The texgit Package

Thomas Weise (汤卫思教授)
Institute of Applied Optimization (应用优化研究所, IAO)
School of Artificial Intelligence and Big Data (人工智能与大数据学院)
Hefei University (合肥大学)
Hefei 230601, Anhui, China (中国安徽省合肥市)
tweise@hfuu.edu.cn · tweise@ustc.edu.cn

June 26, 2025

Abstract

This package allows you to download and access files that reside in a `git` repository from within your LaTeX code. This can be used, for example, to include program code from an actual software in life repository in your LaTeX documents. It allows you to postprocess these files, e.g., to apply programs that remove comments or reformat code and then to include these postprocessed files. It furthermore allows you to execute programs (or scripts from `git` repositories) on your machine and include their output into your LaTeX documents. Finally, it also allows you to allocate files and pass them as parameters to the programs that you execute. With this, you could create PDF figures on the fly and then include them into your LaTeX documents.

This LaTeX package works *only* in combination with the Python package `texgit`. To implement its functionality, it offers the following commands:

- `\gitLoad{id}{repoURL}{pathInRepo}{postproc}` loads a file `pathInRepo` from the `git` repository `repoURL`, *optionally* post-processes it by piping its contents into the standard input of a command `postproc` capturing its standard output.
- `\gitFile{id}` provides a local path to a file created this way. Using the `\gitFile{id}` macro, you can then include the file in LaTeX directly or load it as source code listing.
- `\gitName{id}` provides the name of the file created this way.
- `\gitNameEsc{id}` provides the name of the file created this way, but with the characters `_`, `$`, and escaped to `\_`, `\$`, and `~`, respectively.
- `\gitUrl{id}` provides the URL to the original file in the `git` repository.
- `\gitExec{id}{repoURL}{pathInRepo}{command}` executes an arbitrary command `command`, either in the current directory or inside a directory `pathInRepo` of the `git` repository `repoURL` and fetches the standard output into a local file, the path to which is made available to the file again as macro `\gitFile{id}`. You can also leave the `id` parameter empty. This makes sense if you are not interested in the output of the program. Maybe you just want to execute a script, say, something that creates a figure stored in some file. Without `id` parameter, you cannot access the stdout of this command. And sometimes, this is ok.

- `\gitArg{id}{prefix}{suffix}` allocates an additional file, whose name will be composed of the optional `prefix` and `suffix`. Such files can be passed as arguments to `\gitExec` or `\gitLoad` by including `(?id?)` in their commands' argument list. This way, we can, for example, instruct a program to create a graphic and store it in a certain file that we can later load from `\gitFile{id}`.
- `\gitIf{id}{ifDone}{ifNotDone}` executes the code `ifDone` starting in the second `pdflatex` pass, i.e., after the Python `texgit` package has been applied to the `aux` file generated during the first `pdflatex` pass. During the first `pdflatex` pass and before the Python `texgit` package was applied, `ifNotDone` will be executed.

The functionality of the package is implemented by storing the `git` requests in the `aux` file of the project during the first `pdflatex` pass. The `aux` file is then processed by the Python package which performs the actual `git` queries, program executions, stores the result in local files, and adds the resolved paths to the `aux` file. Thus, during the first `pdflatex` run, `\gitFile` and `\gitUrl` offer dummy results. During the second and later pass, after the Python program `texgit` has been applied to the `aux` file, they then provide the actual paths and URLs. In the essense, `texgit` works somewhat like BibTeX.

Contents

1 Introduction

1.1 Addressed Problem and Use Case

Let's say you want to make teaching material in the field of computer science. In a wide variety of computer science fields, you may want to include source code examples in your lecture script or slides. The standard way is to either write some pseudo-code or program-like snippets. Usually these neither compile nor are they maintained well and they are, hence, often riddled with mistakes. That is not nice.

What we want is to have snippets of "real" code. Code that we can compile, unit test, and run. Now such code naturally would not be sprinkled into our LaTeX teaching material sources. That would be a mess to organize and maintain.

A natural location for source code examples is a separate `git` repository. Maybe on GitHub, maybe somewhere else. If I wanted to do a lecture on, say, optimization algorithms, I would like to have the optimization algorithms implemented in an useful fashion. I would put them into a repository where I can build and test these real codes as a complete and separate piece of work.

Then I want to use them in my lecture scripts (written in LaTeX) as well. However, I do not want to *copy* them there. I want that my lecture scripts directly reference the `git` repository with the real code. I want them to "include" the examples from there. If I change the code in the `git` repository and then recompile my teaching material, these changes should automatically be reflected there.

That is the use case we tackle here. We offer a solution to the question

> *"How can we include snippets of code from a separate, complex code basis (located in a* `git` *repository) into our LaTeX documents?"*

Additionally, sometimes we want to execute the code from that repository and capture the standard output. This output could then be displayed as listing next to the code. This package also provides this functionality.

Maybe we either have local programs or programs in a `git` repository that create complex figures or graphics. Our package also offers us the ability execute such programs, tell them where to store their output, and then allowing us to include this output into our LaTeX documents, say, via `\includegraphics`.

Our package furthermore *caches* such outputs. If you refer to multiple files from one repository, this repository will be cloned only once. If you need to perform several LaTeX passes, say, because you have a bibliography and a glossary and whatnot, then you need to run `texgit` after *each* pass. However, `texgit` will only clone the repositories and execute the programs once, during its first pass.

All results are cached, usually in a directory called `__git__` in your document's direcotry. Unless you delete this directory, the cache will persist, even if you change your LaTeX document and perform an entirely new compilation.

1.2 Provided Functionality

Our package offers a combination of

- a LaTeX package – this package here – with its sources at `https://github.com/thomasWeise/texgit_tex`, and
- a Python program, published at `https://pypi.org/projects/texgit` with sources available at `https://github.com/thomasWeise/texgit_py`.

This LaTeX package provides the command `\gitLoad` that can load a specific file from a specific `git` repository and, optionally, pipe the file contents through a program for post-processing.

It also provides the command `\gitExec`, which can execute either a local program or a program loaded from a git repository and fetch its output.

The additional command `\gitArg` allows you to create files whose paths are passed as arguments to programs, which may be useful to create graphics and other non-textual output.

All three types of requests are stored in the `aux` file during the first `pdflatex` pass, then resolved by the Python program, and their results become available in the second `pdflatex` pass via the commands `\gitFile` and `\gitUrl`.

2 Usage

Using the package requires the following steps:

1. Obviously, both the LaTeX package *and* its Python companion package must be installed (see Section 2.1).
2. In your document, you need to load the package in the preamble (see Section 2.2).
3. Then you can make `git` queries, e.g., via `\gitLoad{id}...` or `\gitExec{id}...` (see Section 2.3).
4. At this stage, `\gitFile{id}` and `\gitUrl{id}` will hold dummy results, meaning that you can still use these commands but they will not yet provide useful data.
5. The Python post-processor package can carry them out after the first `pdflatex` run.

6. In the next `pdflatex` run, `\gitFile{id}` and `\gitUrl{id}` are defined appropriately, see Section 2.6.

If your LaTeX document is called `article.tex`, then you have at least the following workflow:

```
pdflatex article
python3 -m texgit.run article
pdflatex article
```

All files are cached, usually in a directory called `__git__` in your document's direcotry. Unless you delete this directory, the cache will persist, even if you change your LaTeX document and perform an entirely new compilation.

Comprehensive examples are provided in Section 4.

2.1 Installation

2.1.1 LaTeX Package

First, make sure that you have the `texgit.sty` either installed or inside your document's directory. For this, there are several options:

1. You can just download the file from `https://thomasweise.github.io/texgit_tex/texgit.sty` directly. You can now copy it into the folder of your document.

2. You can download `texgit.dtx` and `texgit.ins` from `https://thomasweise.github.io/texgit_tex/texgit.dtx` and `https://thomasweise.github.io/texgit_tex/texgit.ins`. You can then execute

   ```
   pdflatex texgit.ins
   ```

   and you should get the style file `texgit.sty`. You can now copy it into the folder of your document.

3. Or you can download the `texgit.tds.zip` file from `https://thomasweise.github.io/texgit_tex/texgit.tds.zip` and unpack it into your TeX tree. If you know what that is.

2.1.2 Python Package

The Python package is available at `https://github.com/thomasWeise/texgit_py`, `https://thomasweise.github.io/texgit_py`, and `https://pypi.org/project/texgit`. You can most easily install it from PyPI by doing

```
pip install texgit
```

2.1.3 git Executable

Make sure that the `git` executable is available in the `PATH`. On Ubuntu Linux, you could install it via `sudo apt-get install git`, for example. `git` is needed because the primary use case of our package is to clone `git` repositories and include the files from these repositories (or to execute them and to include their output) into LaTeX documents.

2.2 Loading the Package

Load this package using

```
\usepackage{texgit}
```

The package has no options or parameters.

Notice: If you load this package, then you ***must*** run the Python companion package inbetween `pdflatex` runs. Otherwise, there the second `pdflatex` run will abort with an error.

Loading the package will automatically load the package `filecontents` as well, see Section 8.

2.3 Querying a File from a git Repository

To query a file stored at path `path` inside from a `git` repository available under URL `repositoryURL`, you would specify the command

```
\gitLoad{id}{repositoryURL}{path}{postproc}
```

After this command is executed, a local path to the file becomes available in the command `\gitFile{id}`. The full URL to the file in the `git` repository, including the current commit id, becomes available in the fully-expandable command `\gitUrl{id}`.

Notice that you must choose unique values of `id` for every `\gitLoad`, `\gitExec`, and `\gitArg` invocation. You can invoke `\gitLoad` any number of times.

The fourth parameter, `postproc`, which we may often leave empty, can specify an optional post-processing commend. If it is not left empty, this command will be executed in the shell. The contents of the file loaded from the `git` repository will be piped to the `stdin` of the command. The `stdout` of the command will be piped to a file and `\gitFile{id}` will then contain the path to this file instead. For example, under Linux, you could use the `head` command to return only the first 5 lines of a file as follows:

```
\gitLoad{id}{repositoryURL}{path}{head -n 5}
```

2.4 Executing a Command (optionally inside a git Repository

Sometimes, we want to execute a program and fetch its standard output.

```
\gitExec{id}{repositoryURL}{path}{theCommand}
```

The most common use case of our package is that you want to execute a program which is part of a `git` repository. In this case, you would put the URL of the repository in `repositoryURL` and the relative path to the directory inside the repository in which the command should be invoked as `path`. If you want to invoke the command in the root folder of the repository, put `.` as `path`. The `theCommand` then holds the command line to be executed. *Notice:* You can also leave *both* `repositoryURL` and `path` blank. In this case, the command is executed in the current folder. (The use case for this is to fetch the output of stuff like `python3 --version`.) Anyway, after this command is executed, a local path to the file with the captured standard output becomes available in the command `\gitFile{id}`.

Notice that you must choose unique values of `id` for every `\gitLoad`, `\gitExec`, and `\gitArg` invocation. You can invoke `\gitExec` any number of times.

2.5 Creating a File to be used as Argument

Sometimes, we want to execute a program which requires a destination file. Let's say a program that creates a PDF figure. For this, we need to create an argument file on the fly.

```
\gitArg{id1}{prefix}{suffix}
\gitExec{id2}{repositoryURL}{path}{theCommand (?id1?)}
```

For this purpose, use the `\gitArg` command. This command takes a unique identifier `id1`, a prefix, and a suffix as parameters. It will allocate a unique file path. This path can then be passed to a command to later invocation of `\gitExec` as parameter `(?id1?)` somewhere in its argument list. `(?id1?)` is resolved to the automatically generated file name before the actual command is invoked. After that, you can access the path via `\gitFile{id1}`.

2.6 Executing the Python Package

During the first `pdflatex` run, `\gitFile{id}` points to an empty dummy file (`\jobname.texgit.dummy`) and `\gitUrl{id}` points to `http://example.com`. Both commands will only expand to useful information if the Python package `texgit` is applied to the project's `aux` file. This works very similar to BibTeX. If the name of your TeX file is `myfile.tex`, then you would execute

```
python3 -m texgit.run myfile
```

More specifically, the Python package processes the `aux` files, so for a specific `aux` file `myfile.aux`, you could also do:

```
python3 -m texgit.run myfile.aux
```

After this, in the next pass of `pdflatex`, `\gitFile{id}` and `\gitUrl{id}` will contain the right paths and URLs.

2.7 A Note on Python Virtual Environments

The following only applies if you have installed this package inside a virtual environment. It also only applies in conjunction with version 0.8.17 or newer of the `texgit` Python package.

If you are running this package inside a virtual environment, it is important that you create this environment using the `--copies` setting and *not* using the (default) `--symlinks` parameter. In other words, you should have created the virtual environment as follows, where `venvDir` is the directory inside of which the virtual environment is created.

```
python3 -m venv --copies venvDir
```

If you create the environment like this (and activated), then our package will automatically pick it up and use its Python interpreter for any invocation of `python3` or `python3.x` (where `x` is the minor version of the interpreter). If you use the `--symlinks` parameter to create the environment, then invocations of the Python interpreter from our package may instead result in the system's Python interpreter.

3 Provided Macros

Here we discuss the macros that can directly be accessed by the user to make use of the functionality of the `texgit` package. The implementation of these macros is given in Section 8 and several examples can be found in Section 4.

3.1 gitLoad

`\gitLoad` The macro `\gitLoad{`⟨*id*⟩`}{`⟨*repositoryURL*⟩`}{`⟨*path*⟩`}{`⟨*postProcessing*⟩`}` provides a local path to a file from a `git` repository.

`{`⟨*id*⟩`}` is the request ID chosen by the user. It must be unique over all requests made to `texgit`. Imagine it something like a label.

`{`⟨*repositoryURL*⟩`}` is the URL of the `git` repository. It could, e.g., be `https://github.com/thomasWeise/texgit_tex` or `ssh://git@github.com/thomasWeise/texgit_tex` or any other valid repository URL.

`{`⟨*path*⟩`}` is then the path to the file within the repository. This could be, for example, `latex/texgit.dtx`.

`{`⟨*postProcessing*⟩`}` Can either be empty, in which case the repository is downloaded and the the local path to the file is returned. It can also be shell command, e.g., `head -n 5`. In this case, the contents of the file are piped to `stdin` of the command and the text written to the `stdout` by the command is stored in a file whose path is returned.

You can access two results of this command via the following two commands:

`\gitFile{id}` returns the path to the file that was loaded and/or post-processed.

`\gitUrl{id}` returns the full URL to the file in the `git` repository online. This command works for GitHub, but it may not provide the correct URL for other repository types.

3.2 gitExec

`\gitExec` The macro `\gitExec{`⟨*id*⟩`}{`⟨*repositoryURL*⟩`}{`⟨*path*⟩`}{`⟨*theCommand*⟩`}` provides a local path to a file containing the captured standard output of a command (that may have been executed inside a directory inside a `git` repository).

`{`⟨*id*⟩`}` is the request ID chosen by the user. It must be unique over all requests made to `texgit`. Imagine it something like a label.

{⟨*repositoryURL*⟩} is the URL of the `git` repository. It could, e.g., be `https://github.com/thomasWeise/texgit_tex` or `ssh://git@github.com/thomasWeise/texgit_tex` or any other valid repository URL. You can also leave this parameter empty if no `git` repository should be used.

{⟨*path*⟩} is the path to a directory within the repository. This could be, for example, `latex` or `.`. If `path` is provided, then this will be the working directory where the command is executed. If you want to execute a command in the root directory of a `git` repository, you can put `.` here.

{⟨*theCommand*⟩} This is the command which should be executed. If `repositoryURL` and `path` are provided, then the repository will be downloaded and `path` will be resolved relative to the repository root directory. `theCommand` will then be executed in this directory. If neither `repositoryURL` nor `path` are provided, `theCommand` is executed in the current directory. Either way, its `stdout` is captured in a file whose path is returned.

After invoking this command its result can be obtained via the following command:

`\gitFile{id}` returns the path to the file in which the standard output is stored.

3.3 gitArg

`\gitArg` The macro `\gitArg{`⟨*id*⟩`}{`⟨*prefix*⟩`}{`⟨*suffix*⟩`}` allows you to create a file whose name has a given `prefix` and `suffix`. This file can then be used in the argument list of the command invoked by `\gitExec` by writing `(?id?)`. Before that command is executed, `(?id?)` is resolved to the actual file name. This allows you to use commands that generate more structured output, say graphics.

{⟨*id*⟩} is the request ID chosen by the user. It must be unique over all requests made to `texgit`. Imagine it something like a label.

{⟨*prefix*⟩} is a prefix for the file name to be generated. It can be empty.

{⟨*suffix*⟩} is the suffix for the file name to be generated. You can leave it empty.

3.4 gitFile

`\gitFile` The macro `\gitFile{`⟨*id*⟩`}` returns the path to the file with the contents of the `\gitLoad{id}...`, `\gitExec{id}....`, or `\gitArg{id}...` request using ID `id`. During the first `pdflatex` pass, this will be the path to a dummy file. After the Python package has been applied to the `aux` file, then `\gitFile{id}` will point to the proper file during the next `pdflatex` pass.

3.5 gitUrl

`\gitUrl` The macro `\gitUrl{`⟨*id*⟩`}` returns the URL from which the file corresponding to the `\gitLoad{id}...` request was downloaded. This command is designed to work with GitHub. It will be the repository URL combine with the path of the file inside the repository and the commit has code. The Url thus points to the exact same version of the file that was downloaded (and optionally post-processed).

3.6 gitName

`\gitName` The macro `\gitName{`⟨*id*⟩`}` returns the name of the file corresponding to the `\gitLoad{id}...` request. This corresponds to the `basename` on unixoid file systems. In other words, only the local name of the file, without any other path components. This command makes sense only with files that are directly downloaded from `git` or filtered versions thereof, or *maybe* argument files.

3.7 gitNameEsc

`\gitNameEsc` The macro `\gitNameEsc{`⟨*id*⟩`}` returns the *escaped* name of the file corresponding to the `\gitLoad{id}...` request. This corresponds to the `basename` on unixoid file systems, as returned by `\gitName`, but with `_`, `$`, and   escaped to `\_`, `\$`, and ˜ respectively. This allows you to use the name of a file in plain TEX, maybe like `\texttt{\gitNameEsc{myId}}`. This command makes sense only with files that are directly downloaded from `git` or filtered versions thereof, or *maybe* argument files.

3.8 gitIf

`\gitIf` The macro `\gitIf{`⟨*id*⟩`}`ifDoneifNotDone executes the code provided as `ifDone` if the request with ID `id` has already been executed and completed by the `texgit` Python package. Otherwise, it will execute the code provided as `ifNotDone`. As stated before, during the first `pdflatex` pass, all results of `\gitFile` and `\gitUrl` are dummies, either empty files or dummy URLs. However, some LATEX commands cannot deal with that. For example, say that we execute a program to generate a PDF graphic and store it in an argument file. During the first `pdflatex` run, this file is empty. If we try to `\includegraphics` it, this will crash. So we would place the `\includegraphics` into a `\gitIf` block and only execute it once the request is completed. After the first `pdflatex` run, we would execute the Python package `texgit`. This package will complete the request and the argument file will then be a proper graphic. In the second `pdflatex` pass, the `\includegraphics` code could be executed. An example for this is given in Section 4.8.

4 Examples

Here we provide a set of examples for the use of the package. Each example demonstrates another facet of the package and, at the same time, serves as test case. The first example given in Section 4.1 is a Minimal Working Example, i.e., just provides the barest bones. It shows you how to import a single file from a `git` repository. The second example in Section 4.2 shows you how to import multiple different files from different repositories (which equates to just using the same command multiple times) and how to use post-processors. The third example in Section 4.3 shows how to create beautiful (to my standards) listings by including code from a `git` repository, post-processing it, and loading it as a `listing`. Finally, the fourth example in Section 4.4 shows that you can also define macros for your favorite repository and post-processors to have a more convenient way to import files from `git`.

Listing 1: A minimal working example for using the `texgit` package, rendered as Figure 1. The contents of `dummy.tex` are shown in Listing 2.

```
1  \documentclass{article}%
2  \usepackage{texgit}% use our package
3  \usepackage{verbatim}% for loading data
4  \begin{document}%
5  A\gitLoad{R1}{https://github.com/thomasWeise/texgit_tex}{examples/dummy.tex
       }{}BC\input{\gitFile{R1}}D%
6  \end{document}%
```

Listing 2: The contents of the file `dummy.tex` included from `git` in Listing 1.

```
This is a dummy text file.
It just contains this text, nothing else.
It can directly be included into \LaTeX.
Since we directly \verb=\input= it, it can also contain macros.
And math: $1+2=\sqrt{9}$.
```

4.1 Minimal Working Example

This minimal working example shows how to load a file from a `git` repository and directly `\input` its contents. The result can be seen in Figure 2.

As you can see in Listing 1, we first load the package `texgit` in line 2. Inside the document, we define a `git` request via the `\gitLoad` command. This command takes the ID of the request as first parameter. Here, we chose `R1`. Its second parameter is the URL of a `git` repository. In this case, this is `https://github.com/thomasWeise/texgit_tex`, which happens to be the URL where you can find the repository of this package on GitHub. The third parameter is a path to the file in this repository relative to the repository root. In this case, this is the path to the file `examples/dummy.tex`, whose contents you can find in Listing 2.

The fourth parameter shall be ignored for now.

After defining the request, we can now use two commands, `\gitFile{R1}` and `\gitUrl{R1}`. In this Minimal Working Example, we shall only consider the first one. This command expands to a local path of a file with the contents downloaded from the `git` repository.

Well, during the first LaTeX or `pdflatex` run, it just points to a dummy file with the name `\jobname.texgit.dummy`, where `\jobname` evaluates to the name of the main LaTeX document, say `article` for `article.tex`. At that point, the dummy file's content is a single space character followed by a newline.

After the first `pdflatex` pass, you can apply the Python processor (see Section 2.1.2) as follows:

```
python3 -m texgit.run jobname
```

Where `jobname` shall be replaced with the main file name, again `article` for `article.tex`, for instance.

This command then downloads the file from `git` and puts it into a path that can locally accessed by LaTeX. Usually, it will create a folder `__git__` in your

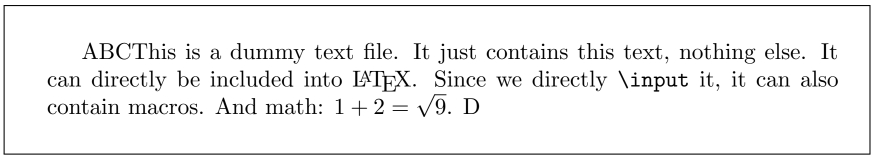

ABCThis is a dummy text file. It just contains this text, nothing else. It can directly be included into LaTeX. Since we directly `\input` it, it can also contain macros. And math: $1 + 2 = \sqrt{9}$. D

Figure 1: The rendered result of Listing 1 (with trimmed page margins and bottom).

project's directory and place the file there.

Anyway, during the second LaTeX or `pdflatex` pass, `\gitFile{R1}` points to a valid file path with actual contents. By doing `\input{\gitFile{R1}}`, we here include this file (remember, its contents are given in Listing 2) as if it was part of our normal LaTeX project. The result of this pass is shown in Figure 1.

If this example was stored as `example_1.tex`, then it could be built via

```
pdflatex example_1
python3 -m texgit.run example_1
pdflatex example_1
```

If we look back at the Listing 1 of our main file, you will notice the four blue marks **A**, **B**, **C**, and **D**. These are just normal letters, colored and emphasized for your convenience. I put them there so that you can see where the action takes place. `\gitLoad` produces no output, so "ABC" come out next to each other. `\input{\gitFile}` between **C** and **D** loads and directly includes the example file, so this is where its content appear.

One small interesting thing is that, since we directly `\input` the file, its contents are interpreted as LaTeX code. This means that you could construct a document by inputting files from different `git` repositories.

However, this is not the envisioned use case. The envisioned use case is to include source codes and snippets from source codes as listings. We will show how this could be done in the next example.

Side note: Our Python companion package `texgit` downloads the `git` repositories into a folder called `__git__` by default. If you do not delete the folder, the same repository will not be downloaded again but the downloaded copy will be used. This significantly increases speed and reduces bandwidth when applying the `texgit` command several times.

4.2 The Second Example: Multiple Files and Post-Processing

In Listing 3 we, use `texgit` to download and present two different files from two different GitHub repositories. We also show how post-processing can work, once using the aforementioned simple `head -n 5` command available in the Linux shell and also by using the Python code formatting tool offered by the `texgit` Python package. The result can be seen in Figure 2.

Listing 3: An example using the texgit package, rendered as Figure 2.

```latex
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{verbatim}% for loading a file verbatim
\usepackage[colorlinks]{hyperref}% for printing the URL
\begin{document}%
%
\section{First File}%
First, we load a file from the GitHub repository
“\url{https://github.com/thomasWeise/texgit\_py}”, where the Python complement
package of our \LaTeX\ package is located. We will then include this file verbatim
without any modification.

\gitLoad{R2}{%
https://github.com/thomasWeise/pycommons}{%
pycommons/io/console.py}{}%
% now, \gitFile and \gitUrl are defined and can be used.
\verbatiminput{\gitFile{R2}}% print the contents of the file
The file \gitNameEsc{R2} was loaded from URL \url{\gitUrl{R2}}.% print url
%
\clearpage\section{Second File}%
We load the same file again, but this time retain only the first five lines.
We do this by specifying that the file contents should be piped through
“\verb=head -n 5=” before inclusion.
\gitLoad{R3}{https://github.com/thomasWeise/pycommons}{%
pycommons/io/console.py}{head -n 5}%
% now, \gitFile and \gitUrl are defined and can be used.
\verbatiminput{\gitFile{R3}}% print the contents of the file
The file \gitNameEsc{R3} was loaded from URL \url{\gitUrl{R3}}.% print url
%
\clearpage\section{Third File}%
We now load a file from the “\url{https://github.com/thomasWeise/moptipy}”
GitHub repository. The contents of this file will be piped through the Python
code formatter, which retains only a snippet of the code and removes type
hints and comments, while keeping the doc strings. (It doesn't really matter
what it does, it is just postprocessing.)
\gitLoad{R4}{%
https://github.com/thomasWeise/moptipy}{moptipy/api/encoding.py}{%
python3 -m texgit.formatters.python --labels book --args doc}% post-processor
% now, \gitFile and \gitUrl are defined and can be used.
\verbatiminput{\gitFile{R4}}% print the contents of the file
The file \gitNameEsc{R4} was loaded from URL \url{\gitUrl{R4}}.% print url
%
\end{document}%
```

The file `example_2.tex` shown in Listing 3 begins by loading our `texgit` package as well as package `verbatim`, which is later used to display the included files. The document creates three sections, each of which is used to display one imported file.

The first section loads one Python source file from the Python package pycommons. The sources of this package are available in the GitHub repository `https://github.com/thomasWeise/pycommons`. We download the file `pycommons/io/console.py`, which is just a small utility for printing log strings to the output together with a time mark. The full request, with the ID `R2`, contains these two components.

Issuing this request will set the command `\gitFile{R2}` to the local file containing the downloaded contents of `pycommons/io/console.py` from the repository `https://github.com/thomasWeise/pycommons`. The command `\gitUrl{R2}` will expand to the URL pointing to the downloaded *version* of the file in the original repository. This command, at the present time, is only really valid for GitHub. It builds a URL relative to the original repository based on the commit ID that was valid when the file was downloaded from the repository. Therefore, the URL then points to the *exact same* contents that were put into the file. Anyway, the file contents and the generated URL are displayed in Figure 2a.

The second section of the example document queries the same file again. However, this time, the fourth parameter of `\gitLoad` is specified. If the fourth parameter is left blank, the downloaded file will be provided as-is. However, especially if we would like to include some snippets of a more complex source file, we sometimes do not want to have the complete original contents. In this case, we can specify a post-processing command as third parameter. This command will be executed in the shell The contents of the downloaded file will then be piped into the `stdin` of the command and everything that the command writes to its `stdout` will be collected in a file. We use ID `R2` for this request. `\gitFile{R2}` then returns the path to that file.

Since you can provide arbitrary commands as post-processors, this allows you to do, well, arbitrary post-processing. This could include re-formatting of code or selecting only specific lines from the file. The command can have arguments, separated by spaces, allowing you to pass information such as line indices or other instructions to your post-processing command.

In the example, we use the standard Linux command `head -n 5`, which writes the first five lines that were written to its `stdin` to its `stdout`.

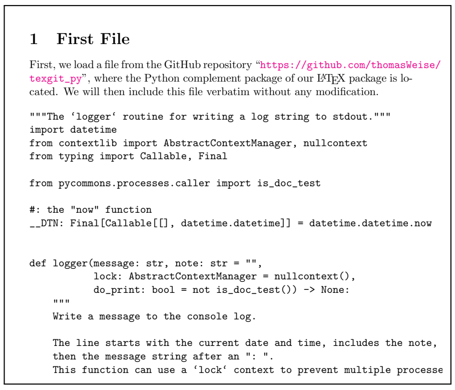
The resulting output in Figure 2b looks thus similar to Figure 2a, but only imports ths first five lines from the downloaded file.

If this example was stored as `example_2.tex`, then it could be built via

```
pdflatex example_2
python3 -m texgit.run example_2
pdflatex example_2
```

Side note: Such post-processing steps are cached by the Python companion package `texgit` in the `__git__` folder as well.

Finally, in the third section, of Listing 3, we import a file from the sources of our Python package for metaheuristic optimization (moptipy). The sources of this package are located on GitHub at `https://github.com/thomasWeise/moptipy`. We download the file `moptipy/api/encoding.py`, which offers a convenient API

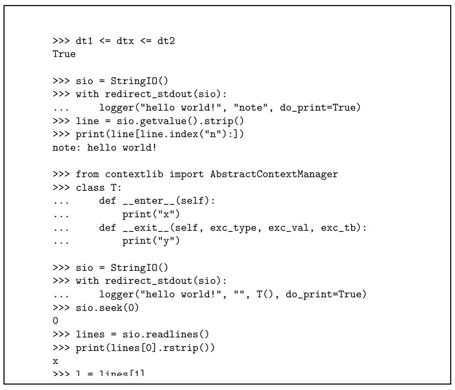


1 First File

First, we load a file from the GitHub repository "https://github.com/thomasWeise/texgit_py", where the Python complement package of our LaTeX package is located. We will then include this file verbatim without any modification.

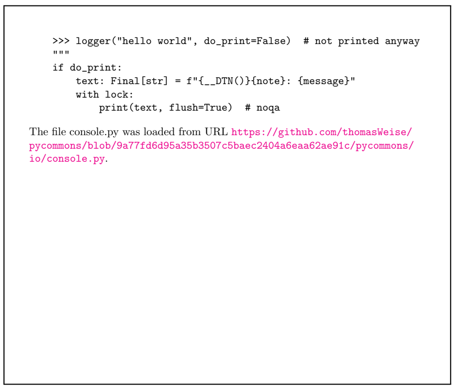
```
"""The ‘logger‘ routine for writing a log string to stdout."""
import datetime
from contextlib import AbstractContextManager, nullcontext
from typing import Callable, Final

from pycommons.processes.caller import is_doc_test

#: the "now" function
__DTN: Final[Callable[[], datetime.datetime]] = datetime.datetime.now


def logger(message: str, note: str = "",
           lock: AbstractContextManager = nullcontext(),
           do_print: bool = not is_doc_test()) -> None:
    """
    Write a message to the console log.

    The line starts with the current date and time, includes the note,
    then the message string after an ": ".
    This function can use a ‘lock‘ context to prevent multiple processe
```

(a) Page 1 of the pdf compiled from Listing 3.



```
    >>> dt1 <= dtx <= dt2
    True

    >>> sio = StringIO()
    >>> with redirect_stdout(sio):
    ...     logger("hello world!", "note", do_print=True)
    >>> line = sio.getvalue().strip()
    >>> print(line[line.index("n"):])
    note: hello world!

    >>> from contextlib import AbstractContextManager
    >>> class T:
    ...     def __enter__(self):
    ...         print("x")
    ...     def __exit__(self, exc_type, exc_val, exc_tb):
    ...         print("y")

    >>> sio = StringIO()
    >>> with redirect_stdout(sio):
    ...     logger("hello world!", "", T(), do_print=True)
    >>> sio.seek(0)
    0
    >>> lines = sio.readlines()
    >>> print(lines[0].rstrip())
    x
    >>> l = lines[1]
```

(b) Page 2 of the pdf compiled from Listing 3.



```
    >>> logger("hello world", do_print=False)  # not printed anyway
    """
    if do_print:
        text: Final[str] = f"{__DTN()}{note}: {message}"
        with lock:
            print(text, flush=True)  # noqa
```

The file console.py was loaded from URL https://github.com/thomasWeise/pycommons/blob/9a77fd6d95a35b3507c5baec2404a6eaa62ae91c/pycommons/io/console.py.

(c) Page 3 of the pdf compiled from Listing 3.

Figure 2: The rendered result of Listing 3 (with trimmed page margins and bottoms).

for implementing an *encoding* which translates from the search to the solution space (but that would lead too far here). Either way, this is a file that has lots of content. So we want to select certain contents while ignoring other. We also remove all Python type hints and all comments from the source and then reformat it.

Luckily, our `texgit` Python package also offers a Python code formatter, namely the executable module texgit.formatters.python. This module takes a set of parameters such as limiting `labels` that denote the start and end of code snippets (in this case, the label "`book`") to include as well `args` telling the system which part of the "omittable" code to preserve (in this case, preserve `docstrings` and delete everything else that is non-essential). If you are interested in such post-processing, feel invited to check out the documentation of the Python companion package at `https://thomasweise.github.io/texgit_py`. Either way, the file is downloaded, piped through this post-processor, and the result is included as shown in Figure 2c.

Listing 4: An example using the `listings` package, rendered as Figure 3.

```
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{listings}% importing external code
\lstset{language=Python,basicstyle=\small\ttfamily,%
keywordstyle=\ttfamily\color{teal!90!black}\bfseries,%
identifierstyle=,commentstyle=\color{gray}\footnotesize,%
stringstyle=\ttfamily\color{red!90!black},%
numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
backgroundcolor=\color{black!10!yellow!5!white}}%
\begin{document}%
%
Behold the beautiful \autoref{l}.%
%
\gitLoad{R5}{https://github.com/thomasWeise/moptipy}{%
moptipy/algorithms/so/rls.py}{%
python3 -m texgit.formatters.python --labels book}% post-processor
%
\lstinputlisting[label=l,caption={%
The RLS Algorithm. (\href{\gitUrl{R5}}{src})}]{\gitFile{R5}}%
%
\end{document}%
```

4.3 The Third Example: Using the `listings` Package

As third example, let us show the interaction with the package `listings`. This is not much different from using the package `verbatim` in the second example above. I just wanted to show you how it looks like. Also, I wanted to show the intended use of `\gitUrl`: You can use it to put some small "(src)" link in the listing's caption. This way, you can create teaching material where every listing is linked to the correct version of source code online without splattering long URLs into your text. Anyway. The source code of the third example is given in Listing 4 and the compiled result as Figure 3.

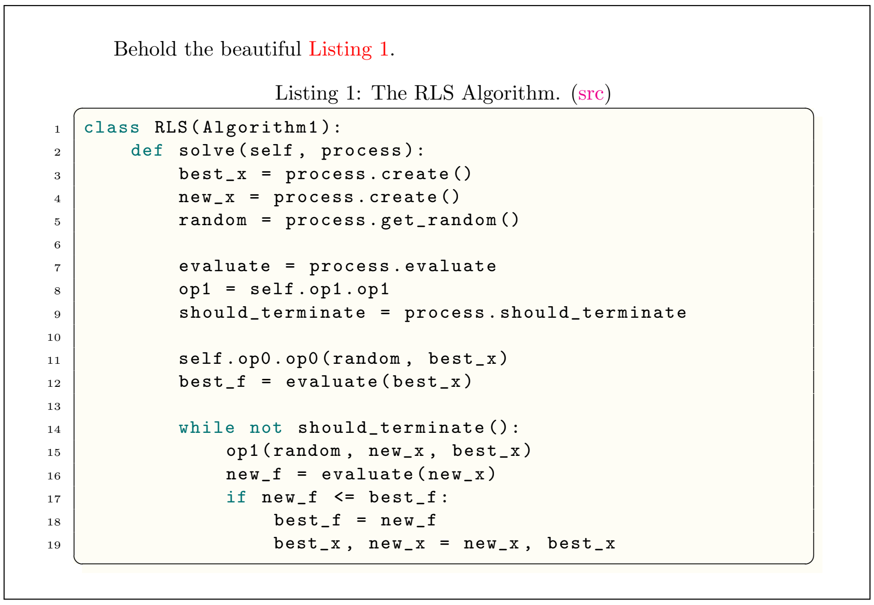
If this example was stored as `example_3.tex`, then it could be built via

```
pdflatex example_3
python3 -m texgit.run example_3
pdflatex example_3
```

Side note: If you actually check the source code of the RLS algorithm, which is linked to by the "(src)" in the caption of the example and that is displayed in the example, you will find that it actually uses Python type hints. It also has a comprehensive doc-string and is commented well. In source code of a real project, we do want this. In a listing in a book, we do not. The post-processor command

```
python3 -m texgit.formatters.python --labels book
```

only keeps the code between the labels "`# start book`" and "`# end book`." It also removes all non-essential stuff such as type hints, comments, and the doc-string.


Behold the beautiful Listing 1.

Listing 1: The RLS Algorithm. (src)

```
class RLS(Algorithm1):
    def solve(self, process):
        best_x = process.create()
        new_x = process.create()
        random = process.get_random()

        evaluate = process.evaluate
        op1 = self.op1.op1
        should_terminate = process.should_terminate

        self.op0.op0(random, best_x)
        best_f = evaluate(best_x)

        while not should_terminate():
            op1(random, new_x, best_x)
            new_f = evaluate(new_x)
            if new_f <= best_f:
                best_f = new_f
                best_x, new_x = new_x, best_x
```

Figure 3: The rendered result of Listing 4 (with trimmed page margins and bottom).

Then it re-formats the code to save space. Again, check out the documentation of our `texgit` Python companion package at https://thomasweise.github.io/texgit_py. This is the main intended use case of our package: Be able to have nicely documented "real" code and to use parts of it in teaching materials.

4.4 The Fourth Example: Using Git Commands in Macros

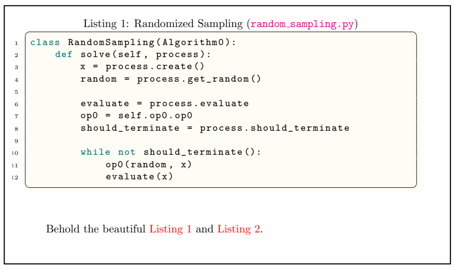
The goal of the fourth example is to show that we can also put the commands from our `texgit` package into LaTeX macros. We define a new command `\moptipySrc` with three parameters. moptipy is a Python package that implements lots of metaheuristic algorithms. We could want to load several files from such a repository https://github.com/thomasWeise/moptipy and post-process and display them all in the same way. Then, it would be annoying to always do `\gitLoad`, `\lstinputlisting`, and spell out the post-processor each time. So we put all of this into a single command whose first argument is the label to put for the listing, whose second command is the caption to use, and whose third command is the path relative to the folder "moptipy" in the `git` repository. We also use this first parameter, the label, as ID for our `texgit` commands. In Listing 5, we can then simply call `\moptipySrc` and it will do the whole process of loading a file from the right repository, post-processing it, putting a floating listing, and even putting a small "(src)" into the caption of the listing. The results are shown in Figure 4 and can be obtained via

```
pdflatex example_4
python3 -m texgit.run example_4
pdflatex example_4
```

(if the example code from Listing 5 was stored in a file called `example_4.tex`, Listing 5: An example using commands from the texgit package in macros, rendered as Figure 4.

```latex
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{listings}% importing external code
\lstset{language=Python,basicstyle=\small\ttfamily,%
keywordstyle=\ttfamily\color{teal!90!black}\bfseries,%
identifierstyle=,commentstyle=\color{gray}\footnotesize,%
stringstyle=\ttfamily\color{red!90!black},%
numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
backgroundcolor=\color{black!10!yellow!5!white}}%
%
\gdef\moptipySrc#1#2#3{%
\gitLoad{#1}{https://github.com/thomasWeise/moptipy}{moptipy/#3}{%
python3 -m texgit.formatters.python --labels book}%
\lstinputlisting[float,label={#1},caption={%
#2~(\href{\gitUrl{#1}}{\texttt{\gitNameEsc{#1}}})}]{\gitFile{#1}}}
%
\begin{document}%
%
Behold the beautiful \autoref{R6} and \autoref{R7}.%
%
\moptipySrc{R6}{Randomized Sampling}{algorithms/random_sampling.py}
\moptipySrc{R7}{Randomized Local Search}{algorithms/so/rls.py}
%
\end{document}%
```

that is.)

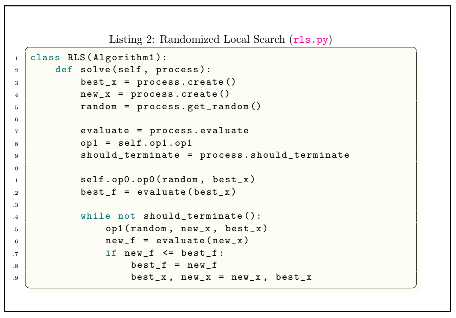


Listing 1: Randomized Sampling (random_sampling.py)

```
class RandomSampling(Algorithm0):
    def solve(self, process):
        x = process.create()
        random = process.get_random()

        evaluate = process.evaluate
        op0 = self.op0.op0
        should_terminate = process.should_terminate

        while not should_terminate():
            op0(random, x)
            evaluate(x)
```

Behold the beautiful Listing 1 and Listing 2.

(a) Page 1 of the pdf compiled from Listing 5.



Listing 2: Randomized Local Search (rls.py)

```
class RLS(Algorithm1):
    def solve(self, process):
        best_x = process.create()
        new_x = process.create()
        random = process.get_random()

        evaluate = process.evaluate
        op1 = self.op1.op1
        should_terminate = process.should_terminate

        self.op0.op0(random, best_x)
        best_f = evaluate(best_x)

        while not should_terminate():
            op1(random, new_x, best_x)
            new_f = evaluate(new_x)
            if new_f <= best_f:
                best_f = new_f
                best_x, new_x = new_x, best_x
```

(b) Page 2 of the pdf compiled from Listing 5.

Figure 4: The rendered result of Listing 5 (with trimmed page margins and bottoms).

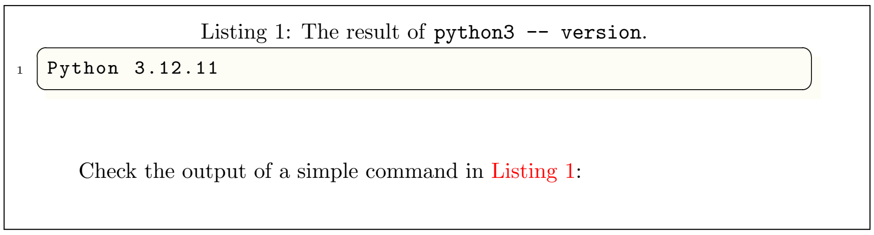
4.5 The Fifth Example: Capturing the Output of a Program

The goal of the fifth example is to show that we can capture the output of a program. In Listing 6, we just invoke `python3 --version` and capture the output in a file. We then load this file as listing. The results are shown in Figure 5 and can be obtained via

```
pdflatex example_5
python3 -m texgit.run example_5
pdflatex example_5
```

Listing 6: An example of capturing the output of a program, rendered as Figure 5.

```
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{listings}% importing external code
\lstset{language={},basicstyle=\small\ttfamily,%
numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
backgroundcolor=\color{black!10!yellow!5!white}}%
%
\begin{document}%
%
Check the output of a simple command in \autoref{lst:out}:
%
\gitExec{R8}{}{}{python3 ——version}%
\lstinputlisting[float,label={lst:out},caption={%
The result of \texttt{python3 {–}{–} version}.}]{\gitFile{R8}}
%
\end{document}%
```


Listing 1: The result of `python3 -- version`.

```
Python 3.12.11
```

Check the output of a simple command in Listing 1:

Figure 5: The rendered result of Listing 6 (with trimmed page margins and bottoms).

(if the example code from Listing 6 was stored in a file called `example_5.tex`, that is.)

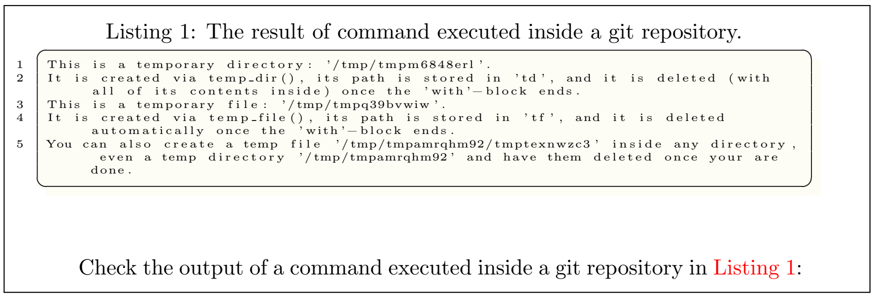
4.6 The Sixth Example: Capturing the Output of a Program Executed Inside a git Repository

The goal of the sixth example is to show that we can capture the output of a program – but this time we execute it inside a `git` repository. In Listing 7, we invoke a program which is part of the examples suite of the pycommons utility package. We capture its standard output in a file. We then load this file as listing. The results are shown in Figure 6 and can be obtained via

```
pdflatex example_6
```

Listing 7: An example of capturing the output of a program executed inside a git repository, rendered as Figure 6.

```
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{listings}% importing external code
\lstset{language={},basicstyle=\tiny,%
breaklines=true,numbers=left,numberstyle=\tiny,%
frame=shadowbox,frameround=tttt,%
backgroundcolor=\color{black!10!yellow!5!white}}%
%
\begin{document}%
%
Check the output of a command executed inside a git
repository in \autoref{lst:out}:
%
\gitExec{R9}{https://github.com/thomasWeise/pycommons}{examples}{python3
    temp.py}%
\lstinputlisting[float,label={lst:out},caption={%
The result of command executed inside a git repository.}]{\gitFile{R9}}
%
\end{document}%
```



Listing 1: The result of command executed inside a git repository.

```
This is a temporary directory: '/tmp/tmpm6848erl'.
It is created via temp_dir(), its path is stored in 'td', and it is deleted (with
     all of its contents inside) once the 'with'-block ends.
This is a temporary file: '/tmp/tmpq39bvwiw'.
It is created via temp_file(), its path is stored in 'tf', and it is deleted
     automatically once the 'with'-block ends.
You can also create a temp file '/tmp/tmpamrqhm92/tmptexnwzc3' inside any directory,
      even a temp directory '/tmp/tmpamrqhm92' and have them deleted once your are
     done.
```

Check the output of a command executed inside a git repository in Listing 1:

Figure 6: The rendered result of Listing 7 (with trimmed page margins and bottoms).

```
python3 -m texgit.run example_6
pdflatex example_6
```

(if the example code from Listing 7 was stored in a file called `example_6.tex`, that is.)

4.7 The Seventh Example: Capturing the Output of Multiple Programs Executed Inside Different git Repositories

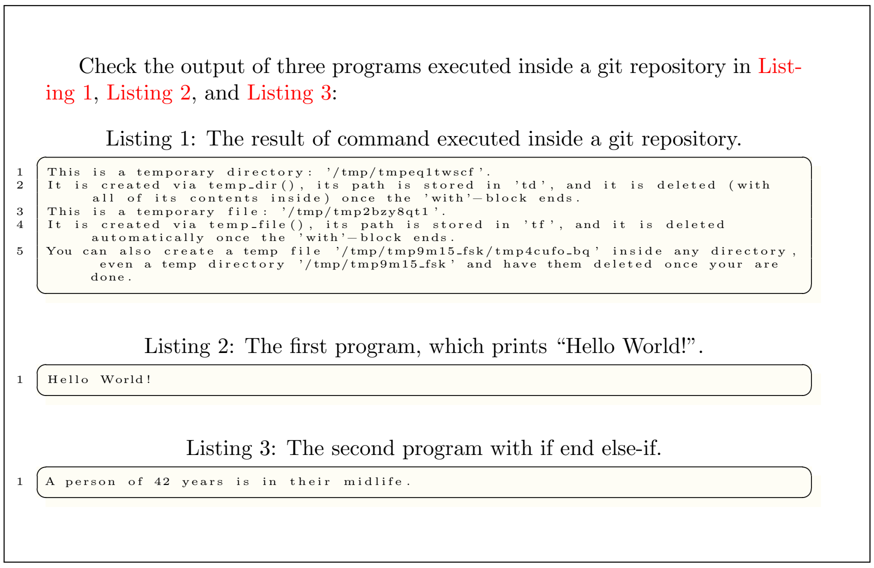
The goal of the seventh example is to show that we can capture the output of multiple programs from inside different `git` repositories. In Listing 8, we invoke the same program as in Listing 7 and of two programs which are part of the examples suite of the Programming with Python book. The examples can be found in the repository `https://github.com/thomasWeise/programmingWithPythonCode`, whereas the book can be downloaded from `https://github.com/thomasWeise/programmingWithPython`. We capture the standard output of both programs in three files. We then load these file as listings. The results are shown in Figure 7 and can be obtained via

```
pdflatex example_7
python3 -m texgit.run example_7
pdflatex example_7
```

(if the example code from Listing 8 was stored in a file called `example_7.tex`, that is.)

Listing 8: An example of capturing the output of three programs executed inside different git repositories, rendered as Figure 7.

```
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{listings}% importing external code
\lstset{language={},basicstyle=\tiny,literate={π}{{$\pi$}}1,%
numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
breaklines=true,backgroundcolor=\color{black!10!yellow!5!white}}%
%
\begin{document}%
%
Check the output of three programs executed inside a git
repository in \autoref{lst:out1}, \autoref{lst:out2}, and \autoref{lst:out3}:
%
\gitExec{R10}{https://github.com/thomasWeise/pycommons}{examples}{python3
    temp.py}%
\lstinputlisting[label={lst:out1},caption={%
The result of command executed inside a git repository.}]{\gitFile{R10}}
%
\gitExec{R11}{https://github.com/thomasWeise/programmingWithPythonCode}{%
veryFirstProject}{python3 very_first_program.py}%
\lstinputlisting[label={lst:out2},caption={%
The first program, which prints ``Hello World!''.}]{\gitFile{R11}}
%
\gitExec{R12}{https://github.com/thomasWeise/programmingWithPythonCode}{%
conditionals}{python3 if_elif_example.py}%
\lstinputlisting[label={lst:out3},caption={%
The second program with if end else-if.}]{%
\gitFile{R12}}
%
\end{document}%
```



Check the output of three programs executed inside a git repository in Listing 1, Listing 2, and Listing 3:

Listing 1: The result of command executed inside a git repository.

```
This is a temporary directory: '/tmp/tmpeq1twscf'.
It is created via temp_dir(), its path is stored in 'td', and it is deleted (with
    all of its contents inside) once the 'with'-block ends.
This is a temporary file: '/tmp/tmp2bzy8qt1'.
It is created via temp_file(), its path is stored in 'tf', and it is deleted
    automatically once the 'with'-block ends.
You can also create a temp file '/tmp/tmp9m15_fsk/tmp4cufo_bq' inside any directory,
     even a temp directory '/tmp/tmp9m15_fsk' and have them deleted once your are
    done.
```

Listing 2: The first program, which prints "Hello World!".

```
Hello World!
```

Listing 3: The second program with if end else-if.

```
A person of 42 years is in their midlife.
```

Figure 7: The rendered result of Listing 8 (with trimmed page margins and bottoms).

4.8 The Eight Example: Using Argument Files

The goal of the eighth example is to show how to use argument files. In Listing 9, we first create an argument file via `\gitArg{R13}{test}{.pdf}`. This file will have the name prefix `test` and the suffix `.pdf`. It will be available under the ID `R13`. The command `\gitExec{}{...}{...}{python3 make_pdf.py (?R13?)}` invokes the Python program `make_pdf.py` residing in a certain directory of a `git` repository (actually, our repository here). We leave the first parameter, the `id`, empty, because we are not interested in the output of this program. The agument `(?R13?)` passed to the program will be resolved to the argument file path before invoking the program. The program is illustrated in Listing 10. It generates a PDF graphic and stores it in the file that it received as argument. We can then include this graphic using `\includegraphics` and passing in `\gitFile{R13}`.

However, `\gitFile{R13}` will only be valid after we applied the Python `texgit` command *after* the first `pdflatex` pass. Therefore, during the first `pdflatex` pass, this file will be empty. `\includegraphics` would fail. Therefore, we protect this code to be only executed once the file has been filled with data. This is done by placing it into the `\gitIf{R13}{<exec if done>}{<otherwise>}` block.

The results are shown in Figure 8 and can be obtained via

```
pdflatex example_8
python3 -m texgit.run example_8
pdflatex example_8
```

(if the example code from Listing 9 was stored in a file called `example_8.tex`, that is.)

Listing 9: An example of running a script residing in git and passing an automatically generated file path to it as input, rendered as Figure 8.

```
\documentclass{article}%
\usepackage{texgit}% use our package
\usepackage{xcolor}% to be able to use colors
\usepackage[colorlinks]{hyperref}% for printing the URL
\usepackage{graphicx}% loading graphics
%
\begin{document}%
%
\gitArg{R13}{test}{.pdf}%
\gitExec{}{https://github.com/thomasWeise/texgit_tex}{%
examples}{python3 make_pdf.py (?R13?)}%
%
\gitIf{R13}{%
\fbox{\includegraphics[width=0.9\linewidth]{\gitFile{R13}}}%
}{}%
%
\end{document}%
```

Listing 10: The Python script invoked in Listing 9 to create a graphic and store it in an argument file.

```python
"""Create a PDF and store it in the file provided as command line argument."""
from sys import argv

from fpdf import FPDF

pdf = FPDF()  # Create an instance of an FPDF class.
pdf.add_page()  # Add a page.

for i in range(1000):  # Draw 1000 almost-random ellipses
    pdf.set_fill_color((i * 4919) % 255, (i * 3527) % 255,
                       (i * 6329) % 255)
    pdf.ellipse((i * 3677) % 130, (i * 5003) % 170,
                (i * 4391) % 130, (i * 9049) % 170, "F")

# Write some text.
pdf.set_font("helvetica", style="B", size=60)  # Set the font.
pdf.set_text_color(255, 255, 255)  # Set text color to white.
pdf.cell(200, 10, text="Hello World!", align="C")

# Here is the important part! argv[1] is our \gitArg result!
pdf.output(argv[1])  # Save the PDF to the specified file.
```

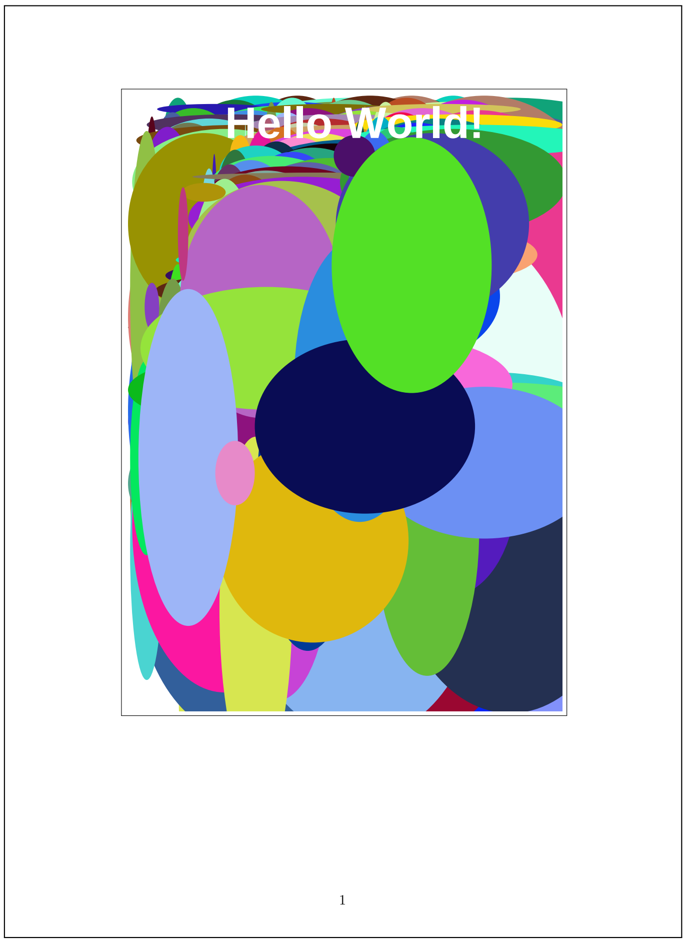


Figure 8: The rendered result of Listing 9 (with trimmed page margins and bottoms).

5 A Note on LaTeX Floating Environments

LaTeX offers us floating environments for laying out tables and figures. Such environments, e.g., `\begin{figure}...\end{figure}` or `\begin{table}...\end{table}` are automatically arranged by LaTeX. This means that their order can change depending how they layout engine determines where to put them. This means that, if you put `\gitLoad`, `\gitArg`, or `\gitExec` commands into such environments, their execution order can change. It is therefore recommended to put these commands always outside of these environments. Then the execution order is always the same and clear.

6 Usage in GitHub Actions

It is possible to use `texgit` in GitHub actions. For this purpose, you would simply copy the `texgit.sty` file into the directory of your LaTeX sources. You can use xu-cheng/texlive-action, which provides TexLive, to run several `pdflatex` / `texgit` cycles. Then, you could deploy the book that was built using JamesIves/github-pages-deploy-action to a GitHub page. This way, whenever you commit changes LaTeX sources residing in a `git` repository, your book would be re-built. The projects below do it this way.

7 Projects using texgit

`texgit` was developed specifically for teaching material book projects. There are several book projects that use it.

- *Programming with Python*, available at `https://thomasWeise.github.io/programmingWithPython` is a book and a set of slides introducing the reader to Python programming. It is enriched with many source code examples, which reside in the repository `https://github.com/thomasWeise/programmingWithPythonCode`. These examples are automatically included and executed in the book building process via `texgit`.
- *Databases*, available at `https://thomasWeise.github.io/databases` is a book and a set of slides introducing the reader to relational databases. It is enriched with many source code examples, which reside in the repository `https://github.com/thomasWeise/databasesCode`. These examples are automatically included and executed in the book building process via `texgit`. Actually, they are `sql` scripts that run on a PostgreSQL database management system. With `texgit`, you can run complex scripts residing in `git` repositories. It is possible to interact with a database server without problems.

8 Implementation

The names of all internal elements of the package are prefixed with `@texgit@`. This naming convention should prevent any name clashes with other packages.

Our `texgit` package requires only one other package:

1. `filecontents` [1] is used to allow us to generate the dummy file on the fly. This package is obsolete for the most recent LaTeX version, where it simply does nothing, but may help us to get our package to work on older systems.

```
\RequirePackage{filecontents}%  Allow us to create the dummy file.
```

```
%
% This is the path to the dummy file.
% The dummy file is created directly below.
% The dummy file is referenced by all invocations of |\gitFile| until the
% Python package has been applied to the |.aux| file and has loaded the
% actual files.
\xdef\@texgit@dummyPath{\jobname.texgit.dummy}%  the dummy file
%
% Create the dummy file that replaces files before they are loaded.
% This file only has one line with one single space.
\expandafter\begin\expandafter{filecontents*}{\@texgit@dummyPath}

\end{filecontents*}
%
%% We need to make sure that the texgit postprocessor is actually
%% properly applied.
%% If you load our package, then if the postprocessor is not applied
%% before the second pdflatex pass, we make sure that pdflatex
%% crashes with an error.
\protected\gdef\@texgit@needsTexgitPass{%
\errmessage{texgit: You must run the Python companion package of texgit %
*before* doing the second pdflatex pass. %
You can do this by invoking 'python3 -m texgit.run \jobname'. %
You can obtain this Python package via 'pip install texgit'. %
Check the documentation at %
https://thomasweise.github.io/texgit_tex/texgit.pdf%
}%
%% force quit
\batchmode\read-1 to \foo%
}%
%% Make sure that the texgit postprocessor is actually applied.
\AtEndDocument{%
\let\@texgit@needsTexgitPass\relax%
\immediate\write\@mainaux{\noexpand\@texgit@needsTexgitPass}}%
%
% The dummy URL that is returned by |\gitURL| unless a proper URL is
% available.
\xdef\@texgit@dummyUrl{https://example.com}%  the dummy URL
%
% This command does nothing and is just a placeholder in the |aux| files.
\gdef\@texgit@gitFile#1#2#3#4{}%
% This command as well.
\gdef\@texgit@process#1#2#3#4{}%
```

```
% This command as well.
\gdef\@texgit@argFile#1#2#3{}%
```

`\gitLoad` The macro `\gitLoad{`⟨*id*⟩`}{`⟨*repositoryURL*⟩`}{`⟨*path*⟩`}{`⟨*postProcessing*⟩`}` defines a query to a `git` repository. The query is stored in the `aux` file of the project and carried out by the Python companion package (see Section 2.6). The results of this macro will become available via the two other macros `\gitFile{id}` and `\gitUrl{id}`. During the first LaTeX build, these macros will return a path to a dummy file which only has a single space character in it followed by a newline and the dummy URL `https://example.com`, respectively. As said, `\gitLoad` will store all information in the `aux` file, which then permits the `texgit` Python package to download (and optionally post-process) the actual file. In the second round of LaTeX building, `\gitFile{id}` and `\gitUrl{id}` will then return the local path to that downloaded file and the actual URL, respectively.

`{`⟨*id*⟩`}` the uniquely chosen ID for this request. Imagine this being something like a label.

`{`⟨*repositoryURL*⟩`}` is the URL of the `git` repository. It could, e.g., be `https://github.com/thomasWeise/texgit_tex` or `ssh://git@github.com/thomasWeise/texgit_tex` or any other valid repository URL.

`{`⟨*path*⟩`}` is then the path to the file within the repository. This could be, for example, `latex/texgit.dtx`.

`{`⟨*postProcessing*⟩`}` Can either be empty, in which case the repository is downloaded and the the local path to the file is returned. It can also be shell command, e.g., `head -n 5`. In this case, the contents of the file are piped to `stdin` of the command and the text written to the `stdout` by the command is stored in a file whose path is returned.

```
%%
%% Define a query to load and post-process a file from a |git| repository.
%% #1 is the request ID
%% #2 is the repository URL
%% #3 is the path to the file inside the repository
%% #4 is a command through which the file contents should be piped
%%%   (leave #4 empty to use the file as-is)
\protected\gdef\gitLoad#1#2#3#4{%
\edef\@texgit@pA{#1}%  fully expand the request ID
\edef\@texgit@pB{#2}%  fully expand the repository URL
\edef\@texgit@pC{#3}%  fully expand the path into the repository
\edef\@texgit@pD{#4}%  fully expand the (optional) shell command
% Write the parameters to the aux file.
\immediate\write\@mainaux{%
\noexpand\@texgit@gitFile{\@texgit@pA}{\@texgit@pB}{%
\@texgit@pC}{\@texgit@pD}}%
}%
```

`\gitExec` The macro `\gitExec{`⟨*id*⟩`}{`⟨*repositoryURL*⟩`}{`⟨*path*⟩`}{`⟨*theCommand*⟩`}` defines a command to be executed either inside a `git` repository or in the current directory. The query is stored in the `aux` file of the project and carried out by the Python companion package (see Section 2.6). The results of this macro will become available via the macro `\gitFile{id}`. During the first LaTeX build, this macro will return a path to a dummy file which only has a single space character in it followed by a newline. As said, `\gitExec` will store all information in the `aux` file, which then permits the `texgit` Python package to download (and optionally post-process) the actual file. In the second round of LaTeX building, `\gitFile{id}` will then return the local path to the file with the standard output of the executed command.

{⟨*id*⟩} the uniquely chosen ID for this request. Imagine this being something like a label.

{⟨*repositoryURL*⟩} is the URL of the `git` repository. It could, e.g., be `https://github.com/thomasWeise/texgit_tex` or `ssh://git@github.com/thomasWeise/texgit_tex` or any other valid repository URL. You can leave this argument empty if you want to execute the command in the current directory.

{⟨*path*⟩} is then the path to the directory within the repository. This could be, for example, `latex`. The command is executed at this directory. Use `.` for the repository root. Leave this empty if no repository is used.

{⟨*theCommand*⟩} The command line to be executed. It can also be shell command, e.g., `python3 --version`. The standard output produced by this command is captured as file.

```
\newcount\@texgit@counter%  The counter for the dummy arguments of \gitExec
\@texgit@counter0\relax%    We start the counter at 0.
%%
%% Define a query to execute a command, optionally in a |git| repository.
%% #1 is the request ID, leave empty if you do not want to access the output
%% #2 is the repository URL, or empty if no repository is needed
%% #3 is the path to a directory inside the repository or empty
%% #4 is a command to be executed
\protected\gdef\gitExec#1#2#3#4{%
\edef\@texgit@pA{#1}%  the request ID
\ifx\@texgit@pA\empty\relax\edef\@texgit@pA{%
@texgit@dummy@\the\@texgit@counter}%
\global\advance\@texgit@counter by 1% step the counter for the next dummy
\fi%
\edef\@texgit@pB{#2}%  fully expand the repository URL, or empty
\edef\@texgit@pC{#3}%  fully expand the path into the repository, or empty
\edef\@texgit@pD{#4}%  fully expand the (optional) shell command
% Write the parameters to the aux file.
\immediate\write\@mainaux{%
\noexpand\@texgit@process{\@texgit@pA}{\@texgit@pB}{%
\@texgit@pC}{\@texgit@pD}}%
}%
```

`\gitArg` The macro `\gitArg{`⟨*id*⟩`}{`⟨*prefix*⟩`}{`⟨*suffix*⟩`}` allocates a unique file name with the given `prefix` and `suffix` that can be passed as argument in invocations of `\gitExec`. The results of this macro will become available via the macro `\gitFile{id}`. During the first LaTeX build, this macro will return a path to a dummy file which only has a single space character in it followed by a newline. As said, `\gitExec` will store all information in the `aux` file, which then permits the `texgit` Python package to download (and optionally post-process) the actual file. In the second round of LaTeX building, `\gitFile{id}` will then return the local path to the file with the standard output of the executed command.

`{`⟨*id*⟩`}` the uniquely chosen ID for this request. Imagine this being something like a label.

`{`⟨*prefix*⟩`}` is the optional prefix, which can be left empty.

`{`⟨*suffix*⟩`}` is the optional suffix, which can be left empty.

```
%%
%% Define a query to allocate a file.
%% #1 is the request ID
%% #2 is the prefix
%% #3 is the suffix
\protected\gdef\gitArg#1#2#3{%
\edef\@texgit@pA{#1}%  the request ID
\edef\@texgit@pB{#2}%  fully expand the prefix
\edef\@texgit@pC{#3}%  fully expand the suffix
% Write the parameters to the aux file.
\immediate\write\@mainaux{%
\noexpand\@texgit@argFile{\@texgit@pA}{\@texgit@pB}{%
\@texgit@pC}}%
}%
```

`\gitIf` The macro `\gitIf{`⟨*id*⟩`}{`⟨*ifDone*⟩`}{`⟨*ifNotDone*⟩`}` executes `ifDone` if the request under the given `id` has already been executed, otherwise executes `ifNotDone`.

`{`⟨*id*⟩`}` the request ID.

`{`⟨*ifDone*⟩`}` executed if the request with the given ID has already been executed.

`{`⟨*ifNotDone*⟩`}` executed if the request with the given ID has not yet been executed.

```
%%
%% Conditionally execute code depending on whether the given request was
%% already executed.
%% #1 is the request ID
%% #2 the code to execute if the request of the given ID has been executed.
%% #3 the code to execute if the request of the given ID has NOT been
%%    executed.
\gdef\gitIf#1#2#3{%
\expandafter\ifx\csname @texgit@path@#1\endcsname\relax%
\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{#3}{#2}%
}%
```

`\gitFile` The macro `\gitFile{`⟨*id*⟩`}` get the file path associated with the request of ID `id`.

`{`⟨*id*⟩`}` the request ID. yet been executed.

```
%%
%% Get the file path associated with the request ID.
%% #1 is the request ID
\gdef\gitFile#1{%
\expandafter\ifx\csname @texgit@path@#1\endcsname\relax%
\@texgit@dummyPath\else\csname @texgit@path@#1\endcsname\fi%
}%
```

\gitUrl The macro \gitUrl{⟨*id*⟩} get the URL associated with the request of ID id.

{⟨*id*⟩} the request ID. yet been executed.

```
%%
%% Get the URL associated with the request ID.
%% #1 is the request ID
\gdef\gitUrl#1{%
\expandafter\ifx\csname @texgit@url@#1\endcsname\relax%
\@texgit@dummyUrl\else\csname @texgit@url@#1\endcsname\fi%
}%
```

\gitName The macro \gitName{⟨*id*⟩} gets the base name of the path associated with the request of ID id. Like \gitUrl, this makes only sense if the file is hosted in a git repository. If the file contains the data generated by a process, then this command returns an empty string.

{⟨*id*⟩} the request ID. yet been executed.

```
%%
%% Get the base name of the path associated with the request ID.
%% #1 is the request ID
\gdef\gitName#1{%
\expandafter\ifx\csname @texgit@name@#1\endcsname\relax%
\else\csname @texgit@name@#1\endcsname\fi%
}%
```

\gitNameEsc The macro \gitNameEsc{⟨*id*⟩} gets the escaped base name of the path associated with the request of ID id. Sometimes, file names may contain characters like _ or $. This means that you could not print the result of \getFileName in plain TEX, because it would cause an error during compilation. \getNameEsc escapes _, $, and to _, \$, and ~, respectively. Therefore, its result can be typeset like normal text, at least in \texttt{...}. Like \gitUrl, this makes only sense if the file is hosted in a git repository. If the file contains the data generated by a process, then this command returns an empty string.

{⟨*id*⟩} the request ID. yet been executed.

```
%%
%% Get the base name of the path associated with the request ID.
%% #1 is the request ID
\gdef\gitNameEsc#1{%
\expandafter\ifx\csname @texgit@escName@#1\endcsname\relax%
\else\csname @texgit@escName@#1\endcsname\fi%
}%
```

References

[1] Scott Pakin. The **filecontents** package. *CTAN Comprehensive TEX Archive Network*, April 2, 2023. URL https://ctan.org/pkg/filecontents

Change History

0.8.0
General: the initial draft version . . 1

0.8.1
General: slightly improved documentation 1

0.8.2
General: improved texgit.tds.zip . . 1

0.8.4
General: improved build process . . 1

0.8.5
General: improved examples: added an example with multiple git command results . . 1

0.9.0
General: complete overhaul of the API 1

0.9.1
General: improved documentation 1

0.9.2
General: gitExec with empty ID permitted 1

0.9.3
General: better documentation . . . 1

0.9.4
General: name change to texgit . . 1

0.9.5
General: improved documentation and slight updates 1

0.9.6
General: now forcing pdflatex to crash if Python postprocessor is not applied 1

0.9.7
General: now offering file names (via gitName) and escaped file names (via gitNameEsc) 1

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.