

The `latexgit` Package

Thomas Weise
Institute of Applied Optimization
School of Artificial Intelligence and Big Data
Hefei University, Hefei 230601, Anhui, China
tweise@hfu.edu.cn · tweise@ustc.edu.cn

September 21, 2024

Abstract

This \LaTeX package works in combination with the Python package `latexgit`. It offers the command `\gitLoad`, which can load a file from a `git` repository, optionally post-process it, and then provide a local path to the file as macro `\gitFile` and the URL to the original file in `\gitUrl`. Using the `\gitFile` macro, you can then include the file in \LaTeX directly or load it as source code listing. It also offers the command `\gitExec`, which can execute an arbitrary command, either in the current directory or inside a directory of a `git` repository and fetch the standard output into a local file, the path to which is made available to the file again as macro `\gitFile` and the URL to the repository in which the command was executed becomes `\gitUrl`. The functionality 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, stores the result in local files, and adds the resolved paths to the `aux` file. During the first `pdflatex` run, `\gitFile` and `\gitUrl` offer dummy results. However, after the resolution via the Python package, in the second `pdflatex` run they then point to the right data.

Contents

1	Introduction	2
1.1	Addressed Problem and Use Case	2
1.2	Provided Functionality	2
2	Usage	3
2.1	Installation	3
2.2	Loading the Package	4
2.3	Querying a File from a <code>git</code> Repository	4
2.4	Executing a Command (optionally inside a <code>git</code> Repository)	4
2.5	Executing the Python Package	5
2.6	A Note on Virtual Environments	5

3	Provided Macros	6
3.1	Examples	7
3.2	Minimal Working Example	7
3.3	The Second Example: Multiple Files and Post-Processing	10
3.4	The Third Example: Using the <code>listings</code> Package	13
3.5	The Fourth Example: Using Git Commands in Macros	14
3.6	The Fifth Example: Capturing the Output of a Program	16
3.7	The Sixth Example: Capturing the Output of a Program Executed Inside a git Repository	17
3.8	The Seventh Example: Capturing the Output of Multiple Programs Executed Inside Different git Repositories	18
4	Implementation	20

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 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 actual 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 re-compile 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.

1.2 Provided Functionality

It does so by offering a combination of a \LaTeX package (this package here) and a Python program (published at <https://pypi.org/projects/latexgit>). This

L^AT_EX provides the command `\gitLoad` that can load a specific file (its second argument) from a specific `git` repository (its first argument) and, optionally, pipe the file contents through a program for post-processing (the third argument, which can be left empty). It also provides the command `\gitExec`, which, too, has three arguments. This time, the first two arguments (the `git` repository URL and the path to a directory inside the repository in which the command should be executed) can be left empty. The third argument, however, is the command line to be executed whose standard output should be fetched. Both 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 [L^AT_EX 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 and using the paths to files holding their results (see [Section 2.3](#)).
4. Finally, the Python package can carry them out after the first `pdflatex` run and in the next `pdflatex` run, `\gitFile` and `\gitUrl` are defined appropriately, see [Section 2.5](#).

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

```
pdflatex article
python3 -m latexgit.aux article
pdflatex article
```

Comprehensive examples are provided in [Section 3.1](#).

2.1 Installation

2.1.1 L^AT_EX Package

First, make sure that you have the `latexgit.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/latexgit_tex/latexgit.sty directly. You can now copy it into the folder of your document.
2. You can download `latexgit.dtx` and `latexgit.ins` from https://thomasweise.github.io/latexgit_tex/latexgit.dtx and https://thomasweise.github.io/latexgit_tex/latexgit.ins. You can then execute

```
pdflatex latexgit.ins
```

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

3. Or you can download the `latexgit.tds.zip` file from https://thomasweise.github.io/latexgit_tex/latexgit.tds.zip and unpack it into your $\text{T}_{\text{E}}\text{X}$ tree. If you know what that is.

2.1.2 Python Package

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

```
pip install latexgit
```

2.2 Loading the Package

Load this package using

```
\usepackage{latexgit}
```

The package has no options or parameters. Loading it will automatically load the packages `alphalph` and `filecontents` as well, see [Section 4](#).

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{repositoryURL}{path}{}
```

After this command is executed, a local path to the file becomes available in the fully-expandable command `\gitFile`. The full URL to the file in the `git` repository, including the current commit id, becomes available in the fully-expandable command `\gitUrl`. Both `\gitFile` and `\gitUrl` will be overwritten every time `\gitLoad` or `\gitExec` (see later) are invoked. You can invoke `\gitLoad` any number of times.

The third parameter, left empty in the above example, can specify an optional post-processing command. 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` 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{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{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 fully-expandable command `\gitFile`. If the command was executed in a `git` repository, then the URL to the `git` repository becomes available in the fully-expandable command `\gitUrl` (otherwise, this command expands to the empty string). Both `\gitFile` and `\gitUrl` will be overwritten every time `\gitLoad` or `\gitExec` are invoked. You can invoke `\gitLoad` any number of times.

2.5 Executing the Python Package

During the first `pdflatex` run, `\gitFile` points to an empty dummy file (`\jobname.latexgit.dummy`) and `\gitUrl` points to `http://example.com`. Both commands will only expand to useful information if the Python package `latexgit` 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 latexgit.aux myfile
```

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

```
python3 -m latexgit.aux myfile.aux
```

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

2.6 A Note on 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 `latexgit` 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 `latexgit` package. The implementation of these macros is given in [Section 4](#) and several examples can be found in [Section 3.1](#).

\gitLoad The macro `\gitLoad{<repositoryURL>}{<path>}{<postProcessing>}` provides a local path to a file from a `git` repository.

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

`{<path>}` is then the path to the file within the repository. This could be, for example, `latex/latexgit.dtx`.

`{<postProcessing>}` Can either be empty, in which case the repository is downloaded and 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.

After invoking this command, two new commands will be defined:

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

\gitUrl 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.

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

`{<repositoryURL>}` is the URL of the `git` repository. It could, e.g., be https://github.com/thomasWeise/latexgit_tex or ssh://git@github.com/thomasWeise/latexgit_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, two new commands will be defined:

\gitFile returns the path to the file in which the standard output is stored.

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

```

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

```

`\gitUrl` returns the full URL to the `git` repository, if any was specified, or the empty string otherwise. online. This command works for GitHub, but it may not provide the correct URL for other repository types.

`\gitFile` The macro `\gitFile` returns the path to the file with the contents of the latest `\gitLoad` or `\gitExec` request. 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` will point to the proper file during the next `pdflatex` pass.

`\gitUrl` The macro `\gitUrl` returns the URL from which the file corresponding to the latest `\gitLoad` request was downloaded. Alternatively, it returns the URL of the `git` repository of the last `\gitExec` invocation. 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).

During the first `pdflatex` pass, this will be <https://example.com>. After the Python package has been applied to the `aux` file, then `\gitUrl` will point to the proper file during the next `pdflatex` pass.

3.1 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 3.2 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 3.3 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 3.4 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 3.5 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`.

3.2 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.

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

```

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

```

As you can see in Listing 1, we first load the package `latexgit` in line 2. Inside the document, we define a `git` request via the `\gitLoad` command. This command takes the URL of a `git` repository as first parameter. In this case, this is https://github.com/thomasWeise/latexgit_tex, which happens to be the URL where you can find the repository of this package on GitHub. The second 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 third parameter shall be ignored for now.

After defining the request, we can now use two commands, `\gitFile` and `\gitUrl`. 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.latexgit.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 latexgit.aux 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 be accessed by \LaTeX . Usually, it will create a folder `__git__` in your project's directory and place the file there.

Anyway, during the second \LaTeX or `pdflatex` pass, `\gitFile` points to a valid file path with actual contents. By doing `\input{\gitFile}`, 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 latexgit.aux 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.

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).

`\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 `latexgit` 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 `latexgit` command several times.

3.3 The Second Example: Multiple Files and Post-Processing

In [Listing 3](#) we, use `latexgit` 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 `latexgit` Python package. The result can be seen in [Figure 2](#).

Listing 3: An example using the `latexgit` package, rendered as [Figure 2](#).

```
1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{verbatim}% for loading a file verbatim
4 \usepackage{colorlinks}{hyperref}% for printing the URL
5 \begin{document}%
6 %
7 \section{First File}%
8 First, we load a file from the GitHub repository
9 “\url{https://github.com/thomasWeise/latexgit\_py}”, where the Python complement
10 package of our \LaTeX package is located. We will then include this file verbatim
11 without any modification.
12
13 \gitLoad{https://github.com/thomasWeise/pycommons}{pycommons/io/console.py}{}%
14 % now, \gitFile and \gitUrl are defined and can be used.
15 \verbatiminput{\gitFile}% print the contents of the file
16 The above file was loaded from URL \url{\gitUrl}.% print the url
17 %
18 \clearpage\section{Second File}%
19 We now load the same file again, but this time retain only the first five lines.
20 We do this by specifying that the file contents should be piped through
21 “\verb=head -n 5=” before inclusion.
22 \gitLoad{https://github.com/thomasWeise/pycommons}{%
23 pycommons/io/console.py}{head -n 5}%
24 % now, \gitFile and \gitUrl are defined and can be used.
25 \verbatiminput{\gitFile}% print the contents of the file
26 The above file was loaded from URL \url{\gitUrl}.% print the url
27 %
28 \clearpage\section{Third File}%
29 We now load a file from the “\url{https://github.com/thomasWeise/moptipy}”
30 GitHub repository. The contents of this file will be piped through the Python code
31 formatter, which retains only a snippet of the code and removes type hints and
32 comments, while keeping the doc strings. (It doesn’t really matter what it does,
33 it is just postprocessing.)
34 \gitLoad{https://github.com/thomasWeise/moptipy}{moptipy/api/encoding.py}{%
35 python3 -m latexgit.formatters.python --labels book --args doc}% post-processor
36 % now, \gitFile and \gitUrl are defined and can be used.
37 \verbatiminput{\gitFile}% print the contents of the file
38 The above file was loaded from URL \url{\gitUrl}.% print the url
39 %
40 \end{document}%
```

The file `example_2.tex` shown in Listing 3 begins by loading our `latexgit` 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 our Python companion package `latexgit.py`. The sources of this package are available in the GitHub repository https://github.com/thomasWeise/latexgit_py. We download the file `latexgit/utils/console.py`, which is just a small utility for printing log strings to the output together with a time mark. The full git request contains these two components.

Issuing this request will set the command `\gitFile` to the local file containing the downloaded contents of `latexgit/utils/console.py` from the repository https://github.com/thomasWeise/latexgit_py. The command `\gitUrl` 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 third parameter of `\gitLoad` is specified. If the third 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. `\gitFile` 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`.

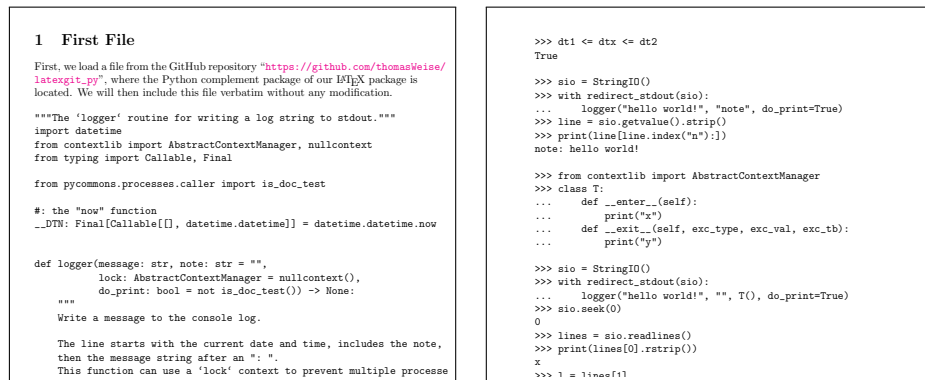
The resulting output in Figure 2b looks thus similar to Figure 2a, but only imports this first five lines from the downloaded file.

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

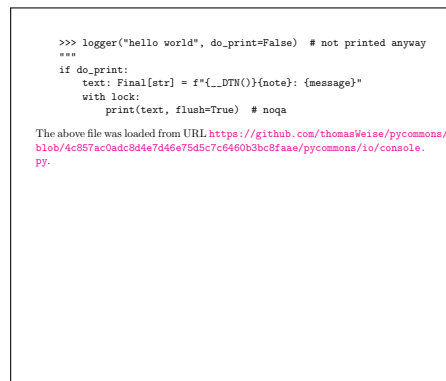
```
pdflatex example_2
python3 -m latexgit.aux example_2
pdflatex example_2
```

Side note: Such post-processing steps are cached by the Python companion package `latexgit` 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 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



(a) Page 1 of the pdf compiled from Listing 3. (b) Page 2 of the pdf compiled from Listing 3.



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

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

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 `latexgit` Python package also offers a Python code formatter, namely the executable module `latexgit.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](https://thomasweise.github.io/latexgit_py) of the Python companion package at https://thomasweise.github.io/latexgit_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.

```

1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{xcolor}% to be able to use colors
4 \usepackage{colorlinks}{hyperref}% for printing the URL
5 \usepackage{listings}% importing external code
6 \lstset{language=Python,basicstyle=\small\ttfamily,%
7 keywordstyle=\ttfamily\color{teal!90!black}\bfseries,%
8 identifierstyle=,commentstyle=\color{gray}\footnotesize,%
9 stringstyle=\ttfamily\color{red!90!black},%
10 numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
11 backgroundcolor=\color{black!10!yellow!5!white}}%
12 \begin{document}%
13 %
14 Behold the beautiful \autoref{1}.%
15 %
16 \gitLoad{https://github.com/thomasWeise/moptipy}{%
17 moptipy/algorithms/so/rls.py}{%
18 python3 -m latexgit.formatters.python --labels book}% post-processor
19 %
20 \lstinputlisting[label=1,caption={%
21 The RLS Algorithm. (\href{\gitUrl}{src})}{\gitFile}%
22 %
23 \end{document}%

```

3.4 The Third Example: Using the listings Package

Finally, 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 latexgit.aux 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 latexgit.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.

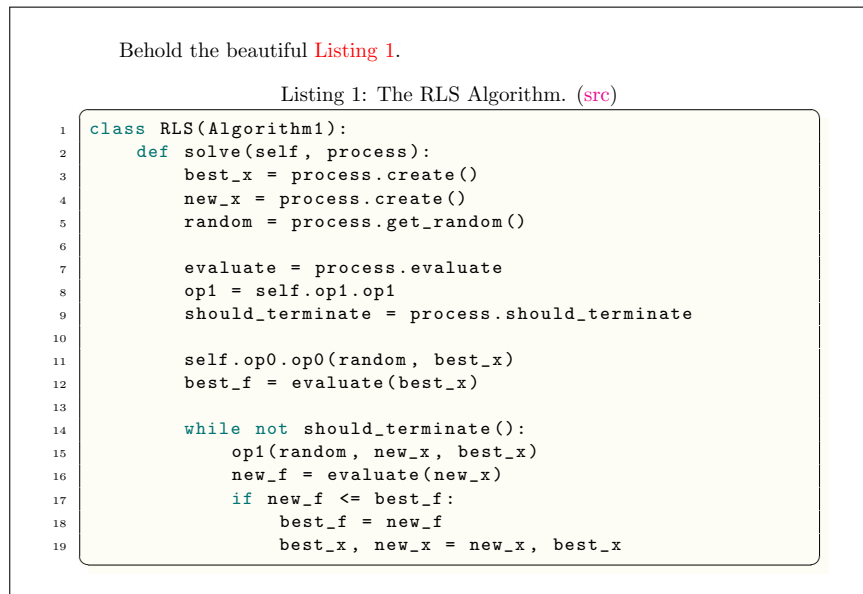


Figure 3: The rendered result of Listing 4 (with trimmed page margins and bottom).

Then it re-formats the code to save space. Again, check out the documentation of our `latexgit` Python companion package at https://thomasweise.github.io/latexgit_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.

3.5 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 `latexgit` package into L^AT_EX 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. 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 latexgit.aux example_4
pdflatex example_4

```

(if the example code from Listing 5 was stored in a file called `example_4.tex`, that is.)

Listing 5: An example using commands from the latexgit package in macros, rendered as Figure 4.

```

1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{xcolor}% to be able to use colors
4 \usepackage{colorlinks}{hyperref}% for printing the URL
5 \usepackage{listings}% importing external code
6 \lstset{language=Python,basicstyle=\small\ttfamily,%
7 keywordstyle=\ttfamily\color{teal!90!black}\bfseries,%
8 identifierstyle=\commentstyle=\color{gray}\footnotesize,%
9 stringstyle=\ttfamily\color{red!90!black},%
10 numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
11 backgroundcolor=\color{black!10!yellow!5!white}}%
12 %
13 \gdef\moptipySrc#1#2#3{%
14 \gitLoad{https://github.com/thomasWeise/moptipy}{moptipy/#3}{%
15 python3 -m latexgit.formatters.python --labels book}%
16 \lstinputlisting[float,label={#1},caption={#2~(\href{\gitUrl}{src})}]{\gitFile}}
17 %
18 \begin{document}%
19 %
20 Behold the beautiful \autoref{a} and \autoref{b}.%
21 %
22 \moptipySrc{a}{Randomized Sampling}{algorithms/random_sampling.py}
23 \moptipySrc{b}{Randomized Local Search}{algorithms/so/rls.py}
24 %
25 \end{document}%

```

Listing 1: Randomized Sampling (src)

```

1 class RandomSampling(Algorithm0):
2     def solve(self, process):
3         x = process.create()
4         random = process.get_random()
5
6         evaluate = process.evaluate
7         op0 = self.op0.op0
8         should_terminate = process.should_terminate
9
10        while not should_terminate():
11            op0(random, x)
12            evaluate(x)

```

Behold the beautiful Listing 1 and Listing 2.

Listing 2: Randomized Local Search (src)

```

1 class RLS(Algorithm1):
2     def solve(self, process):
3         best_x = process.create()
4         new_x = process.create()
5         random = process.get_random()
6
7         evaluate = process.evaluate
8         op1 = self.op1.op1
9         should_terminate = process.should_terminate
10
11        self.op0.op0(random, best_x)
12        best_f = evaluate(best_x)
13
14        while not should_terminate():
15            op1(random, new_x, best_x)
16            new_f = evaluate(new_x)
17            if new_f <= best_f:
18                best_f = new_f
19                best_x, new_x = new_x, best_x

```

(a) Page 1 of the pdf compiled from Listing 5. (b) Page 2 of the pdf compiled from Listing 5.

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

3.6 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 latexgit.aux example_5
pdflatex example_5
```

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

Listing 6: An example of capturing the output of a program, rendered as [Figure 5](#).

```
1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{xcolor}% to be able to use colors
4 \usepackage[colorlinks]{hyperref}% for printing the URL
5 \usepackage{listings}% importing external code
6 \lstset{language={},basicstyle=\small\ttfamily,%
7 numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
8 backgroundcolor=\color{black!10!yellow!5!white}}%
9 %
10 \begin{document}%
11 %
12 Check the output of a simple command in \autoref{lst:out}:
13 %
14 \gitExec{python3 --version}%
15 \lstinputlisting[float,label={lst:out},caption={%
16 The result of \texttt{python3 {-}{-} version}.}]{\gitFile}
17 %
18 \end{document}%
```

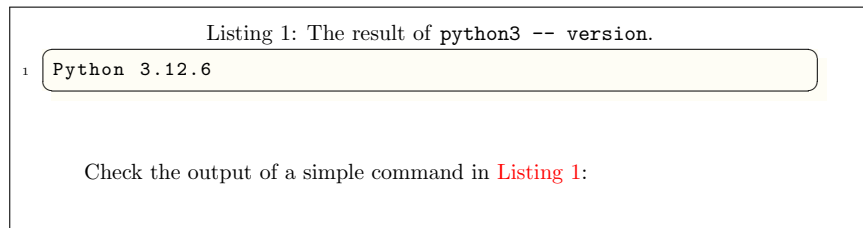



Figure 5: The rendered result of Listing 6 (with trimmed page margins and bottoms).

3.7 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
python3 -m latexgit.aux example_6
pdflatex example_6
```

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

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

```

1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{xcolor}% to be able to use colors
4 \usepackage[colorlinks]{hyperref}% for printing the URL
5 \usepackage{listings}% importing external code
6 \lstset{language={},basicstyle=\small\ttfamily,%
7 numbers=left,numberstyle=\tiny,frame=shadowbox,framewidth=10pt,%
8 backgroundcolor=\color{black!10!yellow!5!white}}%
9 %
10 \begin{document}%
11 %
12 Check the output of a simple command in \autoref{lst:out}:
13 %
14 \gitExec{}{}{python3 --version}%
15 \lstinputlisting[float,label={lst:out},caption={%
16 The result of \texttt{python3 {-}{-} version}.}]{\gitFile}
17 %
18 \end{document}%

```

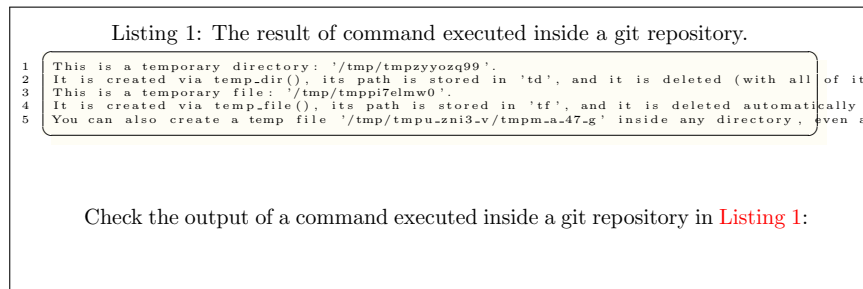


Figure 6: The rendered result of Listing 7 (with trimmed page margins and bottoms).

3.8 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
```

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

```

1 \documentclass{article}%
2 \usepackage{latexgit}% use our package
3 \usepackage{xcolor}% to be able to use colors
4 \usepackage[colorlinks]{hyperref}% for printing the URL
5 \usepackage{listings}% importing external code
6 \lstset{language={},basicstyle=\small\ttfamily,%
7 numbers=left,numberstyle=\tiny,frame=shadowbox,frameround=tttt,%
8 backgroundcolor=\color{black!10!yellow!5!white}}%
9 %
10 \begin{document}%
11 %
12 Check the output of a simple command in \autoref{lst:out}:
13 %
14 \gitExec{python3 --version}%
15 \lstinputlisting[float,label={lst:out},caption={%
16 The result of \texttt{python3 {-}{-} version.}}{\gitFile}
17 %
18 \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.

```
1 This is a temporary directory: '/tmp/tmpzyyozq99'.
2 It is created via temp_dir(), its path is stored in 'td', and it is deleted (with all of it
3 This is a temporary file: '/tmp/tmppt7elnw0'.
4 It is created via temp_file(), its path is stored in 'tf', and it is deleted automatically
5 You can also create a temp file '/tmp/tmpu-zni3-v/tmpm-a-47-g' inside any directory, even a
```

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

```
1 Hello World!
```

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

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

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

```
python3 -m latexgit.aux example_7
pdflatex example_7
```

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

4 Implementation

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

Our `latexgit` package requires only one other package:

1. `alphalph` [1] is required to translate $\text{T}_{\text{E}}\text{X}$ counters to alphabetic series for counters that are outside of the range 1...26. Basically, for each file we include from `git`, we store the corresponding local path in a command of the structure `\@latexgit@pathXXX` where the `XXX` is an alphabetical sequence which is increasing in the form “a,” “b,” ..., “y,” “z,” “aa,” ..., “ay,” “az,” “ba,” “zy,” “zz,” “aaa,” “aab,” ...
2. `filecontents` [2] is used to allow us to generate the dummy file on the fly. This package is obsolete for the most recent $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ version, where it simply does nothing, but may help us to get our package to work on older systems.

```

1 \RequirePackage{alphalph}% Convert counters to alphabetical series.
2 \RequirePackage{filecontents}% Allow us to create the dummy file.
3 \newcount\@latexgit@counter% The counter for the git files included.
4 \@latexgit@counter0\relax% We start the counter at 0.
5 %
6 % This is the path to the dummy file.
7 % The dummy file is created directly below.
8 % The dummy file is referenced by all invocations of |\gitFile| until the
9 % Python package has been applied to the |.aux| file and has loaded the
10 % actual files.
11 \edef\@latexgit@dummyPath{\jobname.latexgit.dummy}% the dummy file
12 %
13 % Create the dummy file that replaces git files before they are loaded.
14 % This file only has one line with one single space.
15 \expandafter\begin\expandafter{filecontents*}\{\@latexgit@dummyPath}
16
17 \end{filecontents*}
18 %
19 % This command does nothing and is just a placeholder in the |aux| files.
20 \protected\gdef\@latexgit@gitFile#1#2#3{%
21 % This command as well.
22 \protected\gdef\@latexgit@process#1#2#3{%

```

`\gitLoad` The macro `\gitLoad{<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 2.5](#)). This macro will define two other macros, `\gitFile` and `\gitUrl`. During the first $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 URL <https://example.com>, respectively. As said, `\gitLoad` will store all information in the `aux` file, which then permits the `latexgit` Python package to download (and optionally post-process) the actual file. In the second round of $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ building, `\gitFile` and `\gitUrl` will then return the local path to that downloaded file and the actual URL, respectively.

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

`{\path}` is then the path to the file within the repository. This could be, for example, `latex/latexgit.dtx`.

`{\postProcessing}` Can either be empty, in which case the repository is downloaded and 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.

```

23 %%
24 %% Define a query to load and post-process a file from a |git| repository.
25 %% #1 is the repository URL
26 %% #2 is the path to the file inside the repository
27 %% #3 is a command through which the file contents should be piped
28 %% (leave #3 empty to use the file as-is)
29 \protected\gdef\gitLoad#1#2#3{%
30 \edef\@latexgit@pA{#1}% fully expand the repository URL
31 \edef\@latexgit@pB{#2}% fully expand the path into the repository
32 \edef\@latexgit@pC{#3}% fully expand the (optional) shell command
33 % Write the parameters to the aux file.
34 \immediate\write\@mainaux{%
35 \noexpand\@latexgit@gitFile{\@latexgit@pA}{\@latexgit@pB}{\@latexgit@pC}}%
36 % Increment the counter for command names by 1.
37 \advance\@latexgit@counter by 1\relax%
38 % We now create the name of the path command based on the structure
39 % |\@latexgit@pathXXX| where |XXX| is a alphabetic sequence representing
40 % the value of |\@latexgit@counter|
41 \edef\@latexgit@pathCmd{\@latexgit@path\alphalph{\the\@latexgit@counter}}%
42 % If the path command exists, then we store it as |\gitFile|.
43 \expandafter\ifcsname\@latexgit@pathCmd\endcsname\relax%
44 \xdef\gitFile{\csname\@latexgit@pathCmd\endcsname}%
45 \else%
46 % But if it does not exist, we assign |\gitFile| to the dummy path.
47 \xdef\gitFile{\@latexgit@dummyPath}%
48 \fi% If we get here, the |\gitFile| command holds a valid path.
49 % We now create the name of the url command based on the structure
50 % |\@latexgit@urlXXX| where |XXX| is a alphabetic sequence representing
51 % the value of |\@latexgit@counter|
52 \edef\@latexgit@urlCmd{\@latexgit@url\alphalph{\the\@latexgit@counter}}%
53 % If the url command exists, then we store it as |\gitUrl|.
54 \expandafter\ifcsname\@latexgit@urlCmd\endcsname\relax%
55 \xdef\gitUrl{\csname\@latexgit@urlCmd\endcsname}%
56 \else%
57 % But if it does not exist, we store the example url in |\gitUrl|.
58 \xdef\gitUrl{http://example.com}%
59 \fi% If we get here, the |\gitUrl| holds a valid URL.
60 }%
```

`\gitExec` The macro `\gitExec{\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 2.5](#)). This macro will define two other macros, `\gitFile` and `\gitUrl`. During the first L^AT_EX 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 URL <https://example.com>, respectively. As said, `\gitExec` will store all information in the aux file, which then permits the `latexgit` Python package to download (and optionally post-process) the actual file. In the second round of L^AT_EX building, `\gitFile` and `\gitUrl` will then return the local path to the file with the standard output of the executed command and the URL to the git repository, respectively.

`{\repositoryURL}` is the URL of the git repository. It could, e.g., be https://github.com/thomasWeise/latexgit_tex or ssh://git@github.com/thomasWeise/latexgit_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.

`{\gitExec}` 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.

```

61 %%
62 %% Define a query to execute a command, optionally in a |git| repository.
63 %% #1 is the repository URL, or empty if no repository is needed
64 %% #2 is the path to a directory inside the repository or empty
65 %% #3 is a command to be executed
66 \protected\gdef\gitExec#1#2#3{%
67 \edef\@latexgit@pA{#1}% fully expand the repository URL
68 \edef\@latexgit@pB{#2}% fully expand the path into the repository
69 \edef\@latexgit@pC{#3}% fully expand the (optional) shell command
70 % Write the parameters to the aux file.
71 \immediate\write\@mainaux{%
72 \noexpand\@latexgit@process{\@latexgit@pA}{\@latexgit@pB}{\@latexgit@pC}}%
73 % Increment the counter for command names by 1.
74 \advance\@latexgit@counter by 1\relax%
75 % We now create the name of the path command based on the structure
76 % |\@latexgit@pathXXX| where |XXX| is a alphabetic sequence representing
77 % the value of |\@latexgit@counter|
78 \edef\@latexgit@pathCmd{\@latexgit@path\alphalph{\the\@latexgit@counter}}%
79 % If the path command exists, then we store it as |\gitFile|.
80 \expandafter\ifcsname\@latexgit@pathCmd\endcsname\relax%
81 \xdef\gitFile{\csname\@latexgit@pathCmd\endcsname}%
82 \else%
83 % But if it does not exist, we assign |\gitFile| to the dummy path.
84 \xdef\gitFile{\@latexgit@dummyPath}%
85 \fi% If we get here, the |\gitFile| command holds a valid path.
86 % We now create the name of the url command based on the structure
87 % |\@latexgit@urlXXX| where |XXX| is a alphabetic sequence representing
88 % the value of |\@latexgit@counter|
89 \edef\@latexgit@urlCmd{\@latexgit@url\alphalph{\the\@latexgit@counter}}%
90 % If the url command exists, then we store it as |\gitUrl|.
91 \expandafter\ifcsname\@latexgit@urlCmd\endcsname\relax%
92 \xdef\gitUrl{\csname\@latexgit@urlCmd\endcsname}%
93 \else%
94 % But if it does not exist, we store the empty url in |\gitUrl|.

```

```

95 \xdef\gitUrl{%
96 \fi% If we get here, the |\gitUrl| holds a valid URL or is empty.
97 }%

```

References

- [1] Heiko Oberdieck. The **alphalph** package. *CTAN Comprehensive T_EX Archive Network*, 2019/12/09 v2.6. URL <https://ctan.org/pkg/alphalph>
- [2] Scott Pakin. The **filecontents** package. *CTAN Comprehensive T_EX Archive Network*, April 2, 2023. URL <https://ctan.org/pkg/filecontents>

Change History

0.8.0	General: the initial draft version . . . 1	0.8.4	General: improved build process . . . 1
0.8.1	General: slightly improved documentation 1	0.8.5	General: improved examples: added an example with multiple git command results . . . 1
0.8.2	General: improved latexgit.tds.zip . . . 1	0.8.6	General: the use case of virtual environments in conjunction with the latexgit Python package of version 0.8.17 or greater is documented 1
0.8.3	General: supporting arbitrary commands via the new latexgit_py version 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.

Symbols	\@latexgit@pathCmd .	\gitUrl 1, 4, 5
\@latexgit@counter 41,	\input 7, 8
. 3, 4, 37,	43, 44, 78, 80, 81	\jobname 8
40, 41, 51, 52,	\@latexgit@pathXXX .	\jobname.latexgit.dummy
74, 77, 78, 88, 89 39, 76 5
\@latexgit@dummyPath	\@latexgit@process .	
. 11, 15, 47, 84 22, 72	A
\@latexgit@gitFile .	\@latexgit@urlCmd 52,	\advance 37, 74
. 20, 35	54, 55, 89, 91, 92	alphalph 4, 20, 23
\@latexgit@pA	\@latexgit@urlXXX 50, 87	\alphalph 41, 52, 78, 89
. 30, 35, 67, 72	\@mainaux 34, 71	aux 1, 5
\@latexgit@pB	_git_ 8, 9, 11	
. 31, 35, 68, 72	\gitExec 1, 4	B
\@latexgit@pC	\gitFile 1, 4, 5	\begin 15
. 32, 35, 69, 72	\gitLoad 1, 4	

C		P	
<code>\csname</code> ..	44, 55, 81, 92	<code>\gitUrl</code>	53, 55, 57, 58, 59, 90, 92, 94, 95, 96
E		R	
<code>\edef</code>	11, 30, 31, 32, 41, 52, 67, 68, 69, 78, 89	<code>pdflatex</code>	1, 3, 5
<code>\else</code>	45, 56, 82, 93	<code>\protected</code>	20, 22, 29, 66
<code>\end</code>	17	PyPI	4
<code>\endcsname</code>	43, 44, 54, 55, 80, 81, 91, 92	Python	2, 4, 8
<code>\expandafter</code>		R	
.	15, 43, 54, 80, 91	<code>\relax</code>	4, 37, 43, 54, 74, 80, 91
F		<code>\RequirePackage</code> ..	1, 2
<code>filecontents</code>	4, 20	S	
G		<code>shell</code>	4, 11
<code>\gdef</code>	20, 22, 29, 66	<code>stdin</code>	4, 6, 11
<code>git</code>	1, 4	<code>stdout</code>	4, 6, 11
<code>\gitExec</code>	61	T	
<code>\gitFile</code>	8, 42, 44, 46, 47, 48, 79, 81, 83, 84, 85	<code>\the</code>	41, 52, 78, 89
<code>\gitLoad</code>	23	W	
		<code>\write</code>	34, 71
		X	
		<code>\xdef</code>	44, 47, 55, 58, 81, 84, 92, 95
		N	
		<code>\newcount</code>	3
		<code>\noexpand</code>	35, 72
		H	
		<code>head</code>	4
		<code>http://example.com</code> ..	5
		I	
		<code>\ifcsname</code>	43, 54, 80, 91
		<code>\immediate</code>	34, 71
		J	
		<code>\jobname</code>	11
		L	
		<code>latexgit.dtx</code>	3
		<code>latexgit.ins</code>	3
		<code>latexgit.sty</code>	4
		<code>latexgit_py</code>	4
		Linux	4