
array(1:3) = [1, 2, 3]
```

- When separating array indices, when the actual index value for an index contains an expression:

```
myArray(ii + 2, jj) = 12
```

You **may omit** white space in following cases:

- When separating array indices and the actual index values are simple and short (typically two letters) variable names, one or two digit integers or the range operator `::`

  ```
  myArray(:,1) = vector
  latVecs(1,1) = 1.0_wp
  myArray(ii,jj) = myArray(jj,ii)
  ```

You **must omit** white spaces in following cases:

- Around opening and closing braces of any kind:

  ```
  call mySubroutine(aa, bb)  ! and NOT call mySubroutine( aa, bb )
  myVector(:) = [1, 2, 3]    ! instead of myVector(:) = [ 1, 2, 3 ]
  tmp = 2 * (aa + bb)        ! instead of 2 * ( aa + bb )
  ```

- Around the equal (=) sign, when passing named arguments to a function or subroutine:

  ```
  call mySubroutine(aa, optionalArgument=.true.)
  ```

- Around the power operator:

  ```
  val = base**power   (instead of val = base ** power)
  ```

Avoid white spaces for **visual aligning** of code, use:

```
integer, intent(in) :: nNeighbors
real(wp), intent(out) :: interaction
```

instead of:

```
integer, intent(in)   :: nNeighbors
real(wp), intent(out) :: energy
```

Although latter may look more readable, it makes rather difficult to track real changes in the code with the revision control system. For example when a new line is added to the block making the realignment of previous (but otherwise unchanged) lines necessary

```
integer, intent(in)             :: nNeighbors
real(wp), intent(out)           :: energy
real(wp), intent(out), optional :: forces(:)
```

the version control system will indicate all of those lines having been modified, although only the alignment (but not the actual instructions) were changed.

3.4 Comments

- **Module**, **Subroutine** and **function** comments should be consistent with doxygen / FORD literate comments for publicly visible interfaces and variables.
- Variable/module/routine comments on multiple lines should use a double-bang for the second and subsequent lines, with the first line capitalised

  ```
  !> This is a multi-line comment for something,
  !! it continues on a second line.
  ```

- Adapt your comment if it starts with a term that is expected to be in lower-case

  ```
  !> The k-point
  ```

  instead of

  ```
  !> K-point
  ```

- Comments are indented to the same position as the code they document:

  ```
  ! Take spin degeneracy into account
  energy = 2.0_wp * energy
  ```

- Generally, write the comment *before* the code snippet it documents:

  ```
  ! Loop over all neighbours
  do iNeigh = 1, nNeighbours
    :
  end do
  ```

- Try to avoid mixing code and comments within one line as this is often hard to read:

  ```
  bb = 2 * aa   ! this comment should be before the line.
  ```

- Never use multi-line suffix comments, as an indenting editor would mess up the indentation of subsequent lines:

  ```
  bb = 2 * aa  ! This comment goes over multiple lines, therefore, it
               ! should stay ALWAYS before the code snippet and NOT HERE.
  ```

- Specifically comment any workarounds, include the compiler name and the version number for which the workaround had to be made. Always use the following pattern, so that searching for workarounds which can be possibly removed is easy:

  ```
  ! Workaround: gfortran 4.8
  ! Finalisation not working, we have to deallocate explicitly
  deallocate(myPointer)
  ```

- Comments should always start with one bang only. Comments with two bangs are reserved for source code documentation systems:

  ```
  ! This block needs a documentation
  do ii = 1, 2
    :
  end do
  ```

- If you need a comment for a longer block of code, consider instead packaging that block of code into a properly named function (if the additional function call would be performance critical, write it as an internal procedure):

  ```
  somePreviousStatement
  ind = getFirstNonZero(array)
  someStatementAfter
  ```

  instead of

```
somePreviousStatement

! Look for the first nonzero element
found = .false.
do ind = 1, size(array)
  if (array(ind) > 0) then
    found = .true.
    exit
  end if
end do
if (.not. found) then
  ind = 0
end if

someStatementAfter
```

3.5 Block constructs

- Block constructs are normally used in their verbose form, with block opening and corresponding block closing in separate lines:

```
if (some_conditions) then
  ! Do something
  ...
end if
```

- The closing form should have a space between the `end` keyword and the construct type:

```
do ii = 1, 10
  ...
end do  ! instead of "enddo"
```

- If the block construct contains an expression within obligatory parentheses, insert one space between the block type and the opening parenthesis:

```
if (some_condition) then  ! instead of "if(some_condition)"

where (aa == 0)           ! instead of "where(aa == 0)"
```

- Some block constructs have alternative one-line short forms without closing statements (e.g. `if`, `where`). Only use their short form, if it is readable and fits into a single line:

```
if (ioStat /= 0) return

if (allocated(someArray)) someArray(:,:) = 0.0_dp

where (abs(aa) >= epsilon(0.0_dp)) aa = 1.0_dp / aa
```

3.6 Allocation status

At several places, the allocation status of a variable is used to signal choices about logical flow in the code:

```
!> SCC module internal variables
type(TScc), allocatable :: sccCalc
.
.
.
if (allocated(sccCalc)) then

end if
```

This is to be preferred to the use of additional logical variables if possible.

Part of the reason for this choice is that from Fortran 2008 onwards, optional arguments to subroutines and functions are treated as not-present if not allocated.

3.7 File I/O

All files must be opened (i.e., connected to a descriptor) by the `openFile()` routine, which initializes a `type(TFileDescriptor)` instance. Whenever possible, use the `mode` argument to specify the file opening type:

```
call openFile(fd, "test.dat", mode="r")
```

The `mode` specifier accepts the following possible options:

- `r`: read (file must exist, the descriptor is at the start of the file contents),
- `r+`: read and write (file must exist, the descriptor is at the start of the file contents),
- `w`: write (file will be replaced if already existing, otherwise created)
- **`w+`: read and write (file will be replaced if it already exists,**
  otherwise created)
- `a`: appended write (file will be opened if it already exists, otherwise created; the descriptor will be positioned at its end)
- `a+`: appended read and write (file will be opened if it already exists, otherwise created; the descriptor will be positioned at its end)

Additionally the letter `b` can be appended to open the file in binary (unformatted) mode (e.g. `rb` for reading a binary file or `a+b` for appending to a binary file in read and write mode).

For reading, writing and rewinding, the `%unit` field of the descriptor should be used. Do not change the value of `%unit`. Do not close the file with the `close` statement, but use the `closeFile()` routine instead. (Actually, files are automatically closed, if the connected descriptor leaves code scope, but for better readability of the code, we close them explicitely by calling the `closeFile()` routine.) Calling `closeFile()` with an unconnected descriptor is fine, it will simply do nothing. This should allow you to eliminate most guarding `if` statements around any `closeFile()` calls.

CHAPTER

FOUR

PYTHON COMPONENTS OF DFTB+

The PEP 8 guidelines are the basis of the python style used in the code.

Pylint files are suplied (*utils/srccheck/pylint/*) for code checking. In many cases, python unit tests are also implemented (see *test/tools/dptools/* for examples).

For example to test the straingen script for Python3 compliance, from the top of the repository type:

```
env PYTHONPATH=$PWD/tools/dptools/src pylint3 --rcfile \
utils/srccheck/pylint/pylintrc-3.ini tools/dptools/bin/straingen
```

while to test for correct performance:

```
make test_dptools
```

CHAPTER

FIVE

CONTRIBUTING DFTB+ RECIPES

There are a number of more detailed examples for the use of DFTB+ in the recipes repository, which is generated on readthedocs. The recipes are written in the ReStructuredText language, and in most cases include downloadable input files for the examples.

Sphinx version 1.8 or above is required to locally generate the recipes. We find that using a virtual environment for python is convenient.

5.1 Repository structure

The main directories in the repository are

- *docs/* Contains sub-folders for examples (i.e. *basics/*)
- ***docs/_archives/recipes/* Contains sub-folders for downloadable input files**
  and scripts
- *docs/_figures/* Contains sub-folders for images in the recipes

5.2 Downloadable files

Constructions like

```
[Input: `recipes/basics/bandstruct/1_density/`]
```

are used to specify the path of files inside the *_archives* directory when HTML documents are generated. This is independent of the actual packaging of the files in *docs/_archives/recipes/* into a downloadable file, so should be manually checked.

CHAPTER

SIX

LICENCE