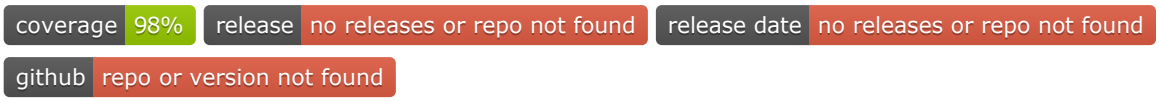
DCR - Development



1. System Environment

DCR is developed on the operating systems **Ubuntu 20.04 LTS** and **Microsoft Windows 10**. Ubuntu is used here via the **VM Workstation Player 16**. **Ubuntu** can also be used in conjunction with the **Windows Subsystem for Linux (WSL2)**.

The GitHub actions for continuous integration run on **Ubuntu 18.04**, **Ubuntu 20.04**, **Micrsoft Windows Server 2019** and **Micrsoft Windows Server 2022**.

Version **3.10** is used for the **Python** programming language.

2. Coding Standards

2.1 **Python**

- The PEP 8{:target="_blank"} style guide for **Python** code is strictly applied and enforced with static analysis tools.
- All program code must be commented with type hinting instructions.
- All functions, modules and packages must be commented with **Docstring**.
- The program code must be covered as far as possible with appropriate tests - the aim is always 100 % test coverage.
- The successful execution of **make dev** ensures that the program code meets the required standards.

2.2. Scripts

- Scripts must always be available in identical functionality for both the Unix shell **bash** and the Windows command interpreter **cmd.exe**.
- The most important dynamic parameters of a script should be requested from the user in a dialogue.
- In the event of an error, the execution of the script must be terminated immediately.
- Apart from the main script, all other scripts should be present in the **scripts** file directory.

3. Code Formatting

The two tools **isort** and **Black** are used for formatting the programme code:

- Black{:target="_blank"} - The uncompromising **Python** code formatter.
- isort{:target="_blank"} - A **Python** utility / library to sort imports.

Both tools are included in the call **make dev**. They can be executed individually with **make black** and **make isort**, whereby **sort** should run first and **black** afterwards.

4. Static Code Analysis

The tools **Bandit**, **Flake8**, **Mypy** and **Pylint** are used for static code analysis:

- Bandit{:target="_blank"} - **Bandit** is a tool designed to find common security issues in **Python** code.
- Flake8{:target="_blank"} - A **Python** tool that glues together **pycodestyle**, **Pyflakes**, **McCabe**, and third-party plugins to check the style and quality of some **Python** code.
- Mypy{:target="_blank"} - Optional static typing for **Python**.
- Pylint{:target="_blank"} - It's not just a linter that annoys you!

All tools are included in the call **make dev**. They can be executed individually with **make bandit**, **make flake8**, **make mypy** and **make pylint**.

Flake8 includes the following tools:

- McCabe{:target="_blank"} - McCabe complexity checker for **Python**.
- pycodestyle{:target="_blank"} - Simple **Python** style checker in one **Python** file.
- Pyflakes{:target="_blank"} - A simple program which checks **Python** source files for errors.
- Radon{:target="_blank"} - Various code metrics for **Python** code.

5. Software Documentation

5.1 API Documentation

The creation of API documentation for functions, modules and packages is mandatory and enforced with the static analysis tool pydocstyle{:target="_blank"}. **pydocstyle** is a static analysis tool for checking compliance with **Python Docstring** conventions. **pydocstyle** can be executed individually with **make pydocstyle** and is also included in the call **make dev**.

The **Docstring** format used in **DCR** is that of type Google. For Visual Studio Code, the extension VSCode Python Docstring Generator{:target="_blank"} can be used when creating API documentation.
With the Pydoc-Markdown{:target="_blank"} tool, the API documentation is extracted from the source files and put into Markdown format. In this format, the API documentation can then be integrated into the user documentation.

Examples for the format of the API documentation:

Package Documentation:

```
Package: DCR libraries.
```

Module Dokumentation:

```
Check and distribute incoming documents.

New documents are made available in the file directory inbox.
These are then checked and moved to the accepted or
rejected file directories depending on the result of the check.
Depending on the file format, the accepted documents are then
```

```
converted into the pdf file format either with the help of Pandoc
or with the help of Tesseract OCR.
```

Function Documentation:

```
Load the command line arguments into memory.

The command line arguments define the process steps to be executed.
The valid arguments are:

    all   - Run the complete processing of all new documents.
    d_c_u - Create or upgrade the database.
    p_i   - Process input folder.

With the option all, the following process steps are executed
in this order:

    1. d_c_u
    2. p_i

Args:
logger (logging.Logger): Current logger.
argv (List[str]): Command line arguments.

Returns:
dict[str, bool]: The command line arguments found.
```

In Visual Studio Code, the VSCode Python Docstring Generator{:target="_blank"} tool can be used to create a framework for API documentation.

5.2 User Documention

The remaining documents for the user documentation can be found in the file directory **docs** in Markdown format:

File	Headline	Remarks
code_of_conduct.md	Code of Conduct	
contributing.md	Contributing Guide	
development.md	Development	Notes on the software development process
index.md	Document Content Recognition	Background, installation and user guide
release_notes.md	Release Notes	
research.md	Research	Reference to the relevant research papers

The MkDocs{:target="_blank"} tool is used to create the user documentation. With the command **`make mkdocs`** the user documentation is created by MkDocs and uploaded to the GitHub pages of the repository. The command **`make mkdocs`** is also included in the call **`make dev`**.

6. Software Testing

pytest{:target="_blank"} is used as a software testing framework with the following plugins::

- pytest-cov{:target="_blank"} for coverage reporting,
- pytest-deadfixture{:target="_blank"} to list unused or duplicate fixtures, and
- pytest-random-order{:target="_blank"} to randomise the order of the tests.

On the one hand, the tests must be as complete as possible, i.e. a test coverage of 100% is aimed for, but on the other hand, the scope of the test code should be minimal, i.e. unnecessary repetitions must be strictly avoided. The best strategy for this is to first create a test case for the normal case and then add special tests for the special cases not yet covered.

Finally, the tool Coveralls for Python{:target="_blank"} is used to enable a connection to Coveralls{:target="_blank"}.

7. Continuous Delivery

The GitHub Actions are used to enforce the following good practices of the software engineering process in the CI/CD process:

- uniform formatting of all source code,
- static source code analysis,
- execution of the software testing framework, and
- creation of up-to-date user documentation.

The branch **`Development Standards`** in the GitHub Actions guarantees compliance with the required standards, the branch **`Production`** ensures error-free compilation for production use and the branch **`Test Framework`** runs the testsd against various operating system and **`Python`** versions. The branches **`Production`** and **`Test Framework`** must be able to run error-free on operating systems **`Ubuntu 18.04`**, **`Ubuntu 20.04`**, **`Micrsoft Windows Server 2019`** and **`2022`** and with **`Python`** version **`3.10`**, the branch **`Development Standards`** is only required error-free for the latest versions of **Ubuntu** and **`Python`**.

The individual steps to be carried out

1. in the branch **`Development Standards`** are:
    1. set up **`Python`**, **`pip`** and **`pipenv`**
    2. install the development specific packages with **`pipenv`**
    3. compile the **`Python`** code
    4. run isort for sorting your imports
    5. run Black for formatting your code
    6. run Bandit for finding common security issues
    7. run Flake8 for style guide enforcement
    8. run MyPy for static typing
    9. run PyLint for checking for errors in **`Python`** code

10. run pydocstyle for docstring style enforcement
11. run MkDocs for creating project documentation with markdown.
12. publish the code coverage results to **`coveralls.io`**

2. in the branch **`Production`** are:

    1. set up **`Python`**, **`pip`** and **`pipenv`**
    2. install the production specific packages with **`pipenv`**
    3. compile the **`Python`** code

3. in the branch **`Test Framework`** are:

    1. set up **`Python`**, **`pip`** and **`pipenv`**
    2. install the development specific packages with **`pipenv`**
    3. compile the **`Python`** code
    4. run pytest for writing better program

8. Development Environment

To set up a suitable development environment under **`Ubuntu 20.04 LTS`**, on the one hand a suitable ready-made Docker image is provided and on the other hand two scripts to create the development system in a standalone system, a virtual environment or the **`Windows Subsystem for Linux (WSL2)`** are available.

8.1 Docker Image

The ready-made Docker images are available on DockerHub{:target="_blank"} under the following link:

dcr_dev - Document Content Recognition Development Image{:target="_blank"}

When selecting the Docker image, care must be taken to select the appropriate version of the Docker image.

8.2 Script-based Solution

Alternatively, for a **`Ubuntu 20.04 LTS`** environment that is as unspoiled as possible, the following two scripts are available in the **`scripts`** file directory:

- **`scripts/run_install_4-vm_wsl2_1.sh`**
- **`scripts/run_install_4-vm_wsl2_2.sh`**

After a **`cd scripts`** command in a terminal window, the script **`run_install_4-vm_wsl2_1.sh`** must first be executed. Administration rights (**`sudo`**) are required for this. Afterwards, the second script **`run_install_4-vm_wsl2_2.sh`** must be executed in a new terminal window.

9. Version Planning

Version	Feature(s)
0.5.0	Inbox processing
0.6.0	Pandoc processing
0.7.0	Tesseract OCR processing

Version	Feature(s)
0.8.0	PDFlib TET processing
0.9.0	Parser

10. Next Development Steps

1st Priority:

- API Documentation
- error handling - highly defensive
- inbox.py - process_inbox() - processing ocr & non-ocr in the same method
- tools.py - verify the content of the inbox directories

2nd Priority:

- Tesseract OCR
    - Installation

3rd Priority

- n/a

No Priority (yet)

- n/a