{robust-externalize}

Cache anything (Ti*k*Z, tikz-cd, python...), in a robust, efficient and pure way.

Léo Colisson   Version 2.2+unstable

github.com/leo-colisson/robust-externalize

Contents

    WARNING: This library is young and might lack some testing (and is an important rewrite of a previous version), but v2.2 is definitely relatively robust. Even if we try to stay backward compatible, the only guaranteed way to be immune to changes is to copy/paste the library in your main project folder. Please report any bug to https://github.com/leo-colisson/robust-externalize!

    WARNING 2: the recent versions 2.1 and 2.2 improves significantly the compilation time compared with 2.0 that was surprisingly slow. Now, it is fairly well optimized, and version 2.2 is even more optimized.

1 A taste of this library

This library allows you to cache any language: not only LaTeX documents and TikZ images, taking into account depth and overlays, but also arbitrary code. To use it, make sure to have v2.0 installed (see instructions below), and load:

```
\usepackage{robust-externalize}
\robExtConfigure{
  % To display instructions in the PDF to manually compile pictures if
  % you forgot/do not want to compile with -shell-escape
  enable fallback to manual mode,
}
```

Then type something like this (note the C for `cached` at the end of `tikzpictureC`, see below to override the original command), and compile with `pdflatex -shell-escape yourfile.tex` (or **read below if you do not want to use `-shell-escape` (note that overleaf enables shell-escape by default, and it not needed once all pictures are cached)**):

The next picture is cached My node that respects baseline ❤. and you can see that overlay and depth works.

```
\robExtConfigure{
    add to preset={tikz}{
        % we load some packages that will be loaded by figures based on the tikz preset
        add to preamble={\usepackage{pifont}}
    }
}
The next picture is cached %
\begin{tikzpictureC}[baseline=(A.base)]
    \node[fill=red, rounded corners](A){My node that respects baseline \ding{164}.};
    \node[fill=red, rounded corners, opacity=.3,overlay] at (A.north east){I am an overlay text};
\end{tikzpictureC} and you can see that overlay and depth works.
```

You can also cache arbitrary code (e.g. python). You can also define arbitrary compilation commands, inclusion commands, and presets to fit you need. For instance, you can create a preset to obtain:

The for loop

```
1  for name in ["Alice", "Bob"]:
2      print(f"Hello {name}")
```

Output:

```
Hello Alice
Hello Bob
```

```
\begin{CacheMeCode}{python print code and result, set title={The for loop}}
for name in ["Alice", "Bob"]:
    print(f"Hello {name}")
\end{CacheMeCode}
```

Actually, we also provide this style by default (and explain how to write it yourself), just make sure to load:

```
\usepackage{pythonhighlight}
\usepackage{tcolorbox}
```

You can also cache any environment and command using something like:

- For environments:
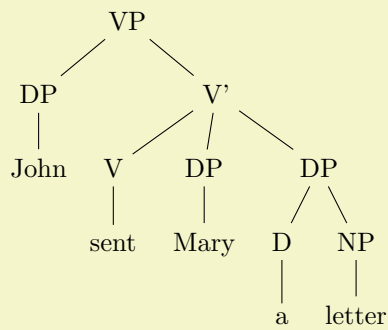
A forest tree automatically cached
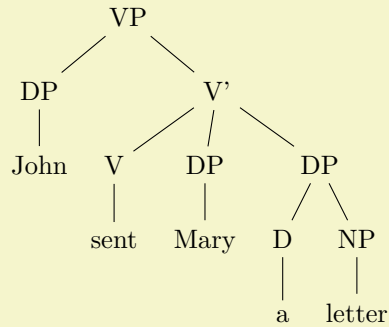
And one not cached:

```
% We cache the environment forest
\cacheEnvironment{forest}{latex, add to preamble={\usepackage{forest}}}

A forest tree automatically cached %
\begin{forest}
  [VP
    [DP[John]]
    [V'
      [V[sent]]
      [DP[Mary]]
      [DP[D[a]][NP[letter]]]
    ]
  ]
\end{forest}\\
And one not cached: %
\begin{forest}<disable externalization>
  [VP
    [DP[John]]
    [V'
      [V[sent]]
      [DP[Mary]]
      [DP[D[a]][NP[letter]]]
    ]
  ]
\end{forest}
```

People interested by `tikz-cd` might like this example:

$$A \longrightarrow B$$

```
\cacheEnvironment{tikzcd}{tikz, add to preamble={\usepackage{tikz-cd}}}
\begin{tikzcd}
  A \rar & B
\end{tikzcd}
```

- For commands:
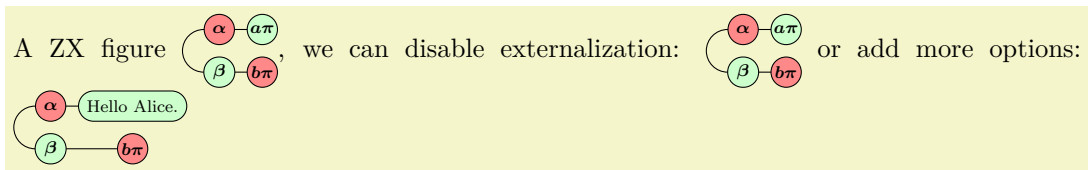
```
% We cache the command \zx
% O{} = optional argument, m = mandatory argument.
% Autodetected if command defined with xparse (NewDocumentCommand)
% (in this example it is redundant since zx IS defined with xparse syntax)
\cacheCommand{zx}[O{}O{}O{}m]{latex, add to preamble={\usepackage{zx-calculus}}}

A ZX figure %
\zx{
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar          & \zxX*{b\pi}
}, we can disable externalization: %
\zx<disable externalization>{
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar          & \zxX*{b\pi}
} or add more options: %
\zx<add to preamble={\usepackage{amsmath}\def\hello#1{Hello #1.}}>{
  \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{\hello{Alice}}} \\
  \zxZ{\beta} \rar          & \zxX*{b\pi}
}
```

You can also cache all tikz pictures by default, except those that are run from some commands (as remember picture does not work yet, this is important). For instance, this code will allow you to freely use the `todonotes` package (that uses internally tikz but is not cachable with this library since it uses remember picture) while caching all other elements:

```
%% this will cache \tikz and tikzpictures by default (using the tikz preset)
\cacheTikz
%% possibly add stuff to the tikz preset
\robExtConfigure{
  add to preset={tikz}{
    add to preamble={
      \def\whereIAm{(I am cached)}
    }
  },
}%
%% We say not to cache the todo commands:
\cacheCommand{todo}[O{}m]{disable externalization}%
\todo{Check how to use cacheCommand and cacheEnvironment}
\tikz[baseline=(A.base)] \node(A)[rounded corners,fill=green]{my cached node \whereIAm};
```

Check how to use cacheCommand and cacheEnvironment

will gives you: my cached node (I am cached)

Note that most options can either be specified anywhere in the document like

```
\robExtConfigure{
  disable externalization
}
```

, inside a preset (that is nothing more than a pgf style), or in the options of the cached environment (either inside `CacheMe` or inside the first parameter `<...>` for automatically cached environments):

```
\robExtConfigure{
  new preset={my preset with xcolor}{
    latex, % load the latex preset, you can also use the tikz preset and more
    add to preamble={
      % load xcolor in the preamble of the cached document
      \usepackage{xcolor}
    },
  },
}
\begin{CacheMe}{my preset with xcolor, add to preamble={\def\hello#1{Hello #1!}} }
  \begin{minipage}{5cm}
    \textcolor{red}{\hello{Alice} See, you can cache any data, including minipages,
      define macros, combine presets, override a single picture or a whole part of
      a document etc.}
  \end{minipage}
\end{CacheMe}
```

(see that `CacheMe` can be used to cache arbitrary pictures)

Or include images generated in python:

```
\begin{CacheMeCode}{python, set includegraphics options={width=.8\linewidth}}
import matplotlib.pyplot as plt
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{CacheMeCode}
```

You can also include files from the root folder, if you want, recompile all pictures if this files changes (nice to ensure maximum purity):

The answer is 42.

```
\begin{CacheMe}{latex,
    add dependencies={common_inputs.tex},
    add to preamble={\input{__ROBEXT_WAY_BACK__common_inputs.tex}}}
  The answer is \myValueDefinedInCommonInputs.
\end{CacheMe}
```

Since v2.1, we also provide easy ways to export counters:

The current page is 6.

```
\begin{tikzpictureC}<forward counter=page>
  \node[rounded corners, fill=red]{The current page is \thepage.};
\end{tikzpictureC}
```

or even macros (we can also export evaluated macros):

Alice   Bob

```
\cacheTikz
\newcommand{\MyNodeWithNewCommand}[2][draw,thick]{\node[rounded corners,fill=red,#1]{#2};}
\begin{tikzpicture}<forward=\MyNodeWithNewCommand>
  \MyNodeWithNewCommand{Alice}
  \MyNodeWithNewCommand[xshift=2cm]{Bob}
\end{tikzpicture}
```

Since manually forwarding elements can be quite verbose, it is also possible to automatically apply some styles depending on the content of the element to cache. For instance, we provide a way to automatically forward some macros:

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyNode}[2][draw,thick]{\node[rounded corners,fill=red,#1]{#2};}
\newcommandAutoForward{\MyGreenNode}[2][draw,thick]{\node[rounded corners,fill=green,#1]{#2};}

\begin{tikzpicture}
  \MyNode{Recompiled only if MyNode is changed}
  \MyNode[xshift=8cm]{but not if the (unused) MyGreenNode is changed.}
\end{tikzpicture}\\

\begin{tikzpicture}
  \MyGreenNode{Recompiled only if MyGreenNode is changed}
  \MyGreenNode[xshift=8cm]{but not if the (unused) MyNode is changed.}
\end{tikzpicture}
```

We also provide a number of other functionalities, among others:

- To achieve a pleasant and configurable interface (e.g. pass options to a preset with default values), we introduced placeholders, that can be of independent interest.

- To get maximum purity while minimizing the number of rebuild, we provide functions to forward macros defined in the parent document, and the set of macros that must be forwarded can be automatically detected for each element to cache. It is also possible to conditionally import a library if the element contains a given macro.

- We provide a way to compile a template via:
  `new compiled preset={compiled ZX}{templateZX, compile latex template}{}`
  in order to make the compilation even faster (but you cannot use presets except `add to preamble`)

- We give a highly configurable template, compilation and inclusion system, to cache basically anything

- We show how to compile all images in parallel using `\robExtConfigure{compile in parallel}` or `\robExtConfigure{compile in parallel after=5}` (make sure to have xargs installed, which is available by default on linux but must be installed on Windows via by installing the lightweight GNU On Windows (Gow) `https://github.com/bmatzelle/gow` if you use the default command) even during the first run, so that the first run becomes faster to compile than a normal run(!)

- We furnish commands to automatically clean the cache while keeping cached elements that are still needed.

- And more…

2 Introduction

2.1 Why do I need to cache (a.k.a. externalize) parts of my document?

One often wants to cache (i.e. store pre-compiled parts of the document, like figures) operations that are long to do: For instance, TikZ is great, but TikZ figures often take time to compile (it can easily take a few seconds per picture). This can become really annoying with documents containing many pictures, as the compilation can take multiple minutes: for instance my thesis needed roughly 30mn to compile as it contains many tiny figures, and LaTeX needs to compile the document multiple times before converging to the final result. But even on much smaller documents you can easily reach a few minutes of compilation, which is not only high to get a

useful feedback in real time, but worse, when using online LATEX providers (e.g. overleaf), this can be a real pain as you are unable to process your document due to timeouts.

Similarly, you might want to cache the result of some codes, for instance a text or an image generated via python and matplotlib, without manually compiling them externally.

2.2 Why not using Ti*k*Z's externalize library?

Ti*k*Z has an externalize library to pre-compile these images on the first run. Even if this library is quite simple to use, it has multiple issues:

- If you add a picture before existing pre-compiled pictures, the pictures that are placed after will be recompiled from scratch. This can be mitigated by manually adding a different prefix to each picture, but it is highly not practical to use.

- To compile each picture, TikZ's externalize library reads the document's preamble and needs to process (quickly) the whole document. In large documents (or in documents relying on many packages), this can result in a significant loading time, sometimes much bigger than the time to compile the document without the externalize library: for instance, if the document takes 10 seconds to be processed, and if you have 200 pictures that take 1s each to be compiled, the first compilation with the TikZ's externalize library will take roughly half an hour ($200 \times 10s = 33mn$) instead of 3mn without the library. And if you add a single picture at the beginning of the document… you need to restart everything from scratch. For these reasons, I was not even able to compile my thesis with TikZ's external library in a reasonable time.

- If two pictures share the same code, it will be compiled twice

- Little purity is enforced: if a macro changes before a pre-compiled picture that uses this macro, the figure will not be updated. This can result in different documents depending on whether the cache is cleared or not.

- As far as I know, it is made for TikZ picture mostly, and is not really made for inserting other stuff, like matplotlib images generated from python etc...

- According to some maintainers of TikZ, "the code of the externalization library is mostly unreadable gibberish[1]", and therefore most of the above issues are unlikely to be solved in a foreseeable future.

2.3 FAQ

What is supported? You can cache most things, including tikz pictures, (including ones with overlays (but not with remember picture), with depth etc.), python code etc. We tried to make the library as customizable as possible to be useful in most scenarios. You can also send some data (e.g. : the current page) to the compiled pictures, and feed some data back to the main document (say that you want to compute a value that takes time to compute, or compute the number of pages of the produced document in order to increase the number of pages accordingly…). Since v2.0, you can also cache automatically any environment.

What is not supported? We do not yet support remember picture, and you can't use (yet) cross-references inside your images (at least not without further hacks) or links as links are stripped when the pdf is included. I might have some ideas to solve this[2], meanwhile you can just disable locally the library on problematic figures. Note that this library is quite young, so expect untested things.

What OS are supported? I tested the library on Linux and Windows, but the library should work on all OS, including MacOs. Please let me know if it fails for you.

---

[1]`https://github.com/pgf-tikz/pgf/issues/758`
[2]`https://tex.stackexchange.com/questions/695277/clickable-includegraphics-for-cross-reference-data`

Do I need to compile using -shell-escape? Since we need to compile the images via an external command, the simpler option is to add the argument `-shell-escape` to let the library run the compilation command automatically (this is also the case of Ti*k*Z's externalize library). Note that overleaf already defines this option by default, so you don't need to do anything special in overleaf. However, people worried by security issues of `-shell-escape` (that allows arbitrary code execution if you don't trust the LaTeX code) might be interested by these facts:

- You can choose to display a dummy content explaining how to manually compile the pictures instead of giving an error until they are compiled, see `enable fallback to manual mode`.

- You can compile manually the images: all the commands that are left to be executed are also listed in `robExt-compile-missing-figures.sh` and you can just inspect and run them, either with `bash robExt-compile-missing-figures.sh` or by typing them manually (most of the time it's only a matter of running `pdflatex somefile.tex` from the `robustExternalize` folder).

- If you allowed:
    - `pdflatex` (needed to compile latex documents)
    - `cd` (not needed when using `no cache folder`)
    - and `mkdir` (not needed when using `no cache folder` or if the cache folder that defaults to `robustExternalize` is already created)

  to run in restricted mode (so without enabling `-shell-escape`), then there is no need for `-shell-escape`. In that case, set `force compilation` and this library will compile even if `-shell-escape` is disabled.

- If images are all already cached, you don't need to enable `-shell-escape` (this might be interesting e.g. to send a pre-cached document to the arxiv or to a publisher: just make sure to include the cache folder).

- We use very few commands to compile latex files, basically only pdflatex, mkdir (to create the cache folder if needed) and cd (if the cache folder is not present). You might want to allow them in restricted mode.

Is it working on overleaf? Yes: overleaf automatically compiles documents with `-shell-escape`, so nothing special needs to be done there (of course, if you use this library to run some code, the programming language might not be available, but I heard that python is installed on overleaf servers for instance, even if this needs to be doubled checked). If the first compilation of the document to cache images times out, you can just repeat this operation multiple times until all images are cached.

Overleaf still times-out, why? During the first compilation, the pictures are not compiled, so you may obtain a time out. However, at each compilation, new pictures should be added to the cache, so you should eventually be able to compile (check the log file, you should read "No need to recompile XX.pdf" if the picture is already in the cache). If you want this to be faster, you can try to enable `\robExtConfigure{compile in parallel}` in order to compile the pictures in parallel (need 2 compilations). If you do not often change all pictures, you can also locally download the project (it might be more practical to use the git bridge if you have it enabled), compile them locally, and copy back the `robustExternalize` folder containing all cached pictures.

If all pictures are still compiled, but you still experience timeouts, you can try to use compiled presets, with something like:

```
\robExtConfigure{
  new compiled preset={compiled ZX}{templateZX, compile latex template}{}
}
```

to create a new, more efficient (we call it compiled) preset called `compiled ZX` from a preset `templateZX`.

You can also try to reduce the compilation time by reducing the number of compilation passes. Indeed, overleaf will run `pdflatex` multiple times to make it converge to the final version, increasing mechanically the compilation time (for instance a file that takes 6s to compile locally was not able to compile online). You can force overleaf to compile a single time the document by creating a `latexmkrc` file at the root of the project, containing:

```
$max_repeat=1
```

Of course, you might need to recompile more than once the document to reach the final version.

Do you have some benchmarks? We completely rewrote the original library to introduce the notion of placeholders (before, the template was fixed forever), allowing for greater flexibility. The original rewrite (1.0) had really poor performance compared to the original library (only x3 improvement compared to no externalization, while the old library could be 20x faster), but we made some changes (v2.0) that correct this issue to recover the original speed (at least when using compiled templates). Here is a benchmark we made on an article with 159 small pictures (ZX diagrams from the zx-calculus library):

- without externalization: 1mn25

- with externalization, v2.0 (first run, not in parallel): 3mn05, so takes twice the time; but (first run in parallel) takes 60s (so faster by a factor of 1.5).

- with externalization, v2.0 (next runs): 5.77s (it is 15x times faster)

- with externalization, v2.0, with a "compiled" template (less flexible) like in the old version: 4.0s (22x faster)

- with externalization, old library (next runs): 4.16s (so 21x faster)

Can I compile pictures in parallel to speed up notably the first run? The first run might be particularly long to compile since many elements are still not cached and re-starting LaTeX on each picture can take quite some time. If you have v2.1 or above, and if you have xargs installed (installed by default on most linux, on Windows you need to install the lightweight GNU On Windows (Gow) `https://github.com/bmatzelle/gow`), you can simply compile all images in parallel by typing at the beginning of the file:

```
\robExtConfigure{
  compile in parallel
}
```

In my benchmark, I observed a compilation 1.5x faster than a normal compilation without any caching involved. We provide more options in section 4.3.10.

Can I use version-control to keep the cached files in my repository? Sure, each cached figure is stored in a few files (typically one pdf and one LaTeX file, plus the source) having the same prefix (the hash), avoiding collision between runs. Just commit these files and you are good to go.

Can I deal with baseline position and overlays ? Yes, the depth of the box is automatically computed and used to include the figure by default, and additional margin is added to the image (30cm by default) to allow overlay text.

How is purity enforced? Purity is the property that if you remove the cached files and re-compile your document, you should end-up with the same output. To enforce purity, we compute the hash of the final program, including the compilation command, the dependency files used for instance in `\input{include.tex}` (unless you prefer not to, for instance to keep parts of the process impure for efficiency reasons), the forwarded macros, and put the code in a file named based on this hash. Then we compile it if it has not been used before, and include the output. Changing a single character in the file, the tracked dependencies, or the compilation command will lead to a new hash, and therefore to a new generated picture.

What if I don't want purity for all files? If you do not want your files to be recompiled if you modify a given file, then just do not add this file to the list of dependencies.

Can I extend it easily? We tried to take a quite modular approach in order to allow easy extensions. Internally, to support a new cache scheme, we only expect a string containing the program (possibly produced using a template), a list of dependencies, a command to compile this program (e.g. producing a pdf and possibly a tex file with the properties (depth…) of the pdf), and a command to load the result of the compilation into the final document (called after loading the previously mentioned optional tex file). Thanks to pgfkeys, it is then possible to create simple pre-made settings to automatically apply when needed.

How does it compare with `https://github.com/sasozivanovic/memoize`? I recently became aware of the great `https://github.com/sasozivanovic/memoize`. While we aim to solve a similar goal, our approaches are quite different. While I focus on purity, and therefore create a different file for each picture, the above project puts all pictures in a single file and compile them all at once to avoid losing time to run the latex command for each picture (this mostly makes a difference for the first compilation). Our understanding of the main differences is the following:

Pros of `https://github.com/sasozivanovic/memoize`:

- The above library might be easier to setup as they do not need to specify the preamble of the file to compile (the preamble of the main file is used), and tikz pictures are automatically memoized (we provide `\cacheTikz` for that, but you still need to specify the preamble once).

- This library allows to compile multiple pictures while loading LaTeX only once, saving the time to start LaTeX for each picture. However, it is not clear that it brings a faster compilation time compared to our library for multiple reasons:

    - While during the first run all pictures are compiled with a single LaTeX instance[3] (and inserted at the beginning of the document), they still need to extract each picture in a separate pdf file to include it at the right place in the document. This is done by default using `pdflatex`, which means that you need to call `pdflatex` once per picture, mitigating the advantage of compiling all pictures at once. Note that they also provide a python script to separate the pictures more efficiently, but this needs further manual interventions.

    - We provide an option `compile in parallel` to compile all images in parallel, and the time is in my benchmark 1.5x faster than a normal compilation without any caching. Using this option, we should therefore compile faster than `memoized` that is not able to run jobs in parallel.

- Even if it does not officially support contexts, context might be automatically transmitted (while we need to do a bit of work to manually or semi-automatically define the context needed to compile a picture). However, this can break purity (see below).

Cons of `https://github.com/sasozivanovic/memoize`:

---

[3]We could also do something similar using some scripting to merge multiple cached files together, but this is not yet implemented.

- The purity is not enforced so strongly since all images are in the same file. Notably, the hash only depends on the picture, but not on its context. So for instance if you define `\def\mycolor{blue}` before the picture, and use `\mycolor` inside the picture, if you change later the color to, say, `\def\mycolor{red}`, the picture will not be recompiled (so cleaning the cache and recompiling would produce a different result). In our case, the purity is always strictly enforced (unless you choose not to) since the context must be passed (explicitly or semi-automatically) to the compiled file.

- As a result, the above library has poor support for contexts (in our case, you can easily, for instance, make a picture depend on the current page, and recompile the picture only if the current page changes: you can also do it the other way, and change some counters, say, depending on the cached file).

- The above library only focuses on LaTeX while our library works for any language.

- The above library can only cache in pdf format, while we can generate any format (text, tex, jpg... and even videos that I include in my beamer presentations... but it is another topic!).

- We have an arguably more complete documentation.

Note that `remember picture` is not supported in both libraries, but we can provide multiple methods to disable externalizations when `remember picture` is used.

3 Tutorial

3.1 Installation

To install the library, just copy the `robust-externalize.sty` file into the root of the project. Then, load the library using:

```
\usepackage{robust-externalize}
```

If you want to display a message in the pdf on how to manually compile the file if `-shell-escape` is disabled, you can also load this configuration (only available on v2.0):

```
\robExtConfigure{
  enable fallback to manual mode,
}
```

(otherwise, it will give you an error if `-shell-escape` is disabled and if some pictures are not yet cached)

If you forget/do not want to enable `-shell-escape`, this will give you this kind of message with instructions to follow to compile without shell escape:

> **Manual mode: either compile with -shell-escape or compile:**
> `robustExternalize/robExt-9A58BBFD5C2B6C079891CF68E406829D.tex`
> **via**
> `cd robustExternalize/ && pdflatex -halt-on-error "robExt-9A58BBFD5C2B6C079891CF68E406829D.tex"`
> **or call**
> `bash robust-externalize-robExt-compile-missing-figures.sh`
> **to compile all missing figures.**

If you use an editor, you can usually configure `-shell-escape` directly inside the editor, even on a per-document basis. Shell-escape is enabled by default on overleaf. In emacs/auctex, just add at the end of your document:

```
% Local Variables:
% TeX-command-extra-options: "--shell-escape"
% End:
```

(Note that it seems that `--shell-escape` can be used with one or two dashes on Linux, I am not sure about the number of dashes needed in Windows.)

For TeXstudio, you can enable shell-escape on the current document by adding in your document:

```
% !TeX TXS-program:compile = txs:///pdflatex/[--shell-escape]
```

If you use `latexmk` (with any editor), you can create a file called `.latexmkrc` containing:

```
$pdf_mode = 1;
$pdflatex = 'pdflatex --shell-escape';
```

For other editors, this answer[4] describe pretty accurately the solutions you have depending on your software.

3.2 Caching a tikz picture

If you only care about Ti*k*Z's picture, you have 3 options:

1. Call once `\cacheTikz` that will redefine `tikzpicture` and `\tikz` to use our library (if you use this solution, make sure to read how to disable externalization (section 4.3.6) as we do not support for instance `remember picture`). Then, configure the default preamble for cached files as explained below by extending the `tikz` or `tikzpicture` presets (that first loads `tikz`).

2. Use the generic commands that allows you to wrap an arbitrary environment:

   ```
   %  name environment ---v          v----- preset options
   \cacheEnvironment{tikzpicture}{tikzpicture}
   ```

   Note that you can also wrap commands (that you call with `\foo` instead of `\begin{foo}...\end{foo}`) using this:
   `\cacheCommand{yourcommand}[O{default val}m]{your preset options}`
   (`O{default val}m` means that the command accepts one optional argument with default value `default val` and one mandatory argument) but **we do recommend** to use `\cacheCommand` to cache `\tikz` since `\tikz` has a quite complicated parsing strategy (e.g. you can write `\tikz[options] \node{foo};` which has no mandatory argument enclosed in `{}`). `\cacheTikz` takes care of this already and caches both the environment `\begin{tikzpicture}` and the macro `\tikz`.

3. Use `tikzpictureC` instead of `tikzpicture` (this is mostly done to easily convert existing code to this library, but works only for `tikz` pictures).

4. Use the more general `CacheMe` environment, that can cache Ti*k*Z, LATEX, python, and much more.

These 4 options are illustrated below (note that all commands accept a first optional argument enclosed in `<...>` that contains the options to pass to `CacheMe` after loading the `tikzpicture` preset, that loads itself the `tikz` preset first):

Option 1:

I am a cached picture: Hello World! . I am a non-cached picture: Hello World! .

---

[4]`https://tex.stackexchange.com/questions/598818/how-can-i-enable-shell-escape`
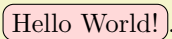
```
%% We override the default tikzpicture environment
%% to externalize all pictures
%% Warning: it will cause troubles with pictures relying on |remember pictures|
\cacheTikz

I am a cached picture: \begin{tikzpicture}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{tikzpicture}. %

I am a non-cached picture: \begin{tikzpicture}<disable externalization>[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{tikzpicture}.
```

Option 2:

I am a cached picture: Hello World!.

```
\cacheEnvironment{tikzpicture}{tikzpicture}
I am a cached picture: \begin{tikzpicture}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{tikzpicture}.
```

Option 3:

I am a cached picture: Hello World!.

```
I am a cached picture: \begin{tikzpictureC}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{tikzpictureC}.
```
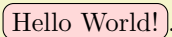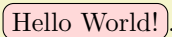
Option 4:

I am a cached picture: Hello World!.

```
I am a cached picture: \begin{CacheMe}{tikzpicture}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{CacheMe}.
```

Since CacheMe is more general as it applies also to non-tikz pictures (just replace tikzpicture with the style of your choice), we will mostly use this syntax from now.

3.3 Custom preamble

Note that the pictures are compiled in a separate document, with a different preamble and class (we use the standalone class). This is interesting to reduce the compilation time of each picture (loading a large preamble is really time consuming) and to avoid unnecessary recompilation (do you want to recompile all your pictures when you add a single new macro?) without sacrificing the purity. But of course, you need to provide the preamble of the pictures. The easiest way is probably to modify the tikz preset (since tikzpicture loads this style first, it will also apply to tikzpictures. Note that you can also modify the latex preset if you want the change to apply to all LaTeX documents):

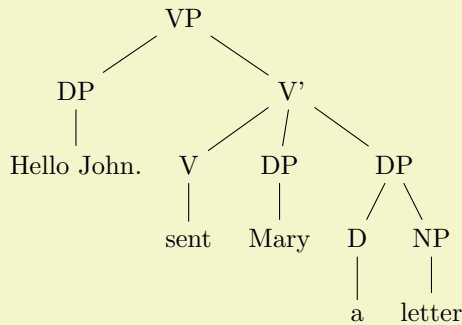See, tikz's style now packs the shadows library by default:
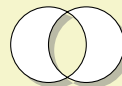
```
\robExtConfigure{
  add to preset={tikz}{
    add to preamble={\usetikzlibrary{shadows}},
  },
}

See, tikz's style now packs the |shadows| library by default: %
\begin{CacheMe}{tikzpicture}[even odd rule]
  \filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{CacheMe}
```

and you can also create a new preset, and why not mix multiple presets together:


```
\robExtConfigure{
  new preset={load forest}{
    latex,
    add to preamble={\usepackage{forest}},
  },
  new preset={load hello}{
    add to preamble={\def\hello#1{Hello #1.}},
  },
}

\begin{CacheMe}{load forest, load hello}
  \begin{forest}
    [VP
      [DP[\hello{John}]]
      [V'
        [V[sent]]
        [DP[Mary]]
        [DP[D[a]][NP[letter]]]
      ]
    ]
  \end{forest}
\end{CacheMe}
```

Note: the add to preset and new preset directives have been added in v2.0, together with the tikzpicture preset. In v1.0, you would do tikz/.append style={...} (and you can still do this if you prefer), the difference is that .append style and .style require the user to double all hashes like \def\mymacro##1{Hello ##1.} which can lead to confusing errors.

You can also choose to overwrite the preset options for a single picture (or even a block of picture if you run the \robExtConfigure and CacheMe inside a group { ... }):

See, you can add something to the preamble of a single picture:
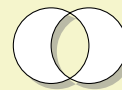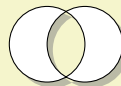
```
See, you can add something to the preamble of a single picture: %
\begin{CacheMe}{tikzpicture, add to preamble={\usetikzlibrary{shadows}}}[even odd rule]
  \filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{CacheMe}
```

Note that if you use the tikzpictureC or tikzpicture syntax, you want to add the options

right after the name of the command or environment, enclosed in `<>` (by default):

See, you can add something to the preamble of a single picture:

```
See, you can add something to the preamble of a single picture: %
\begin{tikzpictureC}<add to preamble={\usetikzlibrary{shadows}}>[even odd rule]
  \filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{tikzpictureC}
```

Important: Note that the preset options can be specified in a number of places: in `\robExtConfigure` (possibly in a group, or inside a preset), in the options of the picture, in the default options of `cacheEnvironment` and `cacheCommand` etc. Most options can be used in all these places, the only difference being the scope of the options.

If you find yourself duplicating code between the main document and the preamble of the cached file, we provide multiple solutions with different properties:

- Recommended: `\newcommandAutoForward` etc, to automatically forward specific macros/colors etc: see section 3.7. Very useful if you want maximum purity without annoying recompilation.

- `\runHereAndInPreambleOfCachedFiles`: mostly useful if you want to always add a text both in the current file and in the cached files, like a font etc.

- Share a common input file with or without dependencies (see section 3.4), useful if you need to work with external files or if you do NOT want purity (i.e. you can choose not to recompile the file even if its content changes).

`\runHereAndInPreambleOfCachedFiles{XXX}` is very simple (need at least version 2.3), as it is simply a shortcut for typing `XXX` in the current file and in the preamble of the cached files (default to latex preset). For instance if you want the current file and all cached files to use the Times New Roman font, you can do:

```
% By default adds
\runHereAndInPreambleOfCachedFiles{
  \usepackage{fontspec}
  % need lualatex or xelatex to compile
  \setmainfont{Times New Roman}
}

\robExtConfigure{
  add to preset={latex}{
    use lualatex, % Make sure to compile with lualatex the cached images.
  },
}
```

3.4 Dependencies

Note: dependencies can sometimes be advantageously replaced with auto-forwarding of arguments, cf. section 3.7, or .

It might be handy to have a file that is loaded in both the main document and in the cached pictures. For instance, if you have a logo that might be included in the cached files, or if you have a file `common_inputs.tex` that you want to input in both the main file and in the cached files, that contains, say:

`\def\myValueDefinedInCommonInputs{42}`

then you can add it as a dependency this way (here we use the `latex` preset that does not wrap the code inside a `tikzpicture` only to illustrate that we can also cache things that are not generated by tikz):

The answer is 42.

```
\begin{CacheMe}{latex,
    add dependencies={common_inputs.tex},
    add to preamble={\input{__ROBEXT_WAY_BACK__common_inputs.tex}}}
  The answer is \myValueDefinedInCommonInputs.
\end{CacheMe}
```

Note that the placeholder `__ROBEXT_WAY_BACK__` contains the path from the cache folder (containing the `.tex` that will be cached) to the root folder, and will be replaced when creating the file. This way, you can easily input files contained in the root folder. You can also create your own placeholders, read more below.

You can note that we used `add dependencies={common_inputs.tex}`: this allows us to recompile the files if `common_inputs.tex` changes. If you do not want this behavior (e.g. `common_inputs.tex` changes too often and you do not want to recompile everything at every change), you can remove this line, but beware: if you do a breaking changes in `common_inputs.tex` (e.g. redefine 42 to 43), then the previously cached picture will not be recompiled! (So you will still read 42 instead of 43.)

3.5   Wrap arbitrary environments

You can wrap arbitrary environments using the already presented `cacheEnvironment` and `cacheCommand`, where the first mandatory argument is the name of the environment/macro, and the second mandatory argument contains the default preset. `cacheEnvironment` works for any environment while `cacheCommand` might need a bit of help to determine the signature of the macro if the function is defined via xparse. Long story short, `O{foo}` is an optional argument with default value foo, `m` is a mandatory argument. By default, you can pass an optional arguments via the first argument enclosed in `<>`:
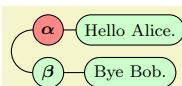

```
\cacheCommand{zx}[O{}O{}O{}m]{latex, add to preamble={\usepackage{zx-
calculus}\def\hello#1{Hello #1.}}}
  \zx<add to preamble={\usepackage{amsmath}\def\bye#1{Bye #1.}}>[mbr=1]{ % amsmath provides \text
    \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{\hello{Alice}}} \\
    \zxZ{\beta} \rar            & \zxZ{\text{\bye{Bob}}}
  }
```

you can also disable externalization for some commands using this same command, here is for instance the code to use `todonotes`:

```
\cacheCommand{todo}[O{}m]{disable externalization}
\todo{Check how to use cacheCommand and cacheEnvironment}
```

Which gives you

See section 4.7 for more details.

Check how to use cacheCommand and cacheEnvironment

3.6   Disabling externalization

You can use `disable externalization` to disable externalization (which is particularly practical if you set `\cacheTikz`). You can configure the exact command run in that case using `command if no externalization/.code={...}`, but most of the time it should work out of the box (see section 4.3.6 for details).

17
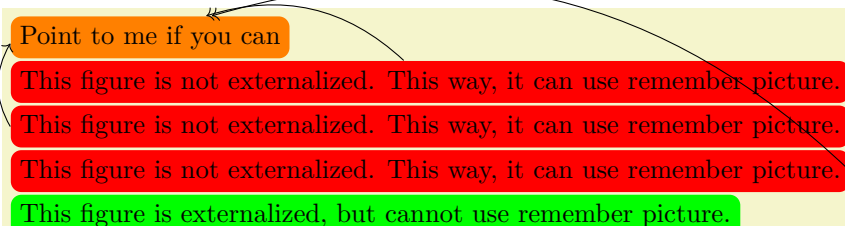
```
% In theory all pictures should be externalized (so remember picture should fail)
\tikz[remember picture,baseline=(pointtome1.base)]
  \node[rounded corners, fill=orange](pointtome1){Point to me if you can};\\
\cacheTikz
% But we can disable it temporarily
\begin{tikzpicture}<disable externalization>[remember picture]
  \node[rounded corners, fill=red](A){This figure is not externalized.
    This way, it can use remember picture.};
  \draw[->,overlay] (A) to[bend right] (pointtome1);
\end{tikzpicture}\\

% You can also disable it globally/in a group:
{
  \robExtConfigure{disable externalization}

  \begin{tikzpicture}[remember picture]
    \node[rounded corners, fill=red](A){This figure is not externalized.
      This way, it can use remember picture.};
    \draw[->,overlay] (A.west) to[bend left] (pointtome1);
  \end{tikzpicture}\\

  \begin{tikzpicture}[remember picture]
    \node[rounded corners, fill=red](A){This figure is not externalized.
      This way, it can use remember picture.};
    \draw[->,overlay] (A.east) to[bend right] (pointtome1);
  \end{tikzpicture}\\
}

\begin{tikzpicture}
  \node[rounded corners, fill=green](A){This figure is externalized, but cannot use remember picture.};
\end{tikzpicture}
```

You can also disable externalization for some kinds of commands. For instance, the package `todonotes` requires `remember picture` and is therefore not compatible with externalization provided by this package. To disable externalization on all `\todo`, you can do:

```
\cacheCommand{todo}[O{}m]{disable externalization}
\todo{Check how to use cacheCommand and cacheEnvironment}
```

Which gives you

You can also disable externalization on elements that contain a specific string/regex. For instance, you can disable externalization on all elements containing `remember picture`:

Point to me Some text

```
\cacheTikz
\robExtConfigure{
  add to preset={tikz}{
    if matches={remember picture}{disable externalization},
  },
}
\begin{tikzpicture}[remember picture]
  \node[fill=green](my node){Point to me};
\end{tikzpicture} %
Some text %
\begin{tikzpicture}[overlay, remember picture]
  \draw[->] (0,0) to[bend left] (my node);
\end{tikzpicture}
```

3.7 Feeding data from the main document to the cached documents

You can feed data from the main document to the cached file using placeholders, since `set placeholder eval={__foo__}{\bar}` will evaluate `\bar` and put the result in `__foo__`. For instance, if the picture depends on the current page, you can do:

> The current page is 18.

```
\begin{tikzpictureC}<set placeholder eval={__thepage__}{\thepage}>
  \node[rounded corners, fill=red]{The current page is __thepage__.};
\end{tikzpictureC}
```

However, since v2.1, we also provide a method to specifically export a counter using `forward counter=my counter` and a macro using `forward=\macroToExport`, possibly by evaluating first using `forward eval=\macroToEval`:

> Alice   Bob

```
\cacheTikz
\NewDocumentCommand{\MyNode}{O{}m}{\node[rounded corners,fill=red,#1]{#2};}
\begin{tikzpicture}<forward=\MyNode>
  \MyNode{Alice}
  \MyNode[xshift=2cm]{Bob}
\end{tikzpicture}
```

To avoid manually writing `forward=...` for each picture, we can instead load `auto forward` and define the macro for instance using `\newcommandAutoForward` (we also provide alternatives for xparse and def-based definitions):

> Recompiled only if MyNode is changed   but not if the (unused) MyGreenNode is changed.
> Recompiled only if MyGreenNode is changed   but not if the (unused) MyNode is changed.

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyNode}[2][draw,thick]{\node[rounded corners,fill=red,#1]{#2};}
\newcommandAutoForward{\MyGreenNode}[2][draw,thick]{\node[rounded corners,fill=green,#1]{#2};}

\begin{tikzpicture}
  \MyNode{Recompiled only if MyNode is changed}
  \MyNode[xshift=8cm]{but not if the (unused) MyGreenNode is changed.}
\end{tikzpicture}\\

\begin{tikzpicture}
  \MyGreenNode{Recompiled only if MyGreenNode is changed}
  \MyGreenNode[xshift=8cm]{but not if the (unused) MyNode is changed.}
\end{tikzpicture}
```

If a macro depends on another macro/package, it is possible to load any additional style like using the last optional argument of `\newcommandAutoForward`:

> ❤ Alice: Recompiled if MyNode or MyName are changed

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyName}{Alice}
\newcommandAutoForward{\MyNode}[2][draw,thick]{
  \node[rounded corners,fill=red,#1]{\ding{164} \MyName: #2};
}[forward=\MyName, add to preamble={\usepackage{pifont}}]

\begin{tikzpicture}
  \MyNode{Recompiled if MyNode or MyName are changed}
\end{tikzpicture}
```

If instead of forwarding the macro you want to, say, load a package or any other style, you should use instead `\robExtConfigIfMacroPresent`:

19

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\robExtConfigIfMacroPresent{\ding}{add to preamble={\usepackage{pifont}}}

\begin{tikzpicture}
  \node[fill=green, circle]{\ding{164}};
\end{tikzpicture}
```

For more details and functions, see section 4.3.8.

3.8 Defining macros

You can define macros in the preamble:

Hello my friend.

```
\cacheMe[latex, add to preamble={\def\sayhello#1{Hello #1.}}]{
  \sayhello{my friend}
}
```

3.9 Feeding data back into the main document

For more advanced usage, you might want to compute a data and cache the result in a macro that you could use later. This is possible if you write into the file `\jobname-out.tex` during the compilation of the cached file (by default, we already open `\writeRobExt` to write to this file). This file will be automatically loaded before loading the pdf (but you can customize all these operations, for instance if you do not want to load the pdf at all; the only requirement is that you should generate a `.pdf` file to specify that the compilation is finished).

For instance:

We computed the cached value 1.61803.

```
\begin{CacheMe}{latex, add to preamble={\usepackage{tikz}}, do not include pdf}
We compute this data that is long to compute:
\pgfmathparse{(1 + sqrt(5))/2}% result is stored in \pgfmathresult
% We write the result to the -out file (\string\foo writes \foo to the file without evaluating it,
% so this will write "\gdef\myLongResult{1.61803}"):
% Note that CacheMe is evaluated in a group, so you want to use \gdef to define it
% outside of the group
\immediate\write\writeRobExt{%
  \string\gdef\string\myLongResult{\pgfmathresult}%
}
\end{CacheMe}

We computed the cached value \myLongResult.
```

3.10 A note on font handling

Cached pictures are basically simple latex files that are compiled in a separate `standalone` document. Therefore, if the main document loads a different font, there is no reason for the cached document to also use that font. Fully-automatically forwarding the current font by default is not so easy because there are so many ways to load a font in LaTeX (by loading a package like `\usepackage{times}`, by using xelatex or lualatex, by calling locally `\rmfamily`, by loading a different class… and here I'm not even mentioning math font where each symbol can use a different font[5], and this is even more complicated if the main file is compiled with a different engine (e.g.

---

[5]https://tex.stackexchange.com/questions/603475/is-there-any-command-to-find-out-the-current-math-font-used)

what should I do if the main document is compiled with lualatex but the inner cached images are compiled with pdflatex, that supports very few fonts by default?). For these reasons, it is simpler to simply let the user load fonts as they want, for instance using something like:

See I am using times font

```
\robExtConfigure{
  add to preset={latex}{
    add to preamble={\usepackage{times}},
  },
}

\begin{tikzpictureC}
\node[fill=green]{See I am using times font};
\end{tikzpictureC}
```

or:

See I am using a 12pt font

```
\robExtConfigure{
  add to preset={latex}{
    add to latex options={12pt},
  },
}

\begin{tikzpictureC}
\node[fill=green]{See I am using a 12pt font};
\end{tikzpictureC}
```

But if you want to use the same font as the current document, you might want to avoid code duplication (this might also be useful for any arbitrary code). Since v2.3, you can do:

```
\runHereAndInPreambleOfCachedFiles{
  \usepackage{fontspec}
  % need lualatex or xelatex to compile
  \setmainfont{Times New Roman}
}

\robExtConfigure{
  add to preset={latex}{
    use lualatex, % Make sure to compile with lualatex the cached images.
  },
}
```

and it will automatically compile the cached images with lualatex, and configure the font of the current document and of the cached documents to follow `Times New Roman`.

If you like to often change the font in the document, you can also do something like that to always compile the document with the font currently in use if you use lualatex (this code fails with xelatex[6], currently trying to understand why):

---

[6]https://tex.stackexchange.com/questions/705208/get-current-font

```latex
\usepackage{fontspec}
\setmainfont{Times New Roman}

%% needed since \f@family contains a @ in its name:
\makeatletter
\def\myCurrentFont{\f@family}
\makeatother

\robExtConfigure{
  add to preset={latex}{
    use lualatex,
    % create a new placeholder containing the name of the current font:
    set placeholder eval={__MY_FONT__}{\myCurrentFont},
    add to preamble={
      \usepackage{fontspec}
      \setmainfont{__MY_FONT__}
    },
  },
}
```

If you want this to work on xelatex, you can either manually attribute a `NFSSFamily` name like:

```latex
\runHereAndInPreambleOfCachedFiles{
  \usepackage{fontspec}
  % need lualatex or xelatex to compile, make sure to use a different NFSSFamily for each new font you add here
  \setmainfont[NFSSFamily=tmsnr]{Times New Roman}
}
```

and use the above solution for lualatex (you can change the font with `\fontfamily{tmsnr}\selectfont`), or do instead something like this:

```latex
% Create a new command \mysetmainfont which forwards the font to the picture.
\NewDocumentCommand{\mysetmainfont}{m}{%
  \def\myCurrentFont{#1}% Stores the name of the current font in \myCurrentFont
  \setmainfont{#1}%
}

% Use mysetmainfont from now to change font:
\mysetmainfont{Times New Roman}

\usepackage{robust-externalize}
\robExtConfigure{
  add to preset={latex}{
    use xelatex,
    set placeholder eval={__MY_FONT__}{\myCurrentFont},
    add to preamble={
      \usepackage{fontspec}
      \setmainfont{__MY_FONT__}
    },
  },
}
```

3.11 Compile in parallel

Make sure to have xargs installed (installed by default on most linux, on Windows you need to install the lightweight GNU On Windows (Gow) https://github.com/bmatzelle/gow), and type in your preamble:

```latex
\robExtConfigure{
  compile in parallel
}
```

to compile all figures in parallel (you need to compile your document twice). Your document should build significantly faster, possibly faster than a normal run. If you want to compile in parallel only if you have more than, say, 5 new elements to cache, do:

```latex
\robExtConfigure{
  compile in parallel after=5
}
```

3.12 Compile a preset

Long story short: you can compile even faster (around 1.5x in our tests) by compiling presets, but beware that you will not be able to modify the placeholders except `add to preamble` with the default compiler we provide:

> **HeyBob!**

```
% We create a latex-based preset and compile it
\robExtConfigure{
  new preset={templateZX}{
    latex,
    add to preamble={
      \usepackage{tikz}
      \usepackage{tikz-cd}
      \usepackage{zx-calculus}
    },
    %% possibly add some dependencies
  },
  % We compile it into a new preset
  new compiled preset={compiled ZX}{templateZX, compile latex template}{},
}

% we use that preset automatically for ZX environments
\cacheEnvironment{ZX}{compiled ZX}
\cacheCommand{zx}{compiled ZX}

% Usage: (you can't use placeholders except for the preamble, trade-off of the compiled template)
\begin{ZX}<add to preamble={\def\sayHey#1{Hey #1!}}>
  \zxX{\sayHey{Bob}}
\end{ZX}
```

For details, see section 4.3.11.

3.13 For non-LaTeX code

Due to the way LaTeX works, non-LaTeX code can't be reliably read inside macros and some environments that parse their body (e.g. align) as some characters are removed (e.g. percent symbols are comments and are removed). For this reason, we sometimes need to separate the time where we define the code and where we insert it (this is done using placeholders, see `PlaceholderFromCode`), and we need to introduce new environments to populate the template (see section 4.5 for more details, to generate them from filename, to get the path of a file etc).

The environment `CacheMeCode` can be used for this purpose.

3.13.1 Python code

Generate an image   For instance, you can use the default `python` template to generate an image with python. The following code:

```
\begin{CacheMeCode}{python, set includegraphics options={width=.8\linewidth}}
import matplotlib.pyplot as plt
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{CacheMeCode}
```

will produce the image visible in fig. 1. **Importantly: you do not want to indent the content of CacheMeCode, or the space will also appear in the final code.**

Compute a value   We also provide by default a number of helper functions. For instance, `write_to_out(text)` will write `text` to the `*-out.tex` file that is loaded automatically by LaTeX.
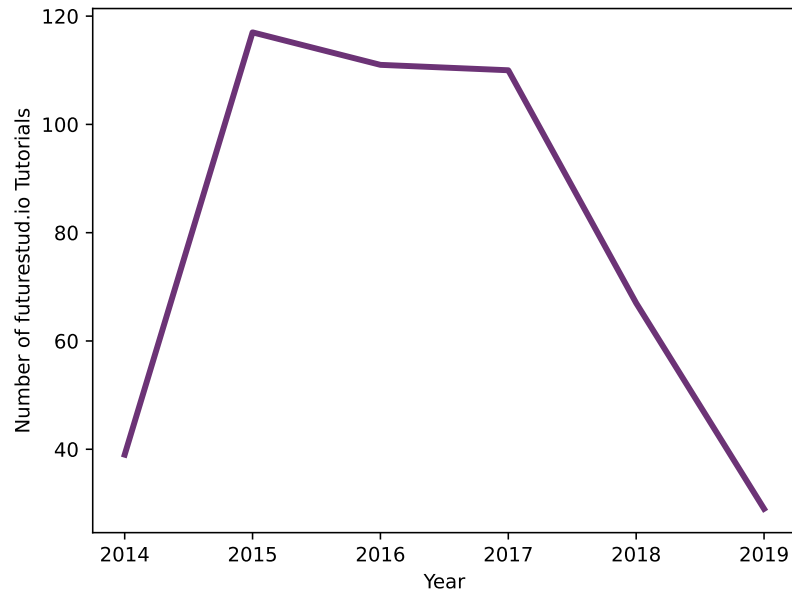
Figure 1: Image generated with python.

This is useful to compute data that is not an image (note that `r"some string"` does not consider backslash as an escape string, which is handy to write LaTeX code in python):

For instance:

$\rightarrow$ The cosinus of 1 is 0.5403023058681398.

```
\begin{CacheMeCode}{python, do not include pdf}
import math
write_to_out(r"\gdef\cosComputedInPython{" + str(math.cos(1)) + r"}")
\end{CacheMeCode}

$\rightarrow$ The cosinus of 1 is \cosComputedInPython.
```

Improve an existing preset If you often use the same code (e.g. load matplotlib, save the file etc), you can directly modify the `__ROBEXT_MAIN_CONTENT__` placeholder to add the redundant information (or create a new template from scratch, see below). By default, it points to `__ROBEXT_MAIN_CONTENT_ORIG__` that contains directly the code typed by the user (this is true for all presets, as `CacheMe*` is in charge of setting this placeholder). When dealing with LaTeX code, `__ROBEXT_MAIN_CONTENT__` should ideally contain a code that can be inserted as-it in the document in order to be compatible by default with `disable externalization`. So if you want to wrap the content of the user in an environment like `\begin{tikzpicture}...\end{tikzpicture}`, this is the placeholder to modify.
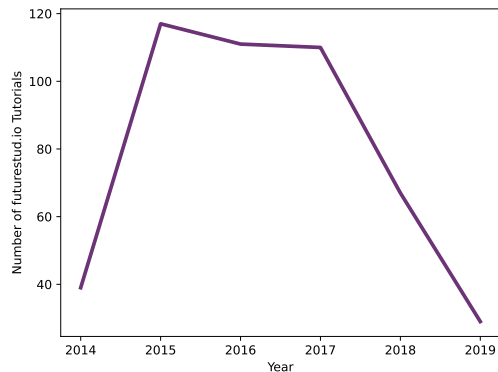
```
%% Create your style:
\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE_BEFORE__}
import matplotlib.pyplot as plt
import sys
\end{PlaceholderFromCode}

\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE_AFTER__}
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{PlaceholderFromCode}

\robExtConfigure{
  new preset={my matplotlib}{
    python,
    add before placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_MATPLOTLIB_TEMPLATE_BEFORE__},
    add to placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_MATPLOTLIB_TEMPLATE_AFTER__},
  },
}

%% Use your style:
%% See, you don't need to load matplotlib or save the file:
\begin{CacheMeCode}{my matplotlib, set includegraphics options={width=.5\linewidth}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{CacheMeCode}
```

Custom parameters and placeholders Let us say that you would like to define a default font size for your figure, but that you would like to allow the user to change this font size. Then, you should create a new placeholder with your default value, and use `set placeholder` to change this value later (see also the documentation of `CacheMeCode` to see how to create a new command to avoid typing `set placeholder`):

Default font size:
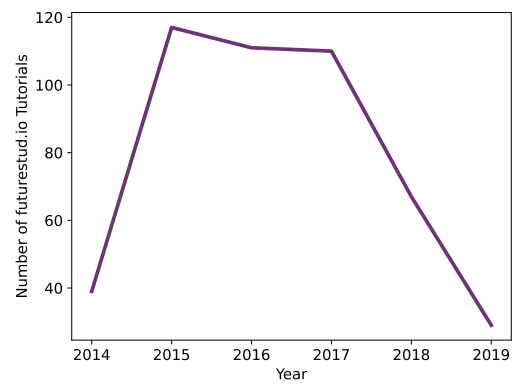
With font size 16:
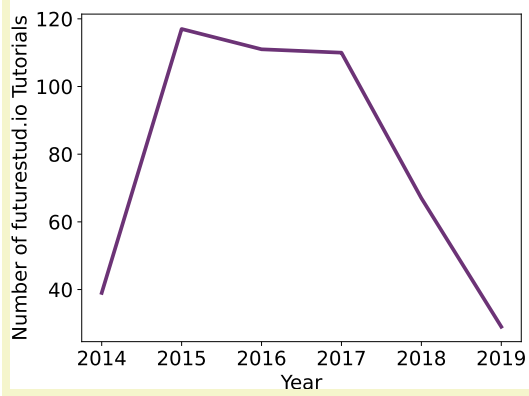
```
%% Create your style:

\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE_BEFORE__}
import matplotlib as mpl
import matplotlib.pyplot as plt
import sys
mpl.rcParams['font.size'] = __MY_MATPLOTLIB_FONT_SIZE__
\end{PlaceholderFromCode}

\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE_AFTER__}
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{PlaceholderFromCode}

\robExtConfigure{
  new preset={my matplotlib}{
    python,
    % We create a new placeholder (it is simple enough that you don't need to use PlaceholderFromCode)
    set placeholder={__MY_MATPLOTLIB_FONT_SIZE__}{12},
    add before placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_MATPLOTLIB_TEMPLATE_BEFORE__},
    add to placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_MATPLOTLIB_TEMPLATE_AFTER__},
  },
}

%% Use your style:
%% See, you don't need to load matplotlib or save the file:
Default font size: \begin{CacheMeCode}{my matplotlib, set includegraphics options={width=.5\linewidth}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{CacheMeCode}

With font size 16:
\begin{CacheMeCode}{my matplotlib,
    set includegraphics options={width=.5\linewidth},
    set placeholder={__MY_MATPLOTLIB_FONT_SIZE__}{16}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{CacheMeCode}
```

Note that if you manage to move all the code in the template and that the user can configure everything using the options and an empty content, you can use `CacheMeNoContent` that takes no argument and that consider its body as the options.

Custom include command There may be some cases where you do not want to include a picture. We already saw the option `do not include pdf` if you do not want to include anything. But you can customize the include function, using notably:
`custom include command={your include command}`
For instance, let us say that you would like to display both the source code used to obtain a given code, together with the output of this code. Then, you can write this style:

```
{
%% Create your style:
\begin{PlaceholderFromCode}{__MY_PRINT_BOTH_TEMPLATE_BEFORE__}
# File where print("bla") should be redirected
# get_filename_from_extension("-foo.txt") will give you the path of the file
# in the cache that looks like robExt-somehash-foo.txt
print_file = open(get_filename_from_extension("-print.txt"),  "w")
sys.stdout = print_file
# This code will read the current code, and extract the lines between
# that starts with "### CODESTARTSHERE" and "### CODESTOPSHERE", and will write
# it into the *-code.text (we do not want to print all these functions in
# the final code)
with open(get_filename_from_extension("-code.txt"), "w") as f:
    # The current script has extension .tex
    with open(get_current_script(), "r") as script:
        should_write = False
        for line in script:
            if line.startswith("### CODESTARTSHERE"):
                should_write = True
            elif line.startswith("### CODESTOPSHERE"):
                should_write = False
            elif "HIDEME" in line:
                pass
            else:
                if should_write:
                    f.write(line)
### CODESTARTSHERE
\end{PlaceholderFromCode}


\begin{PlaceholderFromCode}{__MY_PRINT_BOTH_TEMPLATE_AFTER__}
### CODESTOPSHERE
print_file.close()
\end{PlaceholderFromCode}

\robExtConfigure{
  new preset={my python print both}{
    python,
    add before placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_PRINT_BOTH_TEMPLATE_BEFORE__},
    add to placeholder no space={__ROBEXT_MAIN_CONTENT__}{__MY_PRINT_BOTH_TEMPLATE_AFTER__},
    set title/.style={
      set placeholder={__MY_TITLE__}{#1},
    },
    set title={Example},
    custom include command={
      % Useful to replace __MY_TITLE__:
      \evalPlaceholder{
        \begin{tcolorbox}[colback=red!5!white,colframe=red!75!black,title=__MY_TITLE__]
          \lstinputlisting[frame=single,
            breakindent=.5\textwidth,
            frame=single,
            breaklines=true,
            style=mypython]{\robExtAddCachePathAndName{\robExtFinalHash-code.txt}}
          Output:
          \verbatiminput{\robExtAddCachePathAndName{\robExtFinalHash-print.txt}}
        \end{tcolorbox}
      }
    },
  },
}
```

Once the style is defined (actually we already defined in the library under the name `python print code and result`), you can just write:

```
\begin{CacheMeCode}{my python print both, set title={The for loop}}
for name in ["Alice", "Bob"]:
    print(f"Hello {name}")
\end{CacheMeCode}
```

to get:

> **The for loop**
>
> ```python
> for name in ["Alice", "Bob"]:
>     print(f"Hello {name}")
> ```
>
> Output:
>
> ```
> Hello Alice
> Hello Bob
> ```

3.13.2 Other languages

We also provide support for other languages, notably `bash`, but it is relatively easy to add basic support for any new language. You only need to configure `set compilation command` to your command, `set template` to the file to compile (`__ROBEXT_MAIN_CONTENT__` contains the code typed by the user), and possibly a custom include command with `custom include command` if you do not want to do `\includegraphics` on the final pdf. For instance, to define a basic template for bash, you just need to use:

```
Linux 5.15.90 #1-NixOS SMP Tue Jan 24 06:22:49 UTC 2023
```

```latex
% Create your style
\begin{PlaceholderFromCode}{__MY_BASH_TEMPLATE__}
# Quit if there is an error
set -e
__ROBEXT_MAIN_CONTENT__
# Create the pdf file to certify that no compilation error occured
touch "__ROBEXT_OUTPUT_PDF__"
\end{PlaceholderFromCode}

\robExtConfigure{
  new preset={my bash}{
    set compilation command={bash "__ROBEXT_SOURCE_FILE__"},
    set template={__MY_BASH_TEMPLATE__},
    %%% Version 1:
    % verbatim output,
    %%% Version 2:
    custom include command={%
      \evalPlaceholder{%
        \verbatiminput{__ROBEXT_CACHE_FOLDER____ROBEXT_OUTPUT_PREFIX__-out.txt}%
      }%
    },
    % Ensure that the code does not break when externalization is disabled
    print verbatim if no externalization,
  }
}

% Use your style
\begin{CacheMeCode}{my bash}
# Write the system conf to a file *-out.txt
uname -srv > "__ROBEXT_OUTPUT_PREFIX__-out.txt"
\end{CacheMeCode}
```

Code inside a macro Due to fundamental LaTeX restrictions, it is impossible to use `CacheMeCode` inside a macro or some environments as LaTeX will strip all lines containing a percent character for instance. The solution here is to define our main content before, and then set it using `set main content` (that simply sets `__ROBEXT_MAIN_CONTENT_ORIG__`). In this example, we also show how `CacheMeNoContent` can be used when their is no content (the arguments to `CacheMe` are directly given in the body of `CacheMeNoContent`):
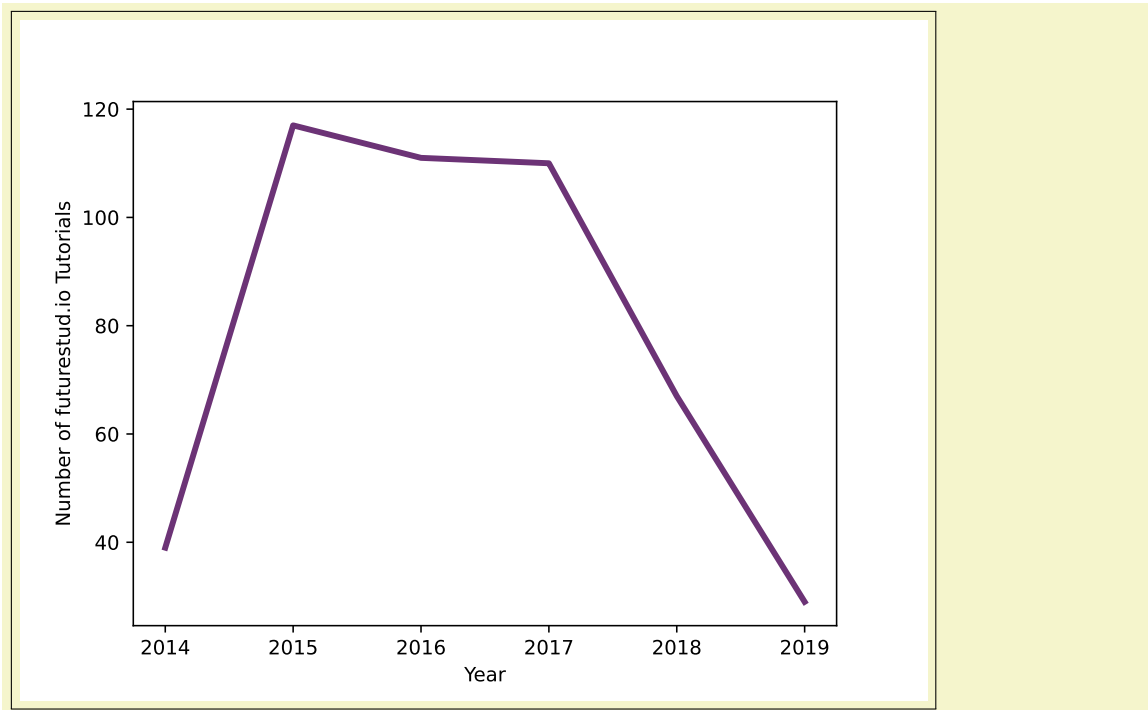
```
\begin{PlaceholderFromCode}{__TMP_MAIN_CONTENT__}
import matplotlib.pyplot as plt
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{PlaceholderFromCode}

\fbox{\begin{CacheMeNoContent}
  python,
  set includegraphics options={width=.8\linewidth},
  set main content={__TMP_MAIN_CONTENT__},
\end{CacheMeNoContent}}
```

3.14 Force to recompile or remove a cached item

If you want to recompile a file (e.g. an untracked dependency was changed... and you do not want to track it to avoid recompilation when you change a single line in this file), you can use the `recompile` style since v2.1:

$$A \longrightarrow B$$

```
\cacheEnvironment{tikzcd}{tikz, add to preamble={\usepackage{tikz-cd}}}
\begin{tikzcd}<recompile>
  A \rar & B
\end{tikzcd}
```

Note that this assumes that your compilation command is idempotent (so running it twice is like running it once) since the aux files are not cached. If you want to clean aux files or if you run an older version, see the documentation of `recompile` in section 4.3.3.

4 Documentation

Before starting this documentation, note that all commands are prefixed with `robExt` and all environments are prefixed with `RobExt`, but we also often define aliases without this prefix. The user is free to use any version, but we recommend to use the non-prefixed version unless a clash with a package forbids you from using it. In the following, we will only print the non-prefixed name when it exists. Note also that we follow the convention that environment names start with an upper case letter while commands start with a lower case letter.

4.1 How it works

This library must be able to generate 3 elements for any cached content:

- a source file, that will be compiled, and is obtained by expanding the placeholder `__ROBEXT_TEMPLATE__` (see section 4.5),

- a compilation command obtained by expanding the placeholder `__ROBEXT_COMPILATION_COMMAND__`,

- a dependency file, that contains the hash of all the dependencies (see section 3.4 for details) and the compilation command,

- an inclusion command (this one is not used during the caching process, it is only used when including the compiled document in the main document), that you can set using `custom include command={your command}`.

The hash of all these elements is computed in order to obtain a reference hash, denoted `somehash` that looks like a unique random value (note that `__ROBEXT_OUTPUT_PDF__` and alike are expanded after knowing the hash since they depend on the final hash value). This hash `somehash` will change whenever a dependency changes, or if the compilation command changes, ensuring purity. Then, the dependency file and the source file are written in the cache, by default in `robustExternalize/robExt-somehash.tex` and `robustExternalize/robExt-somehash.deps`. Then, the compilation command will be run from the cache folder. At the end, by default, we check if a file `robustExternalize/robExt-somehash.pdf` exists: if not we abort, otherwise we `\input` the file `robustExternalize/robExt-somehash-out.tex` and we run the include command (that includes the pdf by default). As we saw earlier, this command can be customized to use other files. **Importantly, all the files created during the compilation must be prefixed by** `robExt-somehash`, which can be obtained at runtime using `__ROBEXT_OUTPUT_PREFIX__`. This way, we can easily clean the cache while ensuring maximum purity.

In the following, we will denote by `*-foo.bar` the file in:
`robustExternalize/robExt-somehash-foo.bar`.

Note also that we usually define two names for each function, one normal and one prefixed with `robExt` (or `RobExt`) for environments. In this documentation, we only write the first form, but the second form is kept in case a conflicting package redefines some functions.

4.2 Placeholders

Placeholders are the main concept allowing this library to generate the content of a source file based on a template (a template will itself be a placeholder containing other placeholders). A placeholder is a special strings like `__COLOR_IMAGE__` that should start and end with two underscores, ideally containing no space or double underscores inside the name directly. Placeholders are inserted for instance in a string and will be given a value later. This value will be used to replace (recursively) the placeholder in the template. For instance, if a placeholder `__LIKES__` contains `I like __FRUIT__ and __VEGETABLE__`, if the placeholder `__FRUIT__` contains `oranges` and if the placeholder `__VEGETABLE__` contains `salad`, then evaluating `__LIKES__` will output `I like oranges and salad`.

Note that you are not strictly forced to follow the above convention (it allows us to optimize the code to find and replace placeholders), but in that case you should enable `not all placeholders have underscores`.

Placeholders are local variables (internally just some LATEX 3 strings). You can therefore define a placeholder in a local group surrounded by brackets { ... } if you want it to have a reduced scope.

4.2.1 Reading a placeholder

`\getPlaceholder[`⟨*new placeholder name*⟩`]{`⟨*name placeholder or string*⟩`}`
`\getPlaceholderInResult[`⟨*new placeholder name*⟩`]{`⟨*name placeholder or string*⟩`}`
`\getPlaceholderInResultFromList{`⟨*list,of,placeholders,to,replace*⟩`}[`⟨*new placeholder name*⟩`]{`⟨*name placeholder or string*⟩`}`

Get the value of a placeholder after replacing (recursively) all the inner placeholders. `\getPlaceholderInResult` puts the resulting string in a LATEX 3 string `\l_robExt_result_str` and in `\robExtResult`, while `\getPlaceholder` directly outputs this string. You can also put inside the argument any arbitrary string, allowing you, for instance, to concatenate multiple placeholders, copy a placeholder etc. Note that you will get a string, but this string will not be evaluated by LATEX (see `\evalPlaceholder` for that), for instance math will not be interpreted:

> The placeholder evaluates to:
> `Hello Alice the great, I am a template $\delta _n$.`
> Combining placeholders produces:
> ``In ``Hello Alice the great, I am a template $\delta _n$.'', the name is Alice the great.``

```
\setPlaceholder{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\setPlaceholder{__NAME__}{Alice __NICKNAME__}
\setPlaceholder{__NICKNAME__}{the great}
The placeholder evaluates to:\\
\texttt{\getPlaceholder{__MY_PLACEHOLDER__}}\\
Combining placeholders produces:\\
\texttt{\getPlaceholder{In ``__MY_PLACEHOLDER__'', the name is __NAME__.}}
```

You can also specify the optional argument in order to additionally define a new placeholder containing the resulting string (but you might prefer to use its alias `\setPlaceholderRec` described below):

> List of placeholders:
> - Placeholder called `__MY_PLACEHOLDER__` contains:
> `Hello __NAME__, I am a template $\delta _n$.`
> - Placeholder called `__NAME__` contains:
> `Alice __NICKNAME__`
> - Placeholder called `__NICKNAME__` contains:
> `the great`
> - Placeholder called `__NEW_PLACEHOLDER__` contains:
> ``In ``Hello Alice the great, I am a template $\delta _n$.'', the name is Alice the great.``

```
\setPlaceholder{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\setPlaceholder{__NAME__}{Alice __NICKNAME__}
\setPlaceholder{__NICKNAME__}{the great}
\getPlaceholderInResult[__NEW_PLACEHOLDER__]{In ``__MY_PLACEHOLDER__'', the name is __NAME__.}
\printImportedPlaceholdersExceptDefaults
```

The variation `\getPlaceholderInResultFromList` allows you to specify a set of placeholder to replace from:

<div style="background-color: #f5f5d0;">

List of placeholders:

- Placeholder called `__MY_PLACEHOLDER__` contains:

`Hello __NAME1__, __NAME2__ and __NAME3__, I am a template $\delta _n$.`

- Placeholder called `__NAME1__` contains:

`Alice`

- Placeholder called `__NAME2__` contains:

`Bob`

- Placeholder called `__NAME3__` contains:

`Charlie`

- Placeholder called `__NEW_PLACEHOLDER__` contains:

`  Here we go: Hello Alice, Bob and __NAME3__, I am a template $\delta _n$.`

</div>

```
\setPlaceholder{__MY_PLACEHOLDER__}{Hello __NAME1__, __NAME2__ and __NAME3__, I am a template $\delta_n$.}
\setPlaceholder{__NAME1__}{Alice}
\setPlaceholder{__NAME2__}{Bob}
\setPlaceholder{__NAME3__}{Charlie}
\getPlaceholderInResultReplaceFromList{__MY_PLACEHOLDER__,__NAME1__,__NAME2__}[__NEW_PLACEHOLDER__]{
  Here we go: __MY_PLACEHOLDER__
}
\printImportedPlaceholdersExceptDefaults
```

\evalPlaceholder{⟨*name placeholder or string*⟩}

Evaluate the value of a placeholder after replacing (recursively) all the inner placeholders. You can also put inside any arbitrary string.

<div style="background-color: #f5f5d0;">

The placeholder evaluates to:
Hello Alice the great, I am a template δ_n.
Combining placeholders produces:
In "Hello Alice the great, I am a template δ_n.", the name is Alice the great.

</div>

```
\setPlaceholder{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\setPlaceholder{__NAME__}{Alice __NICKNAME__}
\setPlaceholder{__NICKNAME__}{the great}
% The placeholder evaluates to \texttt{\getPlaceholder{__MY_PLACEHOLDER__}}.
The placeholder evaluates to:\\
\evalPlaceholder{__MY_PLACEHOLDER__}\\
Combining placeholders produces:\\
\evalPlaceholder{In ``__MY_PLACEHOLDER__'', the name is __NAME__.}
```

4.2.2   List and debug placeholders

It can sometimes be handy to list all placeholders, print their contents etc (see also section 4.9). We list here commands that are mostly useful for debugging purposes.

\printImportedPlaceholdersExceptDefaults*
/robExt/print imported placeholders except default   (style, no value)

Prints the verbatim content of all defined and imported placeholders (without performing any replacement of inner placeholders), except for the placeholders that are defined by default in this library (that we identify as they start with `__ROBEXT_`). The stared version does print the name of the placeholder defined in this library, but not their definition. This is mostly for debugging purposes.

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
\printImportedPlaceholdersExceptDefaults
```

Compare with:

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
\printImportedPlaceholdersExceptDefaults*
```

\printImportedPlaceholders
/robExt/print imported placeholders   (style, no value)

Prints the verbatim content of all defined and imported placeholders (without performing any replacement of inner placeholders), including the placeholders that are defined by default in this library (those starting with `__ROBEXT_`). This is mostly for debugging purposes. Here is the result of `\printImportedPlaceholders`:

List of placeholders:
- Placeholder called `__ROBEXT_MAIN_CONTENT__` contains:

`__ROBEXT_MAIN_CONTENT_ORIG__`

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_OPTIONS__` contains:

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_FILE__` contains:

`\robExtAddCachePathAndName {\robExtFinalHash .pdf}`

\printPlaceholderNoReplacement{⟨*name placeholder*⟩}

Prints the verbatim content of a given placeholder, without evaluating it and **without replacing inner placeholders: it is used mostly for debugging purposes** and will be used in this documentation to display the content of the placeholder for educational purposes. The stared version prints it inline.

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
The (unexpanded) template contains \printPlaceholderNoReplacement{__LIKES__}.\\
The (unexpanded) template contains \printPlaceholderNoReplacement*{__LIKES__}
```

\printPlaceholder{⟨*name placeholder*⟩}

Like **\printPlaceholderNoReplacement** except that it first replaces the inner placeholders. The stared version prints it inline.

> The (unexpanded) template contains
>
> `Hello Alice I am a really basic template $\delta _n$.`
>
> .
> The (unexpanded) template contains `Hello Alice I am a really basic template $\delta _n$.`

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
The (unexpanded) template contains \printPlaceholder{__LIKES__}.\\
The (unexpanded) template contains \printPlaceholder*{__LIKES__}
```

\evalPlaceholderNoReplacement{⟨*name placeholder*⟩}

Evaluates the content of a given placeholder as a LaTeX code, **without replacing the placeholders contained inside (mostly used for debugging purposes).**

> The (unexpanded) template evaluates to "Hello I am a really basic template δ_n.".

```
\placeholderFromContent{__LIKES__}{Hello I am a really basic template $\delta_n$.}
The (unexpanded) template evaluates to ``\evalPlaceholderNoReplacement{__LIKES__}''.
```

\rescanPlaceholderInVariableNoReplacement{⟨*name macro*⟩}{⟨*name placeholder*⟩}

(new v2.0) Create a new macro that executes the LaTeX code in the placeholder.

\getPlaceholderNoReplacement{⟨*name placeholder*⟩}

Like **\evalPlaceholderNoReplacement** except that it only outputs the string without evaluating the macros inside.

> The (unexpanded) template contains `Hello __NAME__ I am a really basic template $\delta _n$.`

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
The (unexpanded) template contains \texttt{\getPlaceholderNoReplacement{__LIKES__}}
```

4.2.3 Setting a value to a placeholder

\placeholderFromContent*{⟨*name placeholder*⟩}{⟨*content placeholder*⟩}
\setPlaceholder*{⟨*name placeholder*⟩}{⟨*content placeholder*⟩}
/robExt/set placeholder={⟨*name placeholder*⟩}{⟨*content placeholder*⟩}   (style, no default)
/robExt/set placeholder no import={⟨*name placeholder*⟩}{⟨*content placeholder*⟩} (style, no default)
/robExt/set placeholder from content={⟨*name placeholder*⟩}{⟨*content placeholder*⟩} (style, no default)

\placeholderFromContent (and its alias **\setPlaceholder** and its equivalent pgf styles **/robExt/set placeholder** and **/robExt/set placeholder from content**) is useful to set a value to a given placeholder.

The (unexpanded) template contains

```
Hello I am a basic template with math $\delta _n$ and macros \hello
```

and after evaluation and setting the value of hello, you get "Hello I am a basic template with math δ_n and macros Hello my friend!".

```
\placeholderFromContent{__LIKES__}{Hello I am a basic template with math $\delta_n$ and macros \hello}
The (unexpanded) template contains \printPlaceholderNoReplacement{__LIKES__} and %
after evaluation and setting the value of hello,%
\def\hello{Hello my friend!}%
you get ``\evalPlaceholder{__LIKES__}''.
```

As you can see, **the precise content is not exactly identical to the original string**: LaTeX comments are removed, spaces are added after macros, some newlines are removed etc. While this is usually not an issue when dealing with LaTeX code, it causes some troubles when dealing with non-LaTeX code. For this reason, we define **other commands** (see for instance `PlaceholderFromCode` below) that can accept verbatim content; the downside being that LaTeX forbids usage of these verbatim commands inside other macros, so you should always define them at the top level (this seems to be fundamental to how LaTeX works, as any input to a macro gets interpreted first as a LaTeX string, losing all comments for instance). Note that this is not as restrictive as it may sound, as it is always possible to define the needed placeholders before any macro, while using them inside the macro, possibly combining them with other placeholders (defined either before or inside the macro).

```
\begin{PlaceholderFromCode}*{⟨name placeholder⟩}
  ⟨environment contents⟩
\end{PlaceholderFromCode}
\begin{SetPlaceholderCode}*{⟨name placeholder⟩}
  ⟨environment contents⟩
\end{SetPlaceholderCode}
```

These two (aliased) environments are useful to set a verbatim value to a given placeholder: the advantage is that you can put inside any code, including LaTeX comments, the downside is that you cannot use it inside macros and some environments (so you typically define it before the macros and call it inside).

```
List of placeholders:
- Placeholder called __PYTHON_CODE__ contains:
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b
```

```
\begin{PlaceholderFromCode}{__PYTHON_CODE__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b
\end{PlaceholderFromCode}
\printImportedPlaceholdersExceptDefaults
```

Note that `PlaceholderFromCode` should not be used inside other macros or inside some environments (notably the ones that need to evaluate the body of the environment, e.g. using `+b` argument or `environ`) as verbatim content is parsed first by the macro, meaning that some characters might be changed or removed. For instance, any percent character would be considered as a comment, removing the rest of the line. However, this should not be be problem if you use it outside of any macro or environment, or if you load it from a file. For instance this code:

```
\begin{PlaceholderFromCode}{__PYTHON_CODE__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
\end{PlaceholderFromCode}
\printImportedPlaceholdersExceptDefaults
```

would produce:

List of placeholders:
- Placeholder called `__PYTHON_CODE__` contains:

```
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
```

`\printImportedPlaceholdersExceptDefaults`

Note that of course, you can define a placeholder before a macro and call it inside (explaining how we can generate this documentation).

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

\placeholderPathFromFilename*{⟨*name placeholder*⟩}{⟨*filename*⟩}
/robExt/set placeholder path from filename={⟨*name placeholder*⟩}{⟨*filename*⟩} (style, no default)

\placeholderPathFromFilename{__MYLIB__}{mylib.py} will copy `mylib.py` in the cache (setting its hash depending on its content), and set the content of the placeholder `__MYLIB__` to the **path** of the library in the cache. Note that the path is relative to the cache folder (it is easier to use for instance if you want to call this library from a code already in the cache).

List of placeholders:
- Placeholder called `__MYLIB__` contains:

`robExt-F6666F86DB0ACE43E817A7EB3729FA56mylib.py`

You can also get the path relative to the root folder:
robustExternalize/robExt-F6666F86DB0ACE43E817A7EB3729FA56mylib.py

```
\placeholderPathFromFilename{__MYLIB__}{mylib.py}
\printImportedPlaceholdersExceptDefaults
You can also get the path relative to the root folder:\\
\robExtAddCachePath{\getPlaceholderNoReplacement{__MYLIB__}}
```

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

\placeholderFromFileContent*{⟨*name placeholder*⟩}{⟨*filename*⟩}
/robExt/set placeholder from file content={⟨*name placeholder*⟩}{⟨*filename*⟩} (style, no default)
/robExt/set placeholder from file content no import={⟨*name placeholder*⟩}{⟨*filename*⟩} (style, no default)

\placeholderFromFileContent{__MYLIB__}{mylib.py} will set the content of the placeholder `__MYLIB__` to the content of `mylib.py`.

> List of placeholders:
> - Placeholder called `__MYLIB__` contains:
> ```
> def mylib():
>     # Some comments
>     a = b % 2
>     return "Hello world"
> ```

```
\placeholderFromFileContent{__MYLIB__}{mylib.py}
\printImportedPlaceholdersExceptDefaults
```

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

`\placeholderPathFromContent`*{⟨*name placeholder*⟩}[⟨*suffix*⟩]{⟨*content*⟩}`
`/robExt/set placeholder path from content`={⟨*name placeholder*⟩}{⟨*suffix*⟩}{⟨*content*⟩}
   (style, no default)
`/robExt/set placeholder path from content no import`={⟨*name placeholder*⟩}{⟨*suffix*⟩}{⟨*content*⟩}   (style, no default)
   `\placeholderPathFromContent{__MYLIB__}{some content}` will copy `some content` in a file in the cache (setting its hash depending on its content, the filename will end with `suffix` that defaults to `.tex`), and set the content of the placeholder `__MYLIB__` to the **path** of the file in the cache. Note that the path is relative to the cache folder (it is easier to use for instance if you want to call this library from a code already in the cache).

> List of placeholders:
> - Placeholder called `__MYLIB__` contains:
>
> `robExt-AC364A656060BFF5643DD21EAF3B64E6.py`
>
> You can also get the path relative to the root folder:
> robustExternalize/robExt-AC364A656060BFF5643DD21EAF3B64E6.py
> As a sanity check, this file contains
>
> ```
>     some contents b
> ```

```
\placeholderPathFromContent{__MYLIB__}[.py]{some contents b}
\printImportedPlaceholdersExceptDefaults
You can also get the path relative to the root folder:\\
\robExtAddCachePath{\getPlaceholderNoReplacement{__MYLIB__}}\\
As a sanity check, this file contains
\verbatiminput{\robExtAddCachePath{\getPlaceholderNoReplacement{__MYLIB__}}}
```

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

`\begin{PlaceholderPathFromCode}`*[⟨*suffix*⟩]{⟨*name placeholder*⟩}`
   ⟨*environment contents*⟩
`\end{PlaceholderPathFromCode}`

This environment is similar to `\placeholderPathFromContent` except that it accepts verbatim code (therefore LaTeX comments, newlines etc. will not be removed). However, due to LaTeX limitations, this environment cannot be used inside macros or some environments, or this property will not be preserved. For instance, if you create your placeholder using:

```
\begin{PlaceholderPathFromCode}[.py]{__MYLIB__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
```

```
        return b % 2
\end{PlaceholderPathFromCode}
```

You can then use it like:

List of placeholders:
- Placeholder called __MYLIB__ contains:

`robExt-CDAE704490400F29B9C0C8DAE2CC48B7.py`

You can also get the path relative to the root folder:
robustExternalize/robExt-CDAE704490400F29B9C0C8DAE2CC48B7.py
As a sanity check, this file contains

```
    def my_function(b): # this is a python code
        c = {}
        d[42] = 0
        return b % 2
```

```
\printImportedPlaceholdersExceptDefaults
You can also get the path relative to the root folder:\\
\robExtAddCachePath{\getPlaceholderNoReplacement{__MYLIB__}}\\
As a sanity check, this file contains
\verbatiminput{\robExtAddCachePath{\getPlaceholderNoReplacement{__MYLIB__}}}
```

Note that the star version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

\copyPlaceholder*{⟨*new placeholder*⟩}{⟨*old placeholder*⟩}
/robExt/copy placeholder={⟨*new placeholder*⟩}{⟨*old placeholder*⟩}   (style, no default)
/robExt/copy placeholder no import={⟨*new placeholder*⟩}{⟨*old placeholder*⟩}   (style, no default)

This creates a new placeholder with the content of `old placeholder`. Note that this is different from:
\setPlaceholder{new placeholder}{old placeholder}
because if we modify `old placeholder`, this will not affect `new placeholder`.

List of placeholders:
- Placeholder called __MY_OLD_CONTENT__ contains:

`Some content`

- Placeholder called __MY_CONTENT__ contains:

`The content used to be __MY_OLD_CONTENT__`

```
\setPlaceholder{__MY_CONTENT__}{Some content}
\copyPlaceholder{__MY_OLD_CONTENT__}{__MY_CONTENT__}
\setPlaceholder{__MY_CONTENT__}{The content used to be __MY_OLD_CONTENT__}
\printImportedPlaceholdersExceptDefaults
```

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4 for details).

Now, we see how we can define a placeholder recursively, by giving it a value based on its previous value (useful for instance in order to add stuff to it).

\setPlaceholderRec{⟨*new placeholder*⟩}{⟨*content with placeholder*⟩}
\setPlaceholderRecReplaceFromList{⟨*list,of,placeholder,to,replace*⟩}{⟨*new placeholder*⟩}{⟨*content with placeholder*⟩}
/robExt/set placeholder rec={⟨*name placeholder*⟩}{⟨*content placeholder*⟩} (style, no default)

/robExt/set placeholder rec replace from list={⟨*list,of,placeholder,to,replace*⟩}{⟨*name placeholder*⟩}{⟨*content placeholder*⟩}   (style, no default)

`\setPlaceholderRec{foo}{bar}` is actually an alias for `\getPlaceholderInResult[foo]{bar}`. Note that contrary to `\setPlaceholder`, it recursively replaces all inner placeholders. This is particularly useful to add stuff to an existing (or not) placeholder:

```
List of placeholders:
- Placeholder called __MY_COMMAND__ contains:
pdflatex myfile
```

```
\setPlaceholderRec{__MY_COMMAND__}{pdflatex}
\setPlaceholderRec{__MY_COMMAND__}{__MY_COMMAND__ myfile}
\printImportedPlaceholdersExceptDefaults
```

Note that the if the placeholder content contains at the end the placeholder name, we will automatically remove it to avoid infinite recursion at evaluation time. This has the benefit that you can add something to a placeholder even if this placeholder does not exists yet (in which case it will be understood as the empty string):

```
List of placeholders:
- Placeholder called __COMMAND_ARGS__ contains:
 -l -s
```

```
\setPlaceholderRec{__COMMAND_ARGS__}{__COMMAND_ARGS__ -l}
\setPlaceholderRec{__COMMAND_ARGS__}{__COMMAND_ARGS__ -s}
\printImportedPlaceholdersExceptDefaults
```

The variation `\setPlaceholderRecReplaceFromList` allows us to specify a subset of placeholder that will be allowed to be expanded, and is an alias for `\getPlaceholderInResultReplaceFromList` (except that the optional argument is mandatory):

```
List of placeholders:
- Placeholder called __MY_PLACEHOLDER__ contains:
Hello __NAME1__, __NAME2__ and __NAME3__, I am a template $\delta _n$.
- Placeholder called __NAME1__ contains:
Alice
- Placeholder called __NAME2__ contains:
Bob
- Placeholder called __NAME3__ contains:
Charlie
- Placeholder called __OTHER_PLACEHOLDER__ contains:
 Here we go: Hello __NAME1__, Bob and Charlie, I am a template $\delta _n$.
```

```
\placeholderFromContent{__MY_PLACEHOLDER__}{Hello __NAME1__, __NAME2__ and __NAME3__,
  I am a template $\delta_n$.
}
\placeholderFromContent{__NAME1__}{Alice}
\placeholderFromContent{__NAME2__}{Bob}
\placeholderFromContent{__NAME3__}{Charlie}
\setPlaceholderRecReplaceFromList{__MY_PLACEHOLDER__,__NAME2__,__NAME3__}{__OTHER_PLACEHOLDER__}{
  Here we go: __MY_PLACEHOLDER__
}
\printImportedPlaceholdersExceptDefaults
```

Note that the star and no import version does NOT import the placeholder it the main group (unless you try to optimize the compilation time you should not need it, but see section 4.2.4

for details).

Note that sometimes, you might not want to use `\setPlaceholderRec` to simply append some data to the placeholder as it will also evaluate the inner placeholders (meaning that you will not be able to redefine them later). For this reason, we also provide functions to add something to the placeholder without evaluating it first:

`\addToPlaceholder*{⟨placeholder⟩}{⟨content to add⟩}`
`\addToPlaceholderNoImport*{⟨placeholder⟩}{⟨content to add⟩}`
`\addBeforePlaceholder*{⟨placeholder⟩}{⟨content to add⟩}`
`\addBeforePlaceholderNoImport*{⟨placeholder⟩}{⟨content to add⟩}`
`/robExt/add to placeholder`={⟨name placeholder⟩}{⟨content to add⟩}   (style, no default)
`/robExt/add to placeholder no space`={⟨name placeholder⟩}{⟨content to add⟩}   (style, no default)
`/robExt/add before placeholder`={⟨name placeholder⟩}{⟨content to add⟩} (style, no default)
`/robExt/add before placeholder no space`={⟨name placeholder⟩}{⟨content to add⟩}   (style, no default)
`/robExt/add before main content`={⟨name placeholder⟩}{⟨content to add⟩} (style, no default)
`/robExt/add to placeholder no import`={⟨name placeholder⟩}{⟨content to add⟩}   (style, no default)
`/robExt/add to placeholder no space no import`={⟨name placeholder⟩}{⟨content to add⟩} (style, no default)
`/robExt/add before placeholder no import`={⟨name placeholder⟩}{⟨content to add⟩} (style, no default)
`/robExt/add before placeholder no space no import`={⟨name placeholder⟩}{⟨content to add⟩}   (style, no default)
`/robExt/add before main content no import`={⟨name placeholder⟩}{⟨content to add⟩}(style, no default)

`\addToPlaceholder{foo}{bar}` adds `bar` at the end of the placeholder `foo` (by default it also adds a space, unless you use the star version), creating it if it does not exist (the `before` variants add the content… before).

List of placeholders:
- Placeholder called `__ENGINE__` contains:

`pdflatex`
- Placeholder called `__COMMAND__` contains:

`time __ENGINE__ --option --other-option`

```
\setPlaceholder{__ENGINE__}{pdflatex}
\setPlaceholder{__COMMAND__}{__ENGINE__ --option}
\addToPlaceholder{__COMMAND__}{--other}
\addToPlaceholder*{__COMMAND__}{-option}
\addBeforePlaceholder{__COMMAND__}{time}
\printImportedPlaceholdersExceptDefaults
```

`add before main content` is a particular case where the placeholder is `__ROBEXT_MAIN_CONTENT__`. It is practical if you want to define for instance a macro, but in a way that even if you disable externalization, the command should still compile (if you define the macro in the preamble, it will not be added when disabling externalization). For instance:

Here I am cached (Hello Alice.) and Here I am not (Hello Bob.).

```
\robExtConfigure{
  new preset={my preset}{latex, add before main content={\def\hello#1{Hello #1.}}},
}
\cacheMe[my preset]{Here I am cached (\hello{Alice})} and \cacheMe[my
preset, disable externalization]{Here I am not (\hello{Bob}).}
```

The `no import` versions do not import the placeholder in the current group (only needed if you want to optimize the compilation time, see section 4.2.4 for details).

\placeholderFromString*{⟨*latex3 string*⟩}
\setPlaceholderFromString*{⟨*latex3 string*⟩}

(new in v2.0) This allows you to assign an existing LaTeX3 string to a placeholder (the star version does not import the placeholder, see section 4.2.4).

```
List of placeholders:
- Placeholder called __my_percent_string__ contains:
%
```

```
\ExplSyntaxOn
\setPlaceholderFromString{__my_percent_string__}{\c_percent_str}
\printImportedPlaceholdersExceptDefaults
\ExplSyntaxOff
```

We provide a list of placeholders that are useful to escape parts of the strings (but you should not really need them, if you need weird characters like percent, most of the time you want to use placeholderFromCode):

```
 String containing __ROBEXT_LEFT_BRACE__, __ROBEXT_RIGHT_BRACE__,
__ROBEXT_BACKSLASH__, __ROBEXT_HASH__, __ROBEXT_UNDERSCORE__,
__ROBEXT_PERCENT__.
```

```
\printPlaceholder{
   String containing
   __ROBEXT_LEFT_BRACE__,
   __ROBEXT_RIGHT_BRACE__,
   __ROBEXT_BACKSLASH__,
   __ROBEXT_HASH__,
   __ROBEXT_UNDERSCORE__,
   __ROBEXT_PERCENT__.
}
```

\placeholderReplaceInplace{⟨*placeholder*⟩}{⟨*from*⟩}{⟨*to*⟩}
\placeholderReplaceInplaceEval{⟨*placeholder*⟩}{⟨*from*⟩}{⟨*to*⟩}
 /robExt/placeholder replace in place={⟨*placeholder*⟩}{⟨*from*⟩}{⟨*to*⟩}   (style, no default)
 /robExt/placeholder replace in place eval={⟨*placeholder*⟩}{⟨*from*⟩}{⟨*to*⟩}   (style, no default)

(new in v2.0) This allows you to replace a value in a placeholder. The `eval` variation first evaluates the string.

```
List of placeholders:
- Placeholder called __NAMES__ contains:
Hello Charlie and Dylan.
```

```
\def\nameFrom{Bob}
\def\nameTo{Dylan}
\robExtConfigure{
  set placeholder={__NAMES__}{Hello Alice and Bob.},
  placeholder replace in place={__NAMES__}{Alice}{Charlie},
  placeholder replace in place eval={__NAMES__}{\nameFrom}{\nameTo},
}
\printImportedPlaceholdersExceptDefaults
```

\placeholderHalveNumberHashesInplace{⟨*placeholder*⟩}

\placeholderDoubleNumberHashesInplace{⟨*placeholder*⟩}
/robExt/placeholder halve number hashes in place={⟨*placeholder*⟩}   (style, no default)
/robExt/placeholder double number hashes in place={⟨*placeholder*⟩}   (style, no default)

    (new in v2.0) This allows you to either turn any `##` into `#` or the other way around (may be practical when dealing with arguments to functions).

> List of placeholders:
> - Placeholder called `__DEMO__` contains:
>
> `\def \hey #1{Hey #1.}`

```
\robExtConfigure{
  set placeholder={__DEMO__}{\def\hey##1{Hey ##1.}},
  placeholder halve number hashes in place={__DEMO__},
}
\printImportedPlaceholdersExceptDefaults
```

\removePlaceholder{⟨*placeholder*⟩}
/robExt/remove placeholder={⟨*placeholder*⟩}   (style, no default)
/robExt/remove placeholders={⟨*list,of,placeholder*⟩}   (style, no default)

    (new in v2.0) Remove placeholders.

> List of placeholders:
> - Placeholder called `__DEMOB__` contains:
>
> `Bob`

```
\robExtConfigure{
  set placeholder={__DEMOA__}{Alice},
  set placeholder={__DEMOB__}{Bob},
  set placeholder={__DEMOC__}{Charlie},
  remove placeholders={__DEMOA__,__DEMOC__},
}
\printImportedPlaceholdersExceptDefaults
```

\evalPlaceholderInplace{⟨*name placeholder*⟩}
/robExt/eval placeholder inplace={⟨*name placeholder*⟩}   (style, no default)

    This command will update (inplace) the content of a macro by first replacing recursively the placeholders, and finally by expanding the LATEX macros.

> List of placeholders:
> - Placeholder called `__MACRO_NOT_EVALUATED__` contains:
>
> `\mymacro`
>
> - Placeholder called `__MACRO_EVALUATED__` contains:
>
> `Initial value`
>
> Compare Initial value and Final value.

```
\def\mymacro{Initial value}
\placeholderFromContent{__MACRO_NOT_EVALUATED__}{\mymacro}
\placeholderFromContent{__MACRO_EVALUATED__}{\mymacro}
\evalPlaceholderInplace{__MACRO_EVALUATED__}
\printImportedPlaceholdersExceptDefaults
\def\mymacro{Final value}
Compare \evalPlaceholder{__MACRO_EVALUATED__} and \evalPlaceholder{__MACRO_NOT_EVALUATED__}.
```

/robExt/set placeholder eval={⟨*name placeholder*⟩}{⟨*content placeholder*⟩}   (style, no default)

43

Alias for `\setPlaceholderRec{#1}{#2}\evalPlaceholderInplace{#1}`: set and evaluate recursively the placeholders and macros. This can be practical to pass the value of a counter/macro to the template (of course, if this value is fixed, you can also directly load it from the preambule):

> The current page is 44.

```
\begin{CacheMe}{tikzpicture, set placeholder eval={__thepage__}{\thepage}}
  \node[rounded corners, fill=red]{The current page is __thepage__.};
\end{CacheMe}
```

Note that this works well for commands that expand completely, but some more complex commands might not expand properly (like `cref`). I need to investigate how to solve this issue, meanwhile you can still disable externalization for these pictures.

4.2.4 Groups: the import system

(This whole system was added v2.0.)

In order to replace placeholders, we need to know their list, but to get the best possible performance we do not maintain a single list of all placeholders since LaTeX is quite slow when doing list manipulations. Therefore, we group them inside smaller groups (e.g. one group for `latex`, one group for `python` etc), and the user is free to import groups. By default, new placeholders are imported inside the `main` group, and this group will be used when doing placeholder replacements. Note that for most of the case, we define a star variant of functions like `\setPlaceholder*{__FOO__}{bar}` (or `set placeholder no import` for styles) in order to create a placeholder without importing it (but this is certainly not used a lot by end users unless they want to make them even faster).

Note that basically all commands to create a placeholder have a command/environment where you add a star like `\setPlaceholder*{}` or a style where you add `no import` like `set placeholder no import` in order to create a placeholder that is not imported at all.

`\importPlaceholder{⟨name placeholder⟩}`
`\importPlaceholdersFromGroup{⟨name placeholder⟩}`
`\importAllPlaceholders`
`\importPlaceholderFirst`
`/robExt/import placeholder={⟨name placeholder⟩}`   (style, no default)
`/robExt/import placeholders from group={⟨name group⟩}`   (style, no default)
`/robExt/import all placeholders`   (style, no value)
`/robExt/import placeholder first={⟨name placeholders⟩}`   (style, no default)

Import placeholders, either individually, from a group of placeholders (exists either in style form or command), or from all registered groups (warning: you should avoid importing all placeholders if care too much about efficiency, as this can significantly slow down the replacement procedure). The `first` variant inserts the placeholder at the beginning of the import list, which can speed up compilation when this placeholder must be replaced first (but not so useful for basic usage).

List of placeholders:
List of placeholders:
- Placeholder called `__name__` contains:

```
Alice
```

List of placeholders:
- Placeholder called `__ROBEXT_MAIN_CONTENT__` contains:

```
__ROBEXT_MAIN_CONTENT_ORIG__
```

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_OPTIONS__` contains:

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_FILE__` contains:

```
\robExtAddCachePathAndName {\robExtFinalHash .pdf}
```

- Placeholder called `__name__` contains:

```
Alice
```

List of placeholders:
- Placeholder called `__ROBEXT_MAIN_CONTENT__` contains:

```
__ROBEXT_MAIN_CONTENT_ORIG__
```

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_OPTIONS__` contains:

- Placeholder called `__ROBEXT_INCLUDEGRAPHICS_FILE__` contains:

```
\robExtAddCachePathAndName {\robExtFinalHash .pdf}
```

- Placeholder called `__name__` contains:

```
Alice
```

- Placeholder called `__ROBEXT_LATEX__` contains:

```
\documentclass[__ROBEXT_LATEX_OPTIONS__]{__ROBEXT_LATEX_DOCUMENT_CLASS__}
__ROBEXT_LATEX_PREAMBLE__
% most packages must be loaded before hyperref
% so we typically want to load hyperref here
__ROBEXT_LATEX_PREAMBLE_HYPERREF__
% some packages must be loaded after hyperref
__ROBEXT_LATEX_PREAMBLE_AFTER_HYPERREF__
\begin{document}%
__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__
\end{document}
```

- Placeholder called `__ROBEXT_LATEX_OPTIONS__` contains:

- Placeholder called `__ROBEXT_LATEX_DOCUMENT_CLASS__` contains:

```
standalone
```

- Placeholder called `__ROBEXT_LATEX_PREAMBLE__` contains:

- Placeholder called `__ROBEXT_LATEX_PREAMBLE_HYPERREF__` contains:

- Placeholder called `__ROBEXT_LATEX_PREAMBLE_AFTER_HYPERREF__` contains:

- Placeholder called `__ROBEXT_LATEX_TRIM_LENGTH__` contains:

```
30cm
```

- Placeholder called `__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__` contains:

```
__ROBEXT_LATEX_CREATE_OUT_FILE__%
\newsavebox\boxRobExt%
```

```
\begin{lrbox}{\boxRobExt}%
   __ROBEXT_MAIN_CONTENT__%
\end{lrbox}%
```

```
\robExtConfigure{
  set placeholder no import={__name__}{Alice},
  set placeholder no import={__name2__}{Bob},
  print imported placeholders except default,
  import placeholder={__name__},
  print imported placeholders except default,
  print imported placeholders,
  import placeholders from group={latex},
  print imported placeholders,
}
```

```
First try: __ROBEXT_LATEX_TRIM_LENGTH__

Second try: 30cm
```

```
\printPlaceholder{First try: __ROBEXT_LATEX_TRIM_LENGTH__}
\importAllPlaceholders
\printPlaceholder{Second try: __ROBEXT_LATEX_TRIM_LENGTH__}
```

\clearImportedPlaceholders{⟨*name placeholder*⟩}
/robExt/clear imported placeholders   (style, no value)

Clear all imported placeholders (comma separated list of placeholders)

```
List of placeholders:
- Placeholder called __name__ contains:

Alice
List of placeholders:
```

```
\robExtConfigure{
  set placeholder={__name__}{Alice},
  print imported placeholders except default,
  clear imported placeholders,
  print imported placeholders except default,
}
```

\removeImportedPlaceholder{⟨*name placeholder*⟩}
/robExt/remove imported placeholders   (style, no value)

Comma separated list of placeholders to remove from the list of imported placeholders.

```
List of placeholders:
- Placeholder called __name__ contains:

Alice
- Placeholder called __name2__ contains:

Alice
List of placeholders:
```

```
\robExtConfigure{
  set placeholder={__name__}{Alice},
  set placeholder={__name2__}{Alice},
  print imported placeholders except default,
  remove imported placeholders={__name__,__name2__},
  print imported placeholders except default,
}
```

\printAllRegisteredGroups

\printAllRegisteredGroupsAndPlaceholders[*]
/robExt/print all registered groups	(style, no value)
/robExt/print all registered groups and placeholders	(style, no value)
/robExt/show all registered groups	(style, no value)
/robExt/show all registered groups and placeholders	(style, no value)

Print (or shows in the console for the alternative versions) all registered groups of placeholders (possibly with the placeholders they contain), mostly for debugging purpose, with the star prints also the content of the inner placeholders.

```
special characters
latex
python
python print code result
verbatim
gnuplot
bash
default
```

```
\printAllRegisteredGroups
```

\printGroupPlaceholders[*]
/robExt/print group placeholders={⟨*name group*⟩}	(style, no default)

Print all placeholders (star = with content) of a given group.

```
Content of group bash:
- Placeholder __ROBEXT_BASH_TEMPLATE__
# Quit if there is an error
set -e
outputTxt="__ROBEXT_OUTPUT_PREFIX__-out.txt"
outputTex="__ROBEXT_OUTPUT_PREFIX__-out.tex"
outputPdf="__ROBEXT_OUTPUT_PDF__"
__ROBEXT_MAIN_CONTENT__
# Create the pdf file to certify that no compilation error occured
touch "${outputPdf}"
- Placeholder __ROBEXT_BASH_SHELL__
bash
```

```
\printGroupPlaceholders*{bash}
```

\printAllRegisteredGroupsAndPlaceholders[*]

Print all registered groups of placeholders with their inner content, mostly for debugging purpose, with the star prints only the content of the inner placeholders.

- Group`special characters`:

Placeholder`__ROBEXT_LEFT_BRACE__`

Placeholder`__ROBEXT_RIGHT_BRACE__`

Placeholder`__ROBEXT_BACKSLASH__`

Placeholder`__ROBEXT_HASH__`

Placeholder`__ROBEXT_PERCENT__`

Placeholder`__ROBEXT_UNDERSCORE__`

- Group`latex`:

Placeholder`__ROBEXT_LATEX__`

Placeholder`__ROBEXT_LATEX_OPTIONS__`

Placeholder`__ROBEXT_LATEX_DOCUMENT_CLASS__`

Placeholder`__ROBEXT_LATEX_PREAMBLE__`

Placeholder`__ROBEXT_LATEX_PREAMBLE_HYPERREF__`

Placeholder`__ROBEXT_LATEX_PREAMBLE_AFTER_HYPERREF__`

Placeholder`__ROBEXT_LATEX_TRIM_LENGTH__`

Placeholder`__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__`

Placeholder`__ROBEXT_LATEX_CREATE_OUT_FILE__`

Placeholder`__ROBEXT_LATEX_WRITE_DEPTH_TO_OUT_FILE__`

Placeholder`__ROBEXT_LATEX_COMPILATION_COMMAND__`

Placeholder`__ROBEXT_LATEX_COMPILATION_COMMAND_OPTIONS__`

Placeholder`__ROBEXT_LATEX_ENGINE__`

- Group`python`:

Placeholder`__ROBEXT_PYTHON__`

Placeholder`__ROBEXT_PYTHON_IMPORT__`

Placeholder`__ROBEXT_PYTHON_MAIN_CONTENT_WRAPPED__`

Placeholder`__ROBEXT_PYTHON_FINISHED_WITH_NO_ERROR__`

Placeholder`__ROBEXT_PYTHON_EXEC__`

- Group`python print code result`:

Placeholder`__ROBEXT_PYTHON_PRINT_CODE_RESULT_TEMPLATE_BEFORE__`

Placeholder`__ROBEXT_PYTHON_PRINT_CODE_RESULT_TEMPLATE_AFTER__`

Placeholder`__ROBEXT_PYTHON_TCOLORBOX_PROPS__`

Placeholder`__ROBEXT_PYTHON_CODE_MESSAGE__`

Placeholder`__ROBEXT_PYTHON_RESULT_MESSAGE__`

Placeholder`__ROBEXT_PYTHON_LSTINPUT_STYLE__`

- Group`verbatim`:

- Group`gnuplot`:

Placeholder`__ROBEXT_GNUPLOT_PRELUDE__`

- Group`bash`:

Placeholder`__ROBEXT_BASH_TEMPLATE__`

Placeholder`__ROBEXT_BASH_SHELL__`

- Group`default`:

Placeholder`__ROBEXT_MAIN_CONTENT__`

Placeholder`__ROBEXT_INCLUDEGRAPHICS_OPTIONS__`

Placeholder`__ROBEXT_INCLUDEGRAPHICS_FILE__`

```
\printAllRegisteredGroupsAndPlaceholders
```

\newGroupPlaceholders{⟨*name group*⟩}
\addPlaceholderToGroup{⟨*name group*⟩}{⟨*list,of,placeholders*⟩}
\addPlaceholdersToGroup{⟨*name group*⟩}{⟨*list,of,placeholders*⟩}
\removePlaceholderFromGroup{⟨*name group*⟩}{⟨*list,of,placeholders*⟩}
\removePlaceholdersFromGroup{⟨*name group*⟩}{⟨*list,of,placeholders*⟩}
/robExt/new group placeholders={⟨*name group*⟩}   (style, no default)
/robExt/add placeholder to group={⟨*name group*⟩}{⟨*name placeholder*⟩}   (style, no default)
/robExt/add placeholders to group={⟨*name group*⟩}{⟨*list,of,placeholders*⟩}(style, no default)
/robExt/add placeholders to group={⟨*name group*⟩}{⟨*list,of,placeholders*⟩}(style, no default)
/robExt/remove placeholders from group={⟨*name group*⟩}{⟨*list,of,placeholders*⟩}   (style, no default)
/robExt/remove placeholder from group={⟨*name group*⟩}{⟨*list,of,placeholders*⟩}   (style, no default)

Create a new group and add placeholders to it (you can put multiple placeholders separated by commas).

> Content of group my dummy group:
> - Placeholder __name__
> Content of group my dummy group:

```
\robExtConfigure{
  new group placeholders={my dummy group},
  set placeholder no import={__name__}{Alice},
  add placeholders to group={my dummy group}{__name__},
}
\printGroupPlaceholders{my dummy group}
\removePlaceholdersFromGroup{my dummy group}{__name__}
\printGroupPlaceholders{my dummy group}
```

\clearGroupPlaceholders{⟨*name group*⟩}

Remove all elements in a group.

> Content of group my dummy group:

```
\robExtConfigure{
  new group placeholders={my dummy group},
  set placeholder no import={__name__}{Alice},
  add placeholders to group={my dummy group}{__name__},
}
\clearGroupPlaceholders{my dummy group}
\printGroupPlaceholders{my dummy group}
```

\copyGroupPlaceholders{⟨*name group*⟩}{⟨*name group to copy*⟩}
/robExt/copy group placeholders={⟨*name group*⟩}{⟨*name group to copy*⟩} (style, no default)

Remove all elements in a group.

> Content of group my copy:
> - Placeholder __name__

```
\robExtConfigure{
  new group placeholders={my dummy group},
  set placeholder no import={__name__}{Alice},
  add placeholders to group={my dummy group}{__name__},
}
\copyGroupPlaceholders{my copy}{my dummy group}
\printGroupPlaceholders{my copy}
```

`\appendGroupPlaceholders`{⟨*name group*⟩}{⟨*name group to append*⟩}
`\appendBeforeGroupPlaceholders`{⟨*name group*⟩}{⟨*name group to append*⟩}
`/robExt/append group placeholders`={⟨*name group*⟩}{⟨*name group to copy*⟩}   (style, no default)
`/robExt/append before group placeholders`={⟨*name group*⟩}{⟨*name group to copy*⟩} (style, no default)

Append a group to another group (after or before).

Content of group `my dummy group`:
- Placeholder `__name__`
- Placeholder `__name2__`

```
\robExtConfigure{
  new group placeholders={my dummy group},
  set placeholder no import={__name__}{Alice},
  add placeholders to group={my dummy group}{__name__},
  new group placeholders={my dummy group2},
  set placeholder no import={__name2__}{Bob},
  add placeholders to group={my dummy group2}{__name2__},
  append group placeholders={my dummy group}{my dummy group2},
}
\printGroupPlaceholders{my dummy group}
```

`\importPlaceholdersFromGroup`{⟨*name group*⟩}
`\importAllPlaceholders`
`/robExt/import placeholders from group`={⟨*name group*⟩}   (style, no default)
`/robExt/import all placeholders`   (style, no value)

Import a group/all registered groups (note that this is equivalent to appending to the group called `main`). Note that for the later, you should be sure that the group is "registered", which is the case if you created it via `new group placeholders`.

List of placeholders:
- Placeholder called `__name__` contains:
`Alice`

```
\robExtConfigure{
  new group placeholders={my dummy group},
  set placeholder no import={__name__}{Alice},
  add placeholder to group={my dummy group}{__name__},
  import all placeholders,
}
\printImportedPlaceholdersExceptDefaults
```

4.3 Caching a content

4.3.1 Basics

`\cacheMe`[⟨*preset style*⟩]{⟨*content to cache*⟩}
`\begin{CacheMe}`{⟨*preset style*⟩}
  ⟨*environment contents*⟩
`\end{CacheMe}`

This command (and its environment alias) is the main entry point if you want to cache the result of a file. The preset style is a pgfkeys-based style that is used to configure the template that is used, the compilation command, and more. You can either in-line the style, or use some presets that configure the style automatically. After evaluating the style, the placeholders `__ROBEXT_TEMPLATE__` (containing the content of the file) and `__ROBEXT_COMPILATION_COMMAND__` (containing the compilation command run in

the cache folder, that can use other placeholders internally like `__ROBEXT_SOURCE_FILE__` to get the path to the source file) should be set. Note that we provide some basic styles that allow settings these placeholders easily. See section 4.5 for a list of existing placeholders and presets. The placeholder `__ROBEXT_MAIN_CONTENT_ORIG__` will automatically be set by this command (or environment) so that it equals the content of the second argument (or the body of the environment). By default, `__ROBEXT_MAIN_CONTENT__` will point to `__ROBEXT_MAIN_CONTENT_ORIG__`, possibly wrapping it inside an environment like `\begin{tikzpicture}` (most of the time, you want to modify and display `__ROBEXT_MAIN_CONTENT__` rather than the `_ORIG_` to easily recover the input of the user). This style can also configure the command to use to include the file and more. By default it will insert the compiled PDF, making sure that the depth is respected (internally, we read the depth from an aux file created by our LATEX preset), but you can easily change it to anything you like.

For an educational purpose, we write here an example that does not exploit any preset. In practice, we recommend however to use our presets, or to define new presets based on our presets (see below for examples).
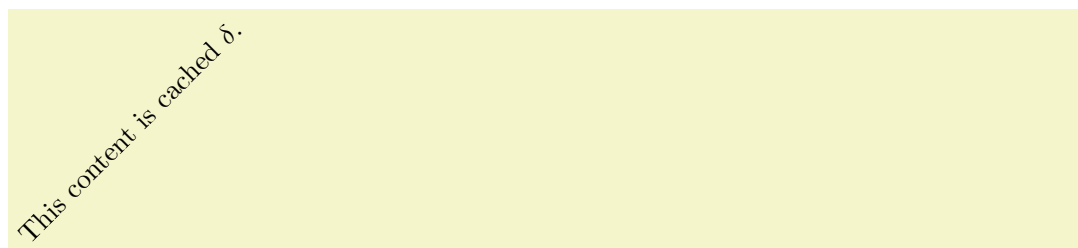

```
\begin{CacheMe}{set template={
        \documentclass{standalone}
        \begin{document}
        __ROBEXT_MAIN_CONTENT__
        \end{document}
    },
    set compilation command={pdflatex -shell-escape -halt-on-error "__ROBEXT_SOURCE_FILE__"},
    custom include command={%
        \includegraphics[width=4cm,angle=45]{\robExtAddCachePathAndName{\robExtFinalHash.pdf}}%
    },
  }
This content is cached $\delta$.
\end{CacheMe}
```

`\robExtConfigure`{⟨*preset style*⟩}
`/robExt/new preset`={⟨*name preset*⟩}{⟨*preset options*⟩}   (style, no default)
`/robExt/add to preset`={⟨*name preset*⟩}{⟨*preset options*⟩}   (style, no default)

You can then create your own style (or preset) in `\robExtConfigure` (that is basically an alias for `\pgfkeys{/robExt/.cd,#1}`) containing your template, add your own placeholders and commands to configure them etc. We provide two helper functions since v2.0:
`new preset={your preset}{your configuration}`
and
`add to preset={your preset}{your configuration}`
in order to create/modify the presets. You can also use `my preset/.style` or
`my preset/.append style`
to configure them instead, but in that case make sure to double the number of hashes like in `\def\mymacro##1{hello ##1}`, as the #1 in `\def\mymacro#1{hello #1}`. would be understood as the (non-existent) argument of `my preset`.

```
%% Define your presets once:
\robExtConfigure{%
  new preset={my latex preset}{
    %% Create a default value for my new placeholders:
    set placeholder={__MY_COLOR__}{red},
    set placeholder={__MY_ANGLE__}{45},
    % We can also create custom commands to "hide" the notion of placeholder
    set my angle/.style={
      set placeholder={__MY_ANGLE__}{#1}
    },
    set template={
      \documentclass{standalone}
      \usepackage{xcolor}
      \begin{document}
      \color{__MY_COLOR__}__ROBEXT_MAIN_CONTENT__
      \end{document}
    },
    set compilation command={pdflatex -shell-escape -halt-on-error "__ROBEXT_SOURCE_FILE__"},
    custom include command={%
      % The include command is a regular LaTeX command, but using
      % \evalPlaceholder avoids the need to play with expandafter, getPlaceholder etc...
      \evalPlaceholder{%
        \includegraphics[width=4cm,angle=__MY_ANGLE__,origin=c]{%
          \robExtAddCachePathAndName{\robExtFinalHash.pdf}%
        }%
      }%
    },
  },
}

% Reuse them later...
\begin{CacheMe}{my latex preset}
This content is cached $\delta$.
\end{CacheMe}
% And configure them at will
\begin{CacheMe}{my latex preset, set placeholder={__MY_COLOR__}{green}, set my angle=-45}
This content is cached $\delta$.
\end{CacheMe}
```

\begin{CacheMeCode}{⟨preset style⟩}
  ⟨environment contents⟩
\end{CacheMeCode}

Like `CacheMe`, except that the code is read verbatim by LaTeX. This way, you can put non-LaTeX code inside safely, but you will not be able to use it inside a macro or some environments that read their body. Here is an example where we define an environment that automatically import matplotlib, save the figure, and insert it into a figure. Note that we define in this example new commands to type `set caption=foo` instead of `set placeholder={__FIG_CAPTION__}{foo}`.

```
%% Define the python code to use as a template
%% (impossible to define it in \robExtConfigure directly since
%% it is a verbatim environment)
\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE__}
import matplotlib.pyplot as plt
import sys
__ROBEXT_MAIN_CONTENT__
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{PlaceholderFromCode}

% Create a new preset called matplotlib
\robExtConfigure{
  new preset={matplotlib figure}{
    set template={__MY_MATPLOTLIB_TEMPLATE__},
    set compilation command={python "__ROBEXT_SOURCE_FILE__"},
    set caption/.style={
      set placeholder={__FIG_CAPTION__}{#1}
    },
    set label/.style={
      set placeholder={__FIG_LABEL__}{#1}
    },
    set includegraphics options/.style={
      set placeholder={__INCLUDEGRAPHICS_OPTIONS__}{#1}
    },
    set caption={},
    set label={},
    set includegraphics options={width=1cm},
    custom include command={%
      \evalPlaceholder{%
        \begin{figure}
          \centering
          \includegraphics[__INCLUDEGRAPHICS_OPTIONS__]{\robExtAddCachePathAndName{\robExtFinalHash.pdf}}%
          \caption{__FIG_CAPTION__a}
          \label{__FIG_LABEL__}
        \end{figure}%
      }%
    },
  },
}

%% Use it
\begin{CacheMeCode}{matplotlib figure, set includegraphics options={width=.8\linewidth}, set caption={Hello}}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{CacheMeCode}
```

Note that as we explained it before, due to LaTeX limitations, it is impossible to call
`CacheMeCode` inside macros and inside some environments that evaluate their body. To avoid
that issue, it is always possible to define the macro before and call it inside. We will exemplify this on the previous example, but note that **this example is only for educational
purposes** since the environment `figure` does not evaluate its body, and `CacheMeCode` can
therefore safely be used inside without using this trickery:
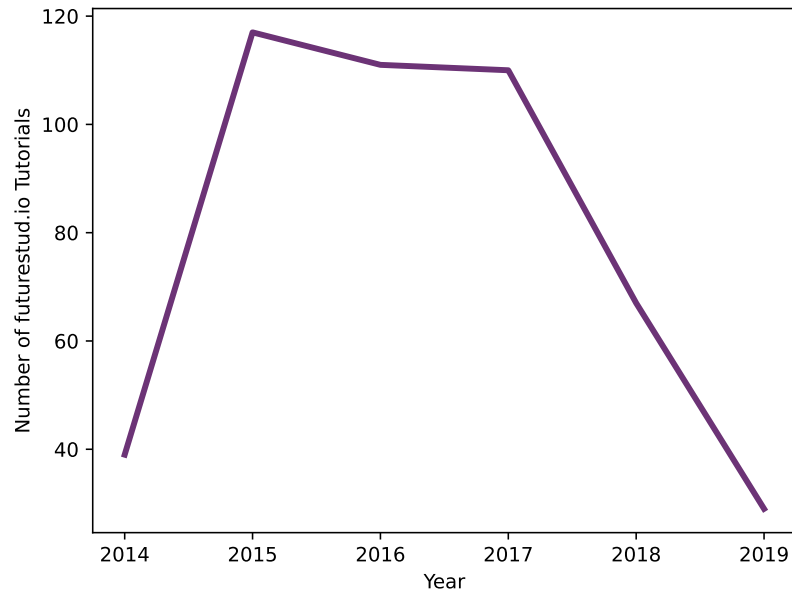
Figure 2: An example to show how matplotlib pictures can be inserted

```
%% Define the python code to use as a template
%% (impossible to define it in \robExtConfigure directly since
%% it is a verbatim environment)
\begin{PlaceholderFromCode}{__MY_MATPLOTLIB_TEMPLATE__}
import matplotlib.pyplot as plt
import sys
__ROBEXT_MAIN_CONTENT__
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{PlaceholderFromCode}

% Create a new preset called matplotlib
\robExtConfigure{
  new preset={matplotlib}{
    set template={__MY_MATPLOTLIB_TEMPLATE__},
    set compilation command={python "__ROBEXT_SOURCE_FILE__"},
    set includegraphics options/.style={
      set placeholder={__INCLUDEGRAPHICS_OPTIONS__}{#1}
    },
    set includegraphics options={width=1cm},
    custom include command={%
      \evalPlaceholder{%
        \includegraphics[__INCLUDEGRAPHICS_OPTIONS__]{\robExtAddCachePathAndName{\robExtFinalHash.pdf}}%
      }%
    },
  },
}


%% You cannot use CacheMeCode inside some macros or environments due to fundamental LaTeX limitations.
%% But you can always define them before, and call them inside:
\begin{SetPlaceholderCode}{__TMP__}
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
\end{SetPlaceholderCode}

\begin{figure}
  \centering
  \cacheMe[matplotlib, set includegraphics options={width=.8\linewidth}, set
caption={Hello}]{__TMP__}
  \caption{An example to show how code can be inserted into macros or environments that evaluate their contents (th
\end{figure}
```

4.3.2 Options to configure the template

/robExt/set template={⟨*content template*⟩}   (style, no default)

    Style that alias to `set placeholder={__ROBEXT_TEMPLATE__}{#1}`, in order to define the placeholder that will hold the template of the final file.

/robExt/set template={⟨*content template*⟩}   (style, no default)

    Style that alias to `set placeholder={__ROBEXT_TEMPLATE__}{#1}`, in order to define the placeholder that will hold the template of the final file.

4.3.3 Options to configure the compilation command

/robExt/set compilation command={⟨*compilation command*⟩}   (style, no default)

    Style that alias to `set placeholder={__ROBEXT_COMPILATION_COMMAND__}{#1}`, in order to define the placeholder that will hold the compilation command.

/robExt/add argument to compilation command={⟨*argument*⟩}   (style, no default)
/robExt/add arguments to compilation command={⟨*argument*⟩}   (style, no default)

    `add argument to compilation command` is a style that alias to:
    `set placeholder={__ROBEXT_COMPILATION_COMMAND__}{__ROBEXT_COMPILATION_COMMAND__ "#1"}`
    in order to add an argument to the compilation command. `add arguments to compilation command`
    (note the `s`) accepts multiple arguments separated by a comma.

/robExt/add key value argument to compilation command={⟨*key=value*⟩}(style, no default)

    Adds to the command line two arguments `key` and `value`. This is a way to quickly pass arguments to a script: the script just needs to loop over the arguments and consider the odd elements as keys and the next elements as the value. Another option is to insert some placeholders directly in the script.

/robExt/add key and file argument to compilation command={⟨*key=filename*⟩} (style, no default)

    `filename` is the path to a file in the root folder. This adds, as:
    `add key value argument to compilation command`
    two arguments, where the first argument is the key, but this time the second argument is the path of `filename` relative to the cache folder (useful since scripts run from this folder). Moreover, it automatically ensures that when `filename` changes, the file gets recompiled. Note that contrary to some other commands, this does not copy the file in the cache, which is practical notably for large files like videos.

/robExt/force compilation   (style, no value)
/robExt/do not force compilation   (style, no value)

    By default, we compile cached documents only if `-shell-escape` is enabled. However, if the user allowed `pdflatex` (needed to compile latex documents), `cd` (not needed when using `no cache folder`), and `mkdir` (not needed when using `no cache folder` or if the cache folder that defaults to `robustExternalize` is already created) to run in restricted mode (so without enabling `-shell-escape`), then there is no need for `-shell-escape`. In that case, set `force compilation` and this library will compile even if `-shell-escape` is disabled.

/robExt/recompile   (style, no value)
/robExt/do not recompile   (style, no value)

    (new to v2.1) `recompile` will force a compilation even if the image is already cached (mostly useful if you made a change to a non-tracked dependency). Note that since this library does not want to remove any file on the hard drive (we do not want to risk to remove important files in case of a bug), this command will NOT clear the auxiliary files already present, so for maximum purity, ensure that your compiling command is idempotent (so that compiling twice gives the same outcome as compiling once).

$$A \longrightarrow B$$

```
\cacheEnvironment{tikzcd}{tikz, add to preamble={\usepackage{tikz-cd}}}
\begin{tikzcd}<recompile>
  A \rar & B
\end{tikzcd}
```

If you want to actually remove the aux files, you can either change the compilation command, or add something like this to remove the file (note that it will prompt a message before running the actual command), using the hook `robust-externalize/just-before-writing-files` that we introduced in v2.1:

```
\cacheTikz

\robExtConfigure{
  clean and recompile/.code={%
    \AddToHook{robust-externalize/just-before-writing-files}{%
      \edef\command{rm -f "\robExtAddCachePathAndName{\robExtFinalHash.pdf}" && %
        rm -f "\robExtAddCachePathAndName{\robExtFinalHash.aux}"}%
      \message{You will run the command ``\command''}%
      \message{Press X if you do NOT want to run it, otherwise press ENTER.}%
      \show\def% this is just used to wait a user input
      \immediate\write18{\command}%
    }%
  },%
}

\begin{tikzpicture}<clean and recompile>
  \node[fill=red] {B};
\end{tikzpicture}
```

Note that if you do not want to run a command from LaTeX, or if you are on a version smaller than 2.1, you can also simply print the name of the file in the pdf and remove it manually:

Hello World!
You want to delete the file `robustExternalize/robExt-8D3C716FE8F65C28FBFB9C4C845D659D`.pdf.

```
\begin{CacheMe}{tikzpicture, name output=filetodelete}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{CacheMe}\\

You want to delete the file \texttt{\filetodeleteInCache.pdf}. %
```

4.3.4 Options to configure the inclusion command

The inclusion command is the command that is run to include the cached file back in the pdf (e.g. based on `\includegraphics`). We describe now how to configure this command.

/robExt/**custom include command advanced**={⟨*include command*⟩}   (style, no default)

Sets the command to run to include the compiled file. You can use:
`\robExtAddCachePathAndName{\robExtFinalHash.pdf}`
in order to get the path of the compiled pdf file. Note that we recommend rather to use `custom include command` that automatically checks if the file compiled correctly and that load the `*-out.tex` file if it exists (useful to pass information back to the pdf).

/robExt/**custom include command**={⟨*include command*⟩}   (style, no default)
/robExt/**include command is input**=[⟨*additional code to run*⟩]   (style, no default)

Sets the command to run to include the compiled file, after checking if the file has been correctly compiled and loading `*-out.tex` (useful to pass information back to the pdf). `include command is input`={SOME CODE} is a shortcut for:

```
custom include command={SOME CODE\input{\robExtAddCachePathAndName{\robExtFinalHash.pdf}}},
```

/robExt/do not include pdf   (style, no value)
/robExt/include command is input=additional command   (style, no default)

> `do not include pdf` do not include the pdf. Useful if you only want to compile the file but use it later (note that you should still generate a `.pdf` file, possibly empty, to indicate that the compilation runs smoothly). Equivalent to:
> `custom include command={}` Use `include command is input` (v2.3 and after) in order to specify that you should `\input` the `.pdf` file (note that despite the extension, it should still contain a LaTeX code). You can also specify an optional command to run before. Equivalent to:
> `custom include command={#1\input{\robExtAddCachePathAndName{\robExtFinalHash.pdf}}}`

/robExt/enable manual mode   (style, no value)
/robExt/disable manual mode   (style, no value)
/robExt/enable fallback to manual mode   (style, no value)
/robExt/disable fallback to manual mode   (style, no value)

> If you do or do not want to ask latex to run the compilation commands itself (for instance for security reasons), you can use these commands and run the command manually later:

> The next picture must be manually compiled (see JOBNAME-robExt-compile-missing-figures.sh):
>
> **Manual mode: either compile with -shell-escape or compile:**
> **robustExternalize/robExt-9A58BBFD5C2B6C079891CF68E406829D.tex**
> **via**
> **cd robustExternalize/ && pdflatex -halt-on-error "robExt-9A58BBFD5C2B6C079891CF68E406829D.tex"**
> **or call**
> **bash robust-externalize-robExt-compile-missing-figures.sh**
> **to compile all missing figures.**

```
\robExtConfigure{
  enable manual mode
}

The next picture must be manually compiled %
(see JOBNAME-robExt-compile-missing-figures.sh):\\ %
\begin{tikzpictureC}[baseline=(A.base)]
  \node[fill=red, rounded corners](A){I must be manually compiled.};
  \node[fill=red, rounded corners, opacity=.3,overlay] at (A.north east){I am an overlay text};
\end{tikzpictureC}
```

> The main difference between `manual mode` and `fallback to manual mode` is that `fallback to manual mode` will try to compile the file if `-shell-escape` is enabled, while `manual mode` will never run any command, even if `-shell-escape` is enabled. Note that `enable fallback to manual mode` is available starting from v2.0.

> See section 4.8 for more details.

/robExt/include graphics args   (style, no value)

> By default, the include commands runs `\includegraphics` on the pdf, and possibly raises it if needed. You can customize the arguments passed to `\includegraphics` here.

/robExt/add cache to graphicspath   (style, no value)

> This will add the folder containing the cache in the `\graphicspath{}`. Mostly useful if you need to input a file from the cache that does `\includegraphics{foo}` where `foo` is an image from the cache itself (looking at you gnuplot).

4.3.5 Configuration of the cache

If needed, you can configure the cache:

/robExt/set filename prefix={⟨*prefix*⟩}   (style, no default)

    By default, the files in the cache starts with `robExt-`. If needed you can change this here, or by manually defining `\def\robExtPrefixFilename{yourPrefix-}`.

/robExt/set subfolder and way back={⟨*cache folder*⟩}{⟨*path to project from cache*⟩}   (style, no default)

/robExt/set cache folder and way back={⟨*cache folder*⟩}{⟨*path to project from cache*⟩}   (style, no default)

    By default, the cache is located in `robustExternalize/`, using:
`set cache folder and way back={robustExternalize/}{../}`,
You can customize it the way you want, just be make sure that going to the second arguments after going to the first argument leads you back to the original position, and make sure to terminate the path with a `/` so that `__ROBEXT_WAY_BACK__common_inputs.tex` gives the path of the file `common_inputs.tex` in the root folder (do not write `__ROBEXT_WAY_BACK__/common_inputs.tex` as this would expand to the absolute path `/common_inputs.tex` if you disable the cache folder). Note that both:
`set cache folder and way back`
and
`set subfolder and way back`
are alias, but the first one was introduced in v2.0 as it is clearer, the second one being kept for backward compatibility.

/robExt/no cache folder   (style, no value)

    Do not put the cache in a separate subfolder.

4.3.6 Customize or disable externalization

You might want (sometimes or always) to disable externalization, for instance to use `remember picture` [Point to me if you can], even if you used \cacheTikz:

/robExt/disable externalization   (style, no value)
/robExt/de   (style, no value)
/robExt/disable externalization now   (style, no value)
/robExt/enable externalization   (style, no value)

Enable or disable externalization.

> This figure is not externalized. This way, it can use remember picture.
> This figure is not externalized. This way, it can use remember picture.
> This figure is not externalized. This way, it can use remember picture.

```
% In theory all pictures should be externalized (so remember picture should fail)
\cacheTikz
% But we can disable it temporarily
\begin{tikzpicture}<disable externalization>[remember picture]
  \node[rounded corners, fill=red](A){This figure is not externalized.
    This way, it can use remember picture.};
  \draw[->,overlay] (A) to[bend right] (pointtome);
\end{tikzpicture}\\

% You can also disable it globally/in a group:
{
  \robExtConfigure{disable externalization}

  \begin{tikzpicture}[remember picture]
    \node[rounded corners, fill=red](A){This figure is not externalized.
      This way, it can use remember picture.};
    \draw[->,overlay] (A.west) to[bend left] (pointtome);
  \end{tikzpicture}\\

  \begin{tikzpicture}[remember picture]
    \node[rounded corners, fill=red](A){This figure is not externalized.
      This way, it can use remember picture.};
    \draw[->,overlay] (A.east) to[bend right] (pointtome);
  \end{tikzpicture}
}
```

`disable externalization now` additionally redefines all automatically cached commands and environments to their default value right now (`disable externalization` would only do so when running the `command if no externalization` is run, which should be preferred if possible). This is mostly useful when an automatically cached command cannot be parsed (e.g. you specified the signature `O{}m` but in fact the command expects a more complicated parsing algorithm. For instance, in `tikz` we can omit the brackets, and it might confuse the system if we defined it as:

> The short version of tikz can be confusing.

```
%% Not the recommended way to proceed, but this is for the example
\cacheCommand{tikz}[O{}m]{tikz}
{
  % Needed, or the parser would be confused by the next command
  \robExtConfigure{disable externalization now}
  \tikz\node[fill=green]{The short version of tikz can be confusing.};
}
```

59

Since v2.1 you can also automatically disable externalization if the input contains a given string, for instance `remember picture` (that is not cachable with this library):

Point to me Some text

```
\cacheTikz
\robExtConfigure{
  add to preset={tikzpicture}{
    % \b means "word boundary", and spaces must be escaped
    if matches={remember picture}{disable externalization},
  },
}
\begin{tikzpicture}[remember picture]
  \node[fill=green](my original node){Point to me};
\end{tikzpicture} %
Some text %
\begin{tikzpicture}[overlay, remember picture]
  \draw[->] (0,0) to[bend left] (my original node);
\end{tikzpicture}
```
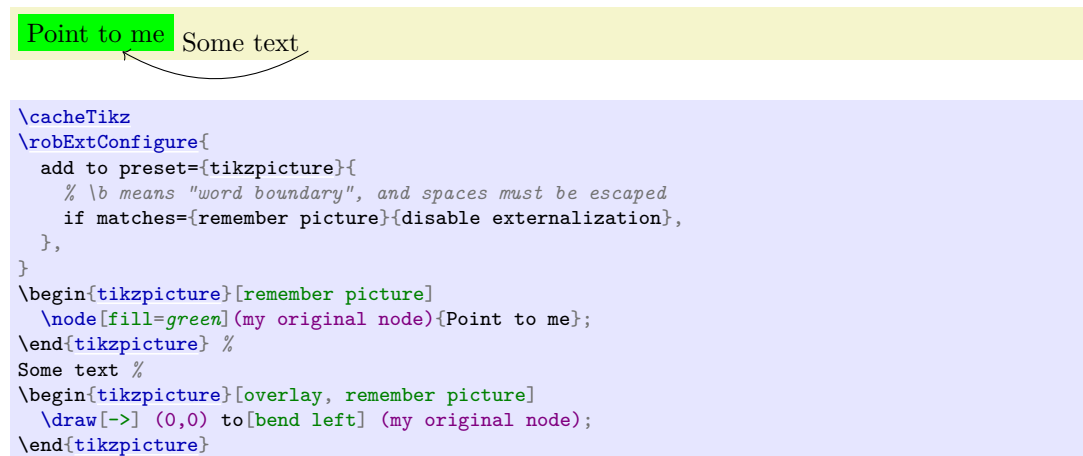
/robExt/command if no externalization   (style, no value)
/robExt/command if externalization   (style, no value)
/robExt/run command if externalization={⟨*code*⟩}   (style, no default)

You can easily change the command to run if externalization is disabled using by setting the **.code** of this key. By default, it is configured as:

```
command if no externalization/.code={%
  \robExtDisableTikzpictureOverwrite\evalPlaceholder{__ROBEXT_MAIN_CONTENT__}%
}
```

Unless you know what you are doing, you should include `\robExtDisableTikzpictureOverwrite` as it is often necessary to avoid infinite recursion when externalization is disabled and the original command has been replaced with a cached version (for instance done by `\cacheTikz`). Note that if you write your own style, try to modify `__ROBEXT_MAIN_CONTENT__` so that it can be included as-it in a document: this way you do not need to change this command. You can also use `command if externalization/.append code={...}` if you want to run a code only if externalization is enabled (this is used for instance by the `forward` functionality), but we advise you rather to use `run command if externalization={...}` as this will also work if the style is executed while `command if externalization` is already running (this might occur when calling the style inside `if matches word`, for instance when forwarding colors).

/robExt/print verbatim if no externalization   (style, no value)

Sets `command if no externalization` to print the verbatim content of `__ROBEXT_MAIN_CONTENT__` if externalization is disabled. Internally, it just sets it to:
`\printPlaceholder{__ROBEXT_MAIN_CONTENT__}`
This is mostly useful when typesetting `__ROBEXT_MAIN_CONTENT__` directly does not make sense (e.g. in python code). This style is used for instance in the `python` preset, allowing us to get:

```
with open("__ROBEXT_OUTPUT_PREFIX__-out.txt", "w") as f:
    for i in range(5):
        f.write(f"Hello {i}, we are on page 60\n")
```

```
\begin{CacheMeCode}{python,
    verbatim output,
    set placeholder eval={__thepage__}{\thepage},
    %% We disable externalization
    disable externalization}
with open("__ROBEXT_OUTPUT_PREFIX__-out.txt", "w") as f:
    for i in range(5):
        f.write(f"Hello {i}, we are on page __thepage__\n")
\end{CacheMeCode}
```

You can also disable the externalization on all elements that use a common preset, for instance you can disable externalization on all `bash` instances (useful if you are on Windows for instance):

```
# $outputTxt contains the path of the file that will be printed via
\verbatiminput
uname -srv > "${outputTxt}"
```

```
\robExtConfigure{
  % bash code will not be compiled (useful on windows for instance)
  add to preset={bash}{
    disable externalization
  },
}
\begin{CacheMeCode}{bash, verbatim output}
# $outputTxt contains the path of the file that will be printed via \verbatiminput
uname -srv > "${outputTxt}"
\end{CacheMeCode}
```

/robExt/execute after each externalization   (style, no value)
/robExt/execute before each externalization   (style, no value)

By doing `execute after each externalization={some code}`, you will run some code after the externalization. This might be practical for instance to update a counter (e.g. the number of pages…) based on the result of the compiled file.

4.3.7 Dependencies

In order to enforce reproducibility, you should tell what are the files that your code depends on, by adding this file as a dependency. This has the advantage that if this file is changed, your code is automatically recompiled. On the other hand, you might not want this behavior (e.g. if this file often changes in a non-important way): in that case, just don't add the file as a dependency (but keep that in mind as you might not be able to recompile your file if you clear the cache if you introduced breaking changes).

/robExt/dependencies={⟨list,of,dependencies⟩}   (style, no default)
/robExt/add dependencies={⟨list,of,dependencies⟩}   (style, no default)
/robExt/reset dependencies   (style, no value)

Set/add/reset the dependencies (you can put multiple files separated by commas). These files should be relative to the main compiled file. For instance, if you have a file `common_inputs.tex` that you want to input in both the main file and in the cached files, that contains, say:
`\def\myValueDefinedInCommonInputs{42}`
then you can add it as a dependency using:

The answer is 42.

```
\begin{CacheMe}{latex,
    add dependencies={common_inputs.tex},
    add to preamble={\input{__ROBEXT_WAY_BACK__common_inputs.tex}}}
  The answer is \myValueDefinedInCommonInputs.
\end{CacheMe}
```

Note that the placeholder `__ROBEXT_WAY_BACK__` contains the path from the cache folder (containing the `.tex` that will be cached) to the root folder.
This way, you can easily input files contained in the root folder.

4.3.8 Forward macros

(since v2.1)
You can also see the tutorial (section 3.7) for some examples. So far, we have seen multiple approaches to use a macro in both the main document and in cached pictures, having different advantages:

- if we define the macro in a file say `common_inputs.tex`, if we include that file in both files **and** if we add it to the list of dependencies (section 4.3.7), then you will have great purity (if `common_inputs.tex` changes, all files are recompiled), but this might be a problem if you often change the file `common_inputs.tex` since you will need to recompile all pictures every time you add a new macro

- If you follow the same procedure, but **without** adding the file in the list of dependencies, then you will save compilation time (no need to recompile everything if you add a macro in `common_inputs.tex`), but you will lose purity (if you change the definition of a macro in that file, you will need to manually specify the list of figures to recompile, e.g. using `recompile`).

- You can also create multiple files like

  `common_inputs_functionality_A.tex`

  and

  `common_inputs_functionality_B.tex`

  and use the first solution (maybe by creating a preset that automatically calls `add to preamble` and `add dependencies`), and only input the appropriate file depending on the set of macro that you need. This gives a tradeoff of above approaches (good purity, with less recompilation), but you will still compile often if you put many macros in a single file (and nobody wants to create one macro per file!).

In this section, we provide a third solution that tries to solve the above issue (purity without frequent recompilations), by allowing the user to forward only a specific macro, with something like `forward=\myMacro`. In order to avoid to manually forward all macros used for each picture, we also provide some functions to automatically detect the macros to forward.

`/robExt/forward`=macro to forward   (style, no default)
`/robExt/fw`=macro to forward   (style, no default)
`/robExt/forward at letter`=macro to forward   (style, no default)

  `forward` (alias `fw`) will forward the definition of a macro to the picture if externalization is enabled:


```
\cacheTikz
\def\myName{Alice}
\begin{tikzpicture}<forward=\myName>
  \node[fill=red]{\myName};
\end{tikzpicture}
```

Note that it works for macro defined with `\def`, `\newcommand` (or alike) and `\NewDocumentCommand` (or other xparse commands). For instance with xparse:

Alice   Bob
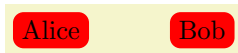
```
\cacheTikz
\NewDocumentCommand{\MyNode}{O{}m}{\node[rounded corners,fill=red,#1]{#2};}
\begin{tikzpicture}<forward=\MyNode>
  \MyNode{Alice}
  \MyNode[xshift=2cm]{Bob}
\end{tikzpicture}
```

and with `\newcommand`:

Alice   Bob

```
\cacheTikz
\newcommand{\MyNodeWithNewCommand}[2][draw,thick]{\node[rounded corners,fill=red,#1]{#2};}
\begin{tikzpicture}<forward=\MyNodeWithNewCommand>
  \MyNodeWithNewCommand{Alice}
  \MyNodeWithNewCommand[xshift=2cm]{Bob}
\end{tikzpicture}
```

Note that this might not work with quite involved macros. Notably, if the macro is defined by calling other macros, you need to also forward these macros.
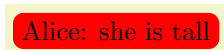
Alice: she is tall

```
\cacheTikz
\def\myName{Alice}
\NewDocumentCommand{\MyNodeWithInnerMacros}{O{}m}{\node[rounded corners,fill=red,#1]{\myName: #2};}
\begin{tikzpicture}<forward=\MyNodeWithInnerMacros,forward=\myName>
  \MyNodeWithInnerMacros{she is tall}
\end{tikzpicture}
```

Really involved macros (e.g. defined inside libraries) might call many other macros, possibly using the @ letter (used only in module code). If you REALLY want, you can forward such macros if you forward **all** macros that are used internally, using `forward at letter` that will automatically wrap the macro inside `\makeatletter...\makeatother`, but you should certainly use `add to preamble={\usepackage{yourlib}}` if you need to do something like that:

Alice

```
\cacheTikz
\makeatletter
\def\module@name{Alice}
\newcommand{\MyNodeWithAtCommands}[2][draw,thick]{\node[rounded corners,fill=red,#1]{\module@name};}
\begin{tikzpicture}<forward at letter=\MyNodeWithAtCommands,forward at letter=\module@name>
  \MyNodeWithAtCommands{Alice}
\end{tikzpicture}
\makeatother
```

/robExt/forward eval=macro to eval and forward   (style, no default)

Evaluate a macro, and defines it using this new value when compiling the function. This might be useful for instance if your macro depends on other macros that you do not want to export:
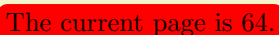
Alice.

```
\def\name{Alice}
\def\fullName{\name}
\begin{tikzpictureC}<forward eval=\fullName>
  \node[rounded corners, fill=red]{\fullName.};
\end{tikzpictureC}
```

/robExt/forward counter=counter to forward   (style, no default)

Forward a counter:

The current page is 64.

```
\begin{tikzpictureC}<forward counter=page>
  \node[rounded corners, fill=red]{The current page is \thepage.};
\end{tikzpictureC}
```

/robExt/forward color=color to forward   (style, no default)
/robExt/auto forward color={⟨*preset*⟩}{⟨*color to forward*⟩}   (style, no default)

`forward color` forwards a color (defined with xcolor). It works with colors defined with `\definecolor`:

A

```
\definecolor{myred}{HTML}{F01122}
\begin{tikzpictureC}<forward color=myred>
  \node[fill=myred]{A};
\end{tikzpictureC}
```

but also with colors defined with `\colorlet`:

B

```
\definecolor{myred}{HTML}{F01122}
\colorlet{myviolet}{blue!50!myred}
\begin{tikzpictureC}<forward color=myviolet>
  \node[fill=myviolet,yshift=-1cm]{B};
\end{tikzpictureC}
```

Note that using `if matches word`, you can also automatically forward colors. This command also scales better than `if matches`: if `if matches` is used to register *N* words, the compilation time will increase for each new `if matches` (since we need to check each time if the string contains the expression). `if matches word`, on the other hand, is more clever, and extracts in a single loop all the words of the string: the running time is therefore independent of the number of times `if matches word` was called. You can call `if matches word` outside of any preset like in:

B

```
\definecolor{myred}{HTML}{F01122}
\colorlet{myviolet}{blue!50!myred}
\colorlet{mypink}{pink!90!orange}
\robExtConfigure{
  if matches word={mypink}{forward color=mypink},
}
\begin{tikzpictureC}
  \node[fill=mypink,yshift=-1cm]{B};
\end{tikzpictureC}
```

And this will automatically allow all presets based on `latex` to forward this color (see also `\definecolorAutoForward` to do that even quicker). You can also simply call it on any preset of your choice like:

B

```
\definecolor{myred}{HTML}{F01122}
\colorlet{myviolet}{blue!50!myred}
\colorlet{mypink}{pink!90!orange}
\robExtConfigure{
  add to preset={latex}{
    if matches word={mypink}{forward color=mypink},
  },
}
\begin{tikzpictureC}
  \node[fill=mypink,yshift=-1cm]{B};
\end{tikzpictureC}
```

or, equivalently:

B

```
\definecolor{myred}{HTML}{F01122}
\colorlet{myviolet}{blue!50!myred}
\colorlet{mypink}{pink!90!orange}
\robExtConfigure{
  auto forward color={tikz}{mypink},
}
\begin{tikzpictureC}
  \node[fill=mypink,yshift=-1cm]{B};
\end{tikzpictureC}
```

/robExt/auto forward   (style, no value)
/robExt/auto forward only macros   (style, no value)
/robExt/auto forward words   (style, no value)

   `auto forward` enables automatic forwarding of macros marked with `*AutoForward` commands or `\configIfMacroPresent` or `if matches word`. This command naively extracts all used macros via a simple regex match and forwards them/execute the style configured in `\configIfMacroPresent` if it exists. It does a similar thing for words registered via `if matches word` or `register word with namespace`. If you want to only forward macros or words, you can use `auto forward only macros` or `auto forward words`.

Recompiled only if MyNode is changed   but not if the (unused) MyGreenNode is changed.
Recompiled only if MyGreenNode is changed   but not if the (unused) MyNode is changed.

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyNode}[2][draw,thick]{\node[rounded corners,fill=red,#1]{#2};}
\newcommandAutoForward{\MyGreenNode}[2][draw,thick]{\node[rounded corners,fill=green,#1]{#2};}

\begin{tikzpicture}
  \MyNode{Recompiled only if MyNode is changed}
  \MyNode[xshift=8cm]{but not if the (unused) MyGreenNode is changed.}
\end{tikzpicture}\\

\begin{tikzpicture}
  \MyGreenNode{Recompiled only if MyGreenNode is changed}
  \MyGreenNode[xshift=8cm]{but not if the (unused) MyNode is changed.}
\end{tikzpicture}
```

`\configIfMacroPresent`{⟨*macro*⟩}{⟨*style to run if macro is present*⟩}

Runs the corresponding style if the macro `macro` is present (make sure to enable `auto forward`).


```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\configIfMacroPresent{\ding}{add to preamble={\usepackage{pifont}}}

\begin{tikzpicture}
  \node[fill=green, circle]{\ding{164}};
\end{tikzpicture}
```

`\newcommandAutoForward`{⟨*macro*⟩}[⟨*nb args*⟩][⟨*optional default value*⟩]{⟨*definition*⟩}[⟨*additional style*⟩]
`\renewcommandAutoForward`{⟨*macro*⟩}[⟨*nb args*⟩][⟨*optional default value*⟩]{⟨*definition*⟩}[⟨*additional style*⟩]
`\providecommandAutoForward`{⟨*macro*⟩}[⟨*nb args*⟩][⟨*optional default value*⟩]{⟨*definition*⟩}[⟨*additional style*⟩]
`\NewDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\RenewDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\ProvideDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\DeclareDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\NewExpandableDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\RenewExpandableDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\ProvideExpandableDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\DeclareExpandableDocumentCommandAutoForward`{⟨*macro*⟩}{⟨*arg spec*⟩}[⟨*additional style*⟩]{⟨*definition*⟩}
`\defAutoForward`{⟨*macro*⟩}[⟨*def-style arg spec*⟩]{⟨*macro*⟩}[⟨*additional style*⟩]
`\genericAutoForward`*[⟨*preset*⟩][⟨*namespace*⟩]{⟨*word*⟩}[⟨*additional style*⟩]{⟨*code to run*⟩}
`\genericAutoForwardStringMatch`*[⟨*preset*⟩]{⟨*string*⟩}[⟨*additional style*⟩]{⟨*code to run*⟩}

`\newcommandAutoForward{\macro}[nb args][default value]{def}[STYLE]`

is like

`\newcommand{\macro}[nb args][default value]{def}`

but it additionally runs

`\configIfMacroPresent{\macro}{forward=\macro,STYLE}`

to automatically forward the macro if it is used. It is really practical to automatically define a macro and forward it:

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyName}{\ding{164} Alice}[add to preamble={\usepackage{pifont}}]
\newcommandAutoForward{\MyNode}[2][draw,thick]{\node[rounded corners,fill=green,#1]{\MyName: #2};}[
  load auto forward macro config=\MyName
]

\begin{tikzpicture}
  \MyNode{Recompiled only if MyNode is changed}
  \MyNode[yshift=-1cm]{or if MyName is changed}
\end{tikzpicture}
```

This works similarly for other commands (note that for `\defAutoForward`, you need to put the arguments in a bracket, like `\defAutoForward\myMacro[#1 and #2]{Hey #1, #2}` for `\def\myMacro#1 and #2{Hey #1, #2}`). `\genericAutoForward` and `\genericAutoForwardStringMatch` are a bit special, as it can be used to run any code assuming that `word`/`string to match` is present (`\genericAutoForward` is more efficient as it does not scale with the number of words, but do not allow spaces etc in the name). For instance, you can use it to automatically forward the code that defines a new tikz style assuming `mystyle` is preset:

A

```
\cacheTikz
\genericAutoForward{mystyle}{
  \tikzset{mystyle/.style={fill=#1!10!white,text=#1!50!black}}
}

\begin{tikzpicture}
  \node[mystyle=green]{A};
\end{tikzpicture}
```

This will also run the code to define it in the current document (practical for instance if some pictures are not externalized), use the stared version of `\genericAutoForward` if you do not want to run the code in the current document. By default, the code is added to the `latex` preset, but you can choose another preset if you want using the optional option. The additional style might be specified to run other style if the matches is true. Note that you should be able to use definitions inside the code (a bug in v2.1 was making this code fail):

Hey Coucou Alice :) $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$

```
\genericAutoForward{mystyle}{
  \tikzset{mystyle/.style={fill=#1!10!white,text=#1!50!black}}
}
\newcommandAutoForward{\myName}{Alice}
\robExtConfigure{
  add to preset={tikz}{
    auto forward, % not needed for genericAutoForward, but for newcommandAutoForward
    if matches word={pmatrix}{add to preamble={\usepackage{amsmath}}}
  }
}

\begin{tikzpicture}
  \node[mystyle=green]{%
    \def\hey#1{Hey #1 :)}% You can define macros inside the cached elements
    \hey{Coucou \myName} %
    $\begin{pmatrix}% Fixing a bug introduced in v2.1, we can also use macros
      A & B\\
      C & D\\
    \end{pmatrix}$
  };
\end{tikzpicture}
```

<div style="background-color:#90EE90;display:inline-block">My style automatically forwarded</div>

```
% space is not allowed in the name with this more efficient version:
\robExtGenericAutoForward{mystyle}{%
  \tikzset{%
    mystyle/.style={fill=#1!30!white,text=#1!40!black},
  }%
}
\begin{tikzpictureC}
  \node[mystyle=green]{My style automatically forwarded};
\end{tikzpictureC}
```

<div style="background-color:#90EE90;display:inline-block">My style with a space in the name automatically forwarded</div>

```
% space is allowed in the name with this slightly less efficient version:
\robExtGenericAutoForwardStringMatch{my style}{%
  \tikzset{%
    my style/.style={fill=#1!30!white,text=#1!40!black},
  }%
}
\begin{tikzpictureC}
  \node[my style=green]{My style with a space in the name automatically forwarded};
\end{tikzpictureC}
```

Note that by default, this forwards elements in the `latex` preset, but you can change it via `preset`. You can also configure additional style to run, and choose another namespace for the word version (practical only if you want to forward different codes to different presets: note that a single namespace is allowed per preset).

\runHereAndInPreambleOfCachedFiles[⟨*preset*⟩]{⟨*code*⟩}

(from v2.3) This will run `code` right now and it will add it in the preamble of the files that use the preset `preset` (defaults to `latex`). This is practical to avoid duplicating the code into `add to preamble`. For instance, if want the configure the font of the current document and of all latex documents at the same time, use:

```
\runHereAndInPreambleOfCachedFiles{
  \usepackage{fontspec}
  \setmainfont{Times New Roman}
}

\robExtConfigure{
  add to preset={latex}{
    use lualatex, % Make sure to compile with lualatex the cached images.
  },
}
```

\definecolorAutoForward{⟨*color*⟩}{⟨*model*⟩}{⟨*value*⟩}[⟨*additional style*⟩]
\colorletAutoForward{⟨*color*⟩}{⟨*value*⟩}[⟨*additional style*⟩]

Define colors like \definecolor and \colorlet, but also runs

`if matches word={COLOR}{forward color=COLOR}`

to automatically forward the color (this will forward it only in latex-based presets). You can provide additional style to run as well in option.

<div style="background-color:red;color:white;display:inline-block">Uses myred</div> <div style="background-color:purple;color:white;display:inline-block">Myviolet</div>

```
\definecolorAutoForward{myred}{HTML}{F01122}
\colorletAutoForward{myviolet}{blue!50!myred}
\begin{tikzpictureC}
  \node[fill=myred,yshift=-1cm]{Uses myred};
\end{tikzpictureC}
\begin{tikzpictureC}
  \node[fill=myviolet,yshift=-1cm]{Myviolet};
\end{tikzpictureC}
```

/robExt/load auto forward macro config={⟨*macro*⟩}   (style, no default)

Like `forward` except that it loads the configuration that would have been loaded if the macro was present in the file. This is mostly useful to say that a macro depends on the style of another macro without copy/pasting the style of the second macro inside the first one.

❤ Alice: Recompiled only if MyNode is changed

❤ Alice: or if MyName is changed

```
\cacheTikz
\robExtConfigure{add to preset={tikz}{auto forward}}

\newcommandAutoForward{\MyName}{\ding{164} Alice}[add to preamble={\usepackage{pifont}}]
\newcommandAutoForward{\MyNode}[2][draw,thick]{\node[rounded corners,fill=green,#1]{\MyName: #2};}[
  load auto forward macro config=\MyName
]

\begin{tikzpicture}
  \MyNode{Recompiled only if MyNode is changed}
  \MyNode[yshift=-1cm]{or if MyName is changed}
\end{tikzpicture}
```

/robExt/if matches={⟨*string*⟩}{⟨*style to apply*⟩}   (style, no default)
/robExt/if matches word={⟨*string*⟩}{⟨*style to apply*⟩}   (style, no default)
/robExt/if matches regex={⟨*latex3 regex*⟩}{⟨*style to apply*⟩}   (style, no default)
/robExt/register word with namespace={⟨*namespace*⟩}{⟨*word*⟩}{⟨*style*⟩}   (style, no default)
/robExt/auto forward words   (style, no value)
/robExt/auto forward words namespace={⟨*namespace*⟩}   (style, no default)

`if matches *` applies the corresponding style if the string (resp. word or regex in LaTeX3 format) matches is matched in the content. `if matches word` is more efficient that `if matches string` since the running time is independent of the number of times `if matches word` is called[7]. `if matches word` make it only match words that are mostly made of consecutive letters and numbers `[A-Za-z0-9]` (more precisely separated by elements in `\robExtWordSeparators`). The regex version can be more expressive, but is significantly slower (it can easily multiply by 2 the compilation time).

Python code

```
1 print(cos(1)+sin(2))
```
Output:
1.4495997326938215

---

[7]Scaling with $O(s)$ instead of $O(sn)$, where s is the size of the string typed in `cacheMe` and n is the number of times `if matches` is used, for instance when replacing multiple strings.

```
\robExtConfigure{
  add to preset={my python}{
    python print code and result,
    % \b is for word border, and ( needs to be escaped in regex
    if matches={cos(}{add import={from math import cos}},
    if matches={sin(}{add import={from math import sin}},
  },
}
\begin{CacheMeCode}{my python}
print(cos(1)+sin(2))
\end{CacheMeCode}
```

With regex:

<div style="border:2px solid red; border-radius:8px;">

Python code

```
1  print(cos(1)+sin(2))
```

Output:
1.4495997326938215

</div>

```
\robExtConfigure{
  add to preset={my python}{
    python print code and result,
    % \b is for word border, and ( needs to be escaped in regex
    if matches regex={\b cos\(}{add import={from math import cos}},
    if matches regex={\b sin\(}{add import={from math import sin}},
  },
}
\begin{CacheMeCode}{my python}
print(cos(1)+sin(2))
\end{CacheMeCode}
```

This can also be practical to disable caching if some pictures uses `remember picture` (which is not supported by this library):


```
\cacheTikz
\robExtConfigure{
  add to preset={tikzpicture}{
    % \b means "word boundary", and spaces must be escaped
    if matches={remember picture}{disable externalization},
  },
}
\begin{tikzpicture}[remember picture]
  \node[fill=green](my original node){Point to me};
\end{tikzpicture} %
Some text %
\begin{tikzpicture}[overlay, remember picture]
  \draw[->] (0,0) to[bend left] (my original node);
\end{tikzpicture}
```

Note that to make `if matches word` even more efficient, you can call it before the preset. It will "register" the corresponding word with `register word` (itself a shortcut for `register word with namespace` with an empty namespace), and call `auto forward words` (shortcut for `auto forward words namespace={}`) in the `latex` preset (or in the current preset if we are already in a preset) to forward all words registered in the corresponding namespace (empty by default). Unless you want to forward different words to different presets, these more advanced functions involving namespaces should not be really useful for the end user.

4.3.9 Pass compiled file to another template

It can sometimes be handy to use the result of a previous cached file to cache another file, or to do anything else (e.g. it can also be practical to debug an issue). `name output` can be used to do that

`/robExt/name output`={⟨*macro name*⟩}   (style, no default)

`name output=foo` will create two global macros `\foo` and `\fooInCache`: `\foo` expands to the prefix of the files created in the template like `robExt-somehash`, and `\fooInCache` also adds the cache folder like `robustExternalize/robExt-somehash`. You can then use `set placeholder eval` to send it to another cached file. It is then your role to add the extension, usually `.tex` to get the source (even if the source is a python file), `.pdf` to get the pdf, `-out.tex` to get the file that is loaded before the import, `-out.txt` if you wanted to make it compatible with `verbatim output` (this list is not exhaustive as each script might decide to create a different output file). Here is a demo:

Hello World!

The prefix is `robExt-533423D6079B5BC1502646EBDF055F42` and with the cache folder it is in:
`robustExternalize/robExt-533423D6079B5BC1502646EBDF055F42`.
It this can be helpful for instance to debug, as you can inspect the source:

```
\documentclass[,margin=0cm]{standalone}
 \usepackage {tikz}
% most packages must be loaded before hyperref
% so we typically want to load hyperref here

% some packages must be loaded after hyperref

\begin{document}%
%% We save the height/depth of the content by using a savebox:
\newwrite\writeRobExt%
\immediate\openout\writeRobExt=\jobname-out.tex%
%
\newsavebox\boxRobExt%
\begin{lrbox}{\boxRobExt}%
  \begin {tikzpicture}[baseline=(A.base)] \node [draw,rounded corners,fill=pink!60](A){Hel
\end{lrbox}%
\usebox{\boxRobExt}%
\immediate\write\writeRobExt{%
  \string\def\string\robExtWidth{\the\wd\boxRobExt}%
  \string\def\string\robExtHeight{\the\ht\boxRobExt}%
  \string\def\string\robExtDepth{\the\dp\boxRobExt}%
}%
%

\end{document}
```

but it is also practical to define a template based on the previously cached files:

A cached file can use result from another cached file: Hello World! Hello World!

```
\begin{CacheMe}{tikzpicture, do not add margins, name output=mycode}[baseline=(A.base)]
  \node[draw,rounded corners,fill=pink!60](A){Hello World!};
\end{CacheMe}\\[3mm]

The prefix is \texttt{\mycode} and with the cache folder it is in:\\
\texttt{\mycodeInCache}.\\
It this can be helpful for instance to debug, as you can inspect the source:
\verbatiminput{\mycodeInCache.tex}
but it is also practical to define a template based on the previously cached files:\\

\begin{CacheMe}{tikzpicture, set placeholder eval={__previous__}{\mycode.pdf}}
  \node[rounded corners, fill=green!50]{A cached file can use result from another cached file:
    \includegraphics[width=2cm]{__previous__}\includegraphics[width=2cm]{__previous__}};
\end{CacheMe}
```

Note that if you do not want to display the first cached file, you can use `do not include pdf` to hide it.

4.3.10 Compile in parallel

(introduced in v2.1, read below on how to do it manually on v2.0)

/robExt/compile in parallel=nb of pictures to compile normally   (style, default 0)
/robExt/compile in parallel after=nb pictures to compile normally   (style, default 0)

These two commands are alias. Typing in the preamble:

```
\robExtConfigure{
  compile in parallel
}
```

will cause the cached elements to be compiled in parallel (this requires two compilations of the main project). For this to work out of the box, you need to have `xargs` installed (on windows, install the lightweight GNU On Windows (Gow) `https://github.com/bmatzelle/gow` to get `xargs`). Typing `compile in parallel after=3` will start the compilation in parallel only if you have more than 3 new elements to compile, and compile the first 3 elements normally (useful when you work on one picture at a time since you do not need to compile twice the document).

/robExt/compile in parallel command={⟨*command to run*⟩}   (style, no default)
/robExt/compile in parallel with xargs=number threads   (style, default 16%)
/robExt/compile in parallel with gnu parallel=–jobs option   (style, default 200%)

You can customize the compilation command to run via something like:

```
\robExtConfigure{
  compile in parallel command={
    parallel --jobs 200\% :::: '\jobname-\robExtAddPrefixName{compile-missing-figures.sh}
  },
}
```

The other styles are predefined commands to compile with xargs (by default, already installed in linux, on Windows you can get it by installing the lightweight GNU On Windows (Gow) `https://github.com/bmatzelle/gow`) or gnu parallel. You can use these commands directly like:

```
\robExtConfigure{
  compile in parallel with gnu parallel
}
```

or use the optional parameter to configure the number of threads (xargs starts with 16 threads by default, and gnu parallel takes twice the number of CPU threads available), like in:

```
\robExtConfigure{
  compile in parallel with xargs=64,% Compiles with 64 processes at a time
}
```

`/robExt/if unix`={⟨*style to run if running linux or unix*⟩} (style, no default)
`/robExt/if windows`={⟨*style to run if windows*⟩} (style, no default)

Since windows does not pack `xparse` by default, you might want to enable parallel compilation only in Linux/MacOs. This can be done using the above flags, like:

```
\robExtConfigure{
  if unix={
    compile in parallel
  },
}
```

(note that `unix` is understood as "non windows", i.e. would run in Unix/Linux/MacOs)

NB: in version smaller than v2.1 (or if you prefer the manual method), you can instead run it in manual mode and use for instance GNU `parallel` (or any tool of you choice) to run all commands in the file `\jobname-robExt-compile-missing-figures.sh` in parallel. To do that:

- First add `\robExtConfigure{enable manual mode}` in your file

- Create the cache folder, e.g. using `mkdir robustExternalize`

- Compile your file, with, e.g. `pdflatex yourfile.tex` (in my benchmark: 4.2s)

- This should create a file `yourfile-robExt-compile-missing-figures.sh`, with each line containing a compilation command like this (the command should be compatible with Windows, Mac and Linux, if not, let me know on github):

  `cd robustExternalize/ && pdflatex -halt-on-error "robExt-4A806941E86C6B657A0CB3D160CFFF3E.tex`

  IMPORTANT: LaTeX might render **multiple times** the same picture if it is inside some particular environments (align, tables...), so this file can contain duplicates if you inserted cached data inside. In version 2.1 this is solved, but on older versions, to avoid running the same command twice, **be sure to remove duplicates!** On Linux, this can be done by piping the file into `sort` and `uniq` (if your files can depend on previously compiled elements as explained in section 4.3.9, you should additionally preserve the order of the lines[8]). Since this is anyway a rather advanced use case, we will not consider that case for simplicity.

- Run all these command in parallel, for instance on Linux you can install `parallel`, and run the following command (`--bar` displays a progress bar):

  `cat yourfile-robExt-compile-missing-figures.sh | sort | uniq | parallel --bar --jobs 200%`

  (note that since v2.1, it is not necessary to remove duplicates using `sort` and `uniq` since the file contains no more duplicates) In my benchmark, it ran during 52s, so 1.6x faster than the original compilation without caching)

- Recompile the original document with `pdflatex yourfile.tex`

In my benchmark, the total time is 2x4.2s + 52s = 60s, so 1.5 times faster than a normal command (of course, this depends a lot on the preamble of the file that you compile, since the loading time of the file is the main bottleneck for the first compilation; the advantage here is that it is easy to include only the necessary things in the preamble of cached pictures, possibly creating different presets if it is easier to manage it this way).

---

[8]`https://unix.stackexchange.com/questions/194780/remove-duplicate-lines-while-keeping-the-order-of-the-lines`

4.3.11 Compile a template to compile even faster

Long story short: you can compile even faster (at least 1.5x in our tests, but we expect this to be more visible on larger tests due to the loading time of the preamble of the main document that takes most of the time of the compilation) by compiling presets, but beware that you will not be able to modify the placeholders except `add to preamble` with the default compiler we provide. The gain comes from the fact that instead of trying to find all placeholders to replace in the string, we can directly replace a few hard-coded placeholders to save time (yes, LaTeX is not really efficient so it can make a difference...).

`/robExt/new compiled preset`={⟨*preset options to compile*⟩}{⟨*runtime options*⟩}   (style, no default)

`/robExt/compile latex template`   (style, no value)

Compile a preset by creating a new placeholders that removes all placeholders except `__ROBEXT_MAIN_CONTENT__`, `__ROBEXT_MAIN_CONTENT_ORIG__` and `__ROBEXT_LATEX_PREAMBLE__`. `preset options to compile` should be sure that the template, compilation command and include command placeholders contain the final version: this can be easily done for latex presets by adding `compile latex template` at the end of `preset options to compile`:.

> *HeyBob!*

```
% We create a latex-based preset and compile it
\robExtConfigure{
  new preset={templateZX}{
    latex,
    add to preamble={
      \usepackage{tikz}
      \usepackage{tikz-cd}
      \usepackage{zx-calculus}
    },
    %% possibly add some dependencies
  },
  % We compile it into a new preset
  new compiled preset={compiled ZX}{templateZX, compile latex template}{},
}

% we use that preset automatically for ZX environments
\cacheEnvironment{ZX}{compiled ZX}
\cacheCommand{zx}{compiled ZX}

% Usage: (you can't use placeholders except for the preamble, trade-off of the compiled template)
\begin{ZX}<add to preamble={\def\sayHey#1{Hey #1!}}>
  \zxX{\sayHey{Bob}}
\end{ZX}
```

Explanations: in v2.0, we made substantial improvements in order to improve significantly the compilation time (our benchmark went from 20s (v1.0) to only 5.77s (v2.0)), but you might want to make this even faster: with a compiled preset, you can improve it further, experimentation showed an additional improvement factor of 1.4x (final compilation time was 4s in our tests, reminder: without the library it would take 1mn25s). The main bottleneck in term of time is the expansion of the placeholders (that allows great flexibility, but can add a significant time). At a high level, we keep track in a list of all declared placeholders, and loop over them to replace each of them until the string is left unmodified by a full turn. This is simple to implement, does not need to assume any shape for the placeholder... but not extremely efficient in LaTeX where string and list manipulations are costly.

We can play on multiple parameters to speed up the process:

- Disable completely the placeholders and only replace a small amount of fixed placeholders in a fixed order and stop expanding the placeholder. This is what we do by default when we compile a preset: it is the quickest solution, but you cannot use arbitrary placeholders.

- Start the expansion using a fixed (preset-dependent) list of placeholders, check if some templates are still present (assuming that templates must contain at least two consecutive underscores `__`), if not stop, otherwise go to the normal (less efficient) placeholder replacement. It is what we do for instance in the latex preset, and gives a great balance between efficiency and flexibility. Details can be found in section 4.10.

- Stop iterating over all placeholders, only choose the one meaningful (why would I care about python placeholders in latex?). For this, we introduce an import system, where you can create placeholders that are not added to the "global" list of placeholders, and import them where you want. In practice, you should not really need to use it unless you do more advanced stuff as we already take care of setting it properly, but details are in section 4.2.4.

Another option could be to rewrite the code to search directly the list of placeholders in the string, but it is certainly not trivial to do in latex, would require even more constraints on the shape of placeholders, and would certainly still no more efficient than the compiled stuff. But anyway the already implemented solutions already give fairly good performances.

4.4 Default presets

We provide by default some presets for famous languages (for now LaTeX and python).

4.4.1 All languages

First, here are a few options that are available irrespective of the used language.

/robExt/set includegraphics options={⟨*options*⟩}	(style, no default)
/robExt/add to includegraphics options={⟨*options*⟩}	(style, no default)

Set/add options to the `\includegraphics` run when inserting the pdf (by the default include command). By default it is empty, but the `latex` preset sets it to:
`trim=__ROBEXT_LATEX_TRIM_LENGTH__ __ROBEXT_LATEX_TRIM_LENGTH__`
`__ROBEXT_LATEX_TRIM_LENGTH__ __ROBEXT_LATEX_TRIM_LENGTH__`
in order to remove the margin added in the standalone package options, which is needed to display overlay texts.

/robExt/verbatim output	(style, no value)

Shortcut for:
```
custom include command={%
  \evalPlaceholder{%
    __ROBEXT_VERBATIM_COMMAND__{%
      __ROBEXT_CACHE_FOLDER____ROBEXT_OUTPUT_PREFIX__-out.txt}%
  }%
},
```
i.e. instead of printing the pdf we print the content of the file `__ROBEXT_OUTPUT_PREFIX__-out.txt` using the command in `__ROBEXT_VERBATIM_COMMAND__`, that defaults to `\verbatiminput`:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

```
\begin{CacheMeCode}{python, verbatim output}
with open("__ROBEXT_OUTPUT_PREFIX__-out.txt", "w") as f:
    for i in range(5):
        f.write(f"Hello {i}\n")
\end{CacheMeCode}
```

(new in v2.0) These two (alias) commands allow you to print the compilation command, the dependency file (useful to be sure you tracked all dependencies), and the source file:

Command: (run in folder robustExternalize/)

```
pdflatex -halt-on-error "robExt-583C5935D363C10E50E66F4A3387957A.tex"
```

Dependencies:

```
    command,pdflatex -halt-on-error "__ROBEXT_SOURCE_FILE__"
    68C1A3F4A0563203D18D71B08F24393B,
    648E35E8624D2EFA2983FE62024FB1FD,common_inputs.tex
```

Source (in robustExternalize/robExt-583C5935D363C10E50E66F4A3387957A.tex):

```
    \documentclass[,margin=30cm]{standalone}
     \input {../common_inputs.tex}
    % most packages must be loaded before hyperref
    % so we typically want to load hyperref here

    % some packages must be loaded after hyperref

    \begin{document}%
    %% We save the height/depth of the content by using a savebox:
    \newwrite\writeRobExt%
    \immediate\openout\writeRobExt=\jobname-out.tex%
    %
    \newsavebox\boxRobExt%
    \begin{lrbox}{\boxRobExt}%
      The answer is \myValueDefinedInCommonInputs .%
    \end{lrbox}%
    \usebox{\boxRobExt}%
    \immediate\write\writeRobExt{%
      \string\def\string\robExtWidth{\the\wd\boxRobExt}%
      \string\def\string\robExtHeight{\the\ht\boxRobExt}%
      \string\def\string\robExtDepth{\the\dp\boxRobExt}%
    }%
    %

    \end{document}
```

```
\begin{CacheMe}{latex,
    add dependencies={common_inputs.tex},
    add to preamble={\input{__ROBEXT_WAY_BACK__common_inputs.tex}},
    debug
  }
  The answer is \myValueDefinedInCommonInputs.
\end{CacheMe}
```

4.4.2 LaTeX and Ti*k*Z

The latex preset is used to cache any LaTeX content, like tikz pictures. Note that as of today, it supports overlay content out of the box (if the overlay is more than 30cm long, you might want to customize a placeholder), but not images that need to use remember picture.

76

This style sets the template `__ROBEXT_LATEX__` and the compilation command:
`__ROBEXT_LATEX_COMPILATION_COMMAND__`
(cf section 4.5 for details), and adds a number of styles described below, to easily configure the most common options. You can use it as follows:

> The next picture is cached My node that respects baseline. and you can see that overlay and depth works.

```
The next picture is cached %
\begin{CacheMe}{latex, add to preamble={\usepackage{tikz}}}
  \begin{tikzpicture}[baseline=(A.base)]
    \node[fill=red, rounded corners](A){My node that respects baseline.};
    \node[fill=red, rounded corners, opacity=.3,overlay] at (A.north east){I am an overlay text};
  \end{tikzpicture}
\end{CacheMe} and you can see that overlay and depth works.
```

To see how to create your own preset or automatically load a library, see section 4.6.

The next options can be used after calling the `latex` style:

`/robExt/latex/use latexmk`	(style, no value)
`/robExt/latex/use lualatex`	(style, no value)
`/robExt/latex/use xelatex`	(style, no value)

Use latexmk/lualatex/xelatex to compile. It is a shortcut for:
`set placeholder={__ROBEXT_LATEX_ENGINE__}{yourfavoriteengine}`

`/robExt/latex/set latex options`={⟨*latex options*⟩}	(style, no default)
`/robExt/latex/add to latex options`={⟨*latex options*⟩}	(style, no default)

Set/add elements to the set of latex options of the `\documentclass` (it will automatically add a comma before if you add an element). Internally it sets `__ROBEXT_LATEX_OPTIONS__`. By default, it sets:
`margin=__ROBEXT_LATEX_TRIM_LENGTH__` (where `__ROBEXT_LATEX_TRIM_LENGTH__` is defined as 30cm by default) in order to add a margin that will be trimmed later in the `\includegraphics`. This is useful not to cut stuff displayed outside of the bounding box (overlays).

`/robExt/latex/set documentclass`={⟨*documentclass*⟩}	(style, no default)

Set the documentclass of the document (defaults to `standalone`). Internally, it sets the placeholder `__ROBEXT_LATEX_DOCUMENT_CLASS__`.

`/robExt/latex/set preamble`={⟨*code of preamble*⟩}	(style, no default)
`/robExt/latex/add to preamble`={⟨*code of preamble*⟩}	(style, no default)
`/robExt/latex/set preamble hyperref`={⟨*code of preamble*⟩}	(style, no default)
`/robExt/latex/add to preamble hyperref`={⟨*code of preamble*⟩}	(style, no default)
`/robExt/latex/set preamble after hyperref`={⟨*code of preamble*⟩}	(style, no default)
`/robExt/latex/add to preamble after hyperref`={⟨*code of preamble*⟩}	(style, no default)

Set/add element to the preamble (defaults to `standalone`). Internally, it sets the placeholder `__ROBEXT_LATEX_PREAMBLE__`. The variations `hyperref` and `after hyperref` are used to put stuff after the preambule, as `hyperref` typically needs to be loaded last (or nearly): packages that must be loaded after `hyperref` like `cref` can be added via `add to preamble after hyperref`.

`/robExt/latex/do not wrap code`	(style, no value)

By default, the main content is wrapped into a box in order to measure its depth to properly set the baseline. If you do not want to do this wrapping, you can set this option. Internally, it is a shortcut for:
`set placeholder={__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__}{__ROBEXT_MAIN_CONTENT__}`

IMPORTANT: note that it means that you might need to adapt your code to take into account the fact that they are inside a box (I don't know of any other solution to compute the depth, but it does not mean that there is none).

/robExt/latex/in command   (style, no value)

Sets `__ROBEXT_MAIN_CONTENT` to point to `__ROBEXT_MAIN_CONTENT_ORIG__` to remove any wrapping of the user input, for instance made by tikzpicture.

/robExt/tikz   (style, no value)
/robExt/tikzpicture   (style, no value)

`tikz` loads `latex` and then adds `tikz` to the preamble. `tikzpicture` first load `tikz` and wraps the main content within `\begin{tikzpicture}` and `\end{tikzpicture}` using:

```
set placeholder={__ROBEXT_MAIN_CONTENT__}{%
  \begin{tikzpicture}__ROBEXT_MAIN_CONTENT_ORIG__\end{tikzpicture}%
}
```

so that the user does not need to type it. See for instance the introduction for examples of use.

4.4.3 Python

We provide support for python:

/robExt/python   (style, no value)

Load the `python` preset (inspect `__ROBEXT_PYTHON__`) for details on the exact template, but note that this template might be subject to changes. We also provide a few helper functions:

- `write_to_out(text)` writes `text` to the `*-out.tex` file that will be loaded automatically before running the include function

- `parse_args()` is a function that returns a dictionary mapping some keys to values depending on the called arguments: for instance, if you call the python file with `python script key1 value1 key2 value2`, then the dictionary will map `key1` to `value1` and `key2` to `value2`. You might like this in conjunction with commands presented in section 4.3.3. Note that if you place placeholders in your code, you might not need this, but this is used if you plan to use your script outside of this library.

- `get_cache_folder()` outputs the cache folder.

- `get_file_base()` outputs the prefix of all files that should be created by this script, that looks like `robExt-somehash`.

- `get_current_script()` returns the current script.

- `get_filename_from_extension(extension)` outputs the prefix `robExt-somehash` concatenated with the extension. You often need this function to get the path of a file that your script is creating, for instance, `get_filename_from_extension("-out.txt")` is the path `*-out.txt` of the file that is read by `verbatim output`.

- `get_verbatim_output()` returns `get_filename_from_extension("-out.txt")`

- `finished_with_no_error()` creates the pdf file if it does not exists (to certify that the compilation ran without issues). The template automatically runs this function at the end.

We demonstrate its usage on a few examples:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

78
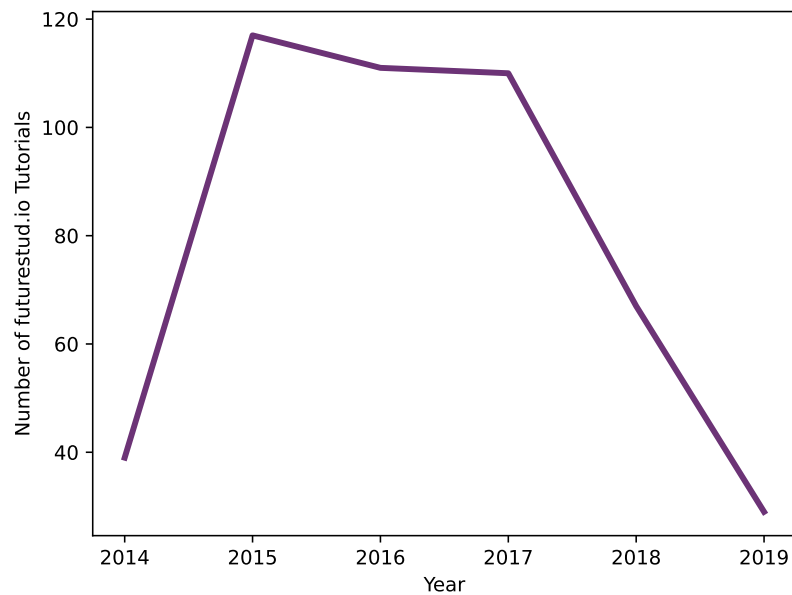
Figure 3: Image generated with python.

```
\begin{CacheMeCode}{python, verbatim output}
with open(get_verbatim_output(), "w") as f:
    for i in range(5):
        f.write(f"Hello {i}\n")
\end{CacheMeCode}
```

Importantly: you do not want to indent the whole content of CacheMeCode, or the spaces will also appear in the final code.

You can also generate some images. This code will produce the image in fig. 3:

```
\begin{CacheMeCode}{python, set includegraphics options={width=.8\linewidth}}
import matplotlib.pyplot as plt
year = [2014, 2015, 2016, 2017, 2018, 2019]
tutorial_count = [39, 117, 111, 110, 67, 29]
plt.plot(year, tutorial_count, color="#6c3376", linewidth=3)
plt.xlabel('Year')
plt.ylabel('Number of futurestud.io Tutorials')
plt.savefig("__ROBEXT_OUTPUT_PDF__")
\end{CacheMeCode}
```

Note that by default, the executable called `python` is run. It seems like on windows `python3` is not created and only `python` exists, while on linux the user can choose whether `python` should point to `python3` or `python2` (on NixOs, I directly have `python` pointing to `python3`, and in ubuntu, you might need to install `python-is-python3` or create a symlink, as explained here). In any case, you can customize the name of the executable by setting something like:

`\setPlaceholder{__ROBEXT_PYTHON_EXEC__}{python3}`

or using the style `force python3` that forces `python3`.

/robExt/python print code and result (style, no value)

This is a demo style that can print a python code and its result.

```
\begin{CacheMeCode}{python print code and result, set title={The for loop}}
for name in ["Alice", "Bob"]:
    print(f"Hello {name}")
\end{CacheMeCode}
```

You can set `__ROBEXT_PYTHON_TCOLORBOX_PROPS__` the options of the tcolorbox, `__ROBEXT_PYTHON_CODE_MESSAGE__` and `__ROBEXT_PYTHON_RESULT_MESSAGE__` which are displayed before the corresponding block, `__ROBEXT_PYTHON_LSTINPUT_STYLE__` which contains the default lstinput style and `__MY_TITLE__` (cf `set title`) that contains the title of the box. Make sure to have the following packages to use the default styling:

```
\usepackage{pythonhighlight}
\usepackage{tcolorbox}
```

/robExt/python/add import   (style, no value)

(since v2.1) Add an import statement in the first part of the file.

```
\robExtConfigure{
  add to preset={my python}{
    python print code and result,
    add import={from math import cos},
    add import={from math import sin},
  },
}
\begin{CacheMeCode}{my python}
print(cos(1)+sin(2))
\end{CacheMeCode}
```

4.4.4 Bash

We provide a basic bash template, that sets:

```
set -e
outputTxt="__ROBEXT_OUTPUT_PREFIX__-out.txt"
outputTex="__ROBEXT_OUTPUT_PREFIX__-out.tex"
outputPdf="__ROBEXT_OUTPUT_PDF__"
```

in order to quit when an error occurs, and to define two variables containing the path to the `pdf` file and to the file that is read by the `verbatim output` setting (that just apply a `\verbatiminput` on that file). Finally, it also creates the file `outputPdf` with `touch` in order to notify that the compilation succeeded.

In practice:

```
Linux 5.15.90 #1-NixOS SMP Tue Jan 24 06:22:49 UTC 2023
```

```
\begin{CacheMeCode}{bash, verbatim output}
# $outputTxt contains the path of the file that will be printed via \verbatiminput
uname -srv > "${outputTxt}"
\end{CacheMeCode}
```

4.4.5 Verbatim text

Sometimes, it might be handy to write the text to a file and use it somehow. This is possible using `verbatim text`, that defaults to calling `\verbatiminput` on that file:

```
def some_verbatim_fct(a):
    # See this is a verbatim code where I can use the % symbol
    return a % b
```

```
\begin{CacheMeCode}{verbatim text}
def some_verbatim_fct(a):
    # See this is a verbatim code where I can use the % symbol
    return a % b
\end{CacheMeCode}
```

You can also call `verbatim text no include`: it will not include the text, but it sets a macro `\robExtPathToInput` containing the path to the input file. Use it the way you like! For instance, we define here a macro `codeAndResult` that prints the code and runs it (we use a pretty printer from pgf, so you need to load `\usepackage{tikz}\input{pgfmanual-en-macros.tex}` to use it). It is what we use right now in this documentation for verbatim blocks like here. You can obtain a simpler version using:

```
We will input the file robustExternalize/robExt-DDA097E3F2A45DB958F5A00BFAFF9B93.tex:

Demo % with percent

This file contains:

\NewDocumentCommand{\testVerbatim}{+v}{
\begin{flushleft}\ttfamily%
#1
\end{flushleft}}
\testVerbatim{Demo % with percent}
```

```
\begin{CacheMeCode}{verbatim text no include}
\NewDocumentCommand{\testVerbatim}{+v}{
\begin{flushleft}\ttfamily%
#1
\end{flushleft}}
\testVerbatim{Demo % with percent}
\end{CacheMeCode}
We will input the file \robExtPathToInput{}:
\input{\robExtPathToInput}
This file contains:
\verbatiminput{\robExtPathToInput}
```

You might also like to use `name output=yourfile` that will create two macros `\yourfile` and `\yourfileInCache`, equal respectively to the prefix `robExt-somehash` and `pathOfCache/robExt-somehash`.

4.4.6 Gnuplot

We also provide support for gnuplot since v2.3.

/robExt/gnuplot	(style, no value)
/robExt/gnuplot/set terminal={⟨*terminal and options*⟩}	(style, no default)
/robExt/gnuplot/pdf terminal=pdf terminal options	(style, no default)
/robExt/gnuplot/tikz terminal=tikz terminal options	(style, no default)

Loads the `gnuplot` preset. By default, it uses the `pdf` terminal:
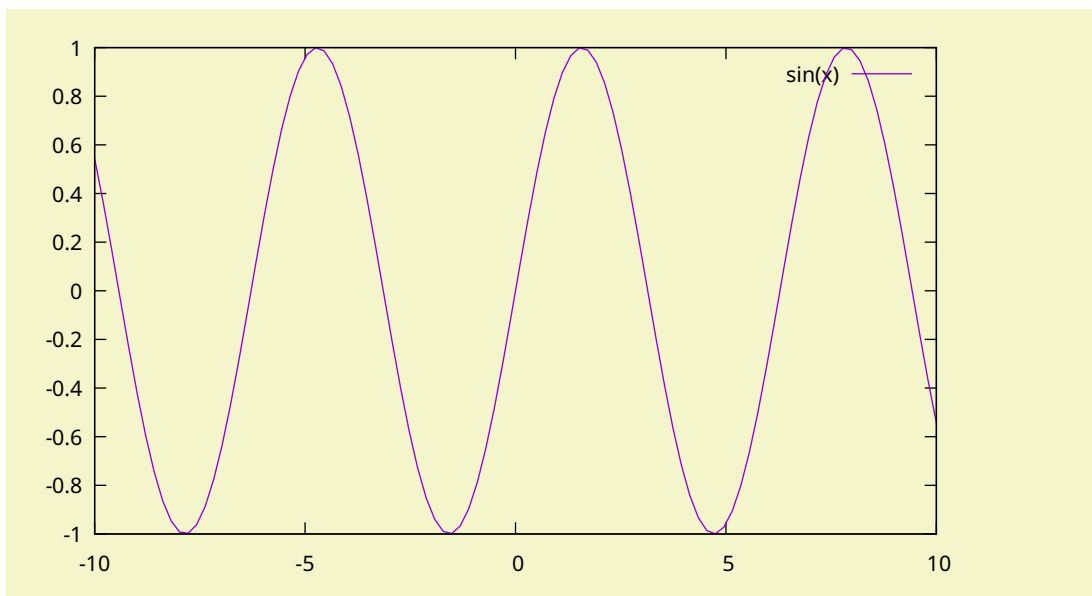

```
\begin{CacheMeCode}{gnuplot}
plot sin(x)
\end{CacheMeCode}
```

WARNING: gnuplot seems to be somehow sensible to indentation, notably to tabulation. So do NOT add any indentation in front of the code, or this will be copied to the source code, and therefore might disturb gnuplot.

You can also specify the gnuplot options to the pdf terminal, like the size, using:
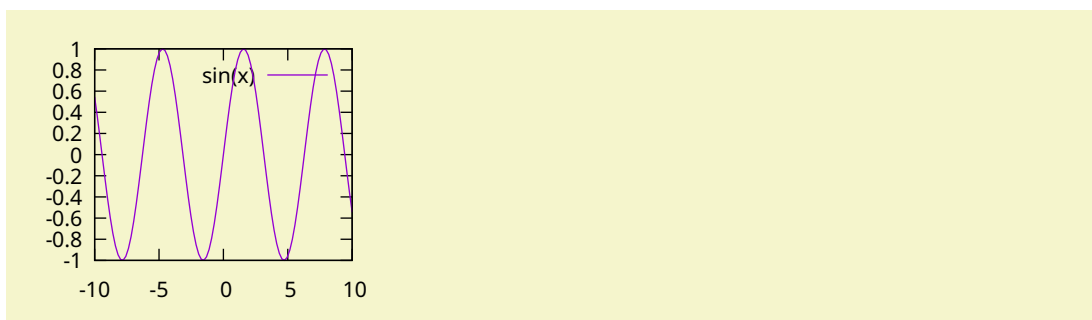

```
\begin{CacheMeCode}{gnuplot,pdf terminal={size 5cm,4cm}}
plot sin(x)
\end{CacheMeCode}
```

You can also change the options of the `includegraphics` command used to include the image, using as usual:
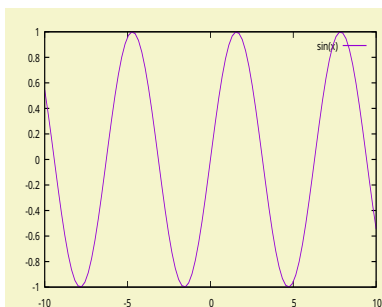
```
\begin{CacheMeCode}{gnuplot,set includegraphics options={width=5cm,height=4cm}}
plot sin(x)
\end{CacheMeCode}
```

You can also use other terminals using the `set terminal` option. For instance, you can use the png terminal and configure its size using:
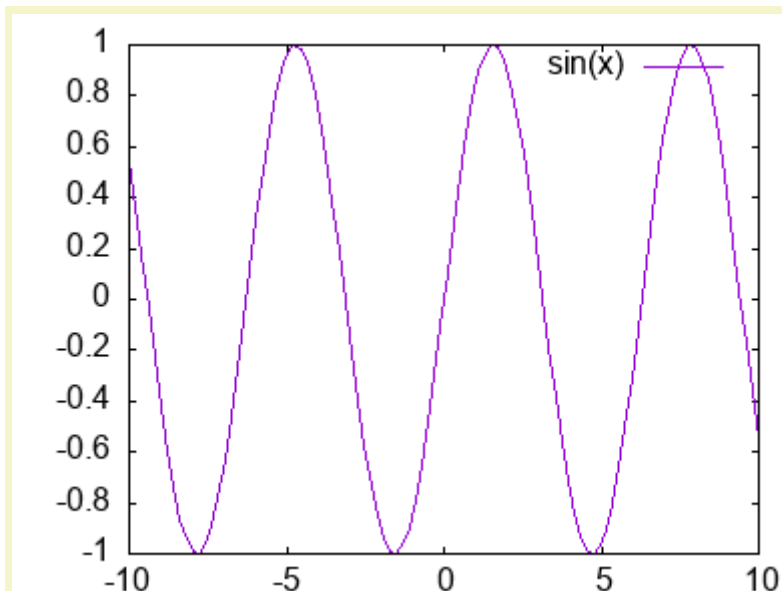

```
\begin{CacheMeCode}{gnuplot,set terminal={png size 400,300}}
plot sin(x)
\end{CacheMeCode}
```

The tikz terminal is a bit special since we do not want to use `\includegraphics` but `\input` to include the file. You can use it like this, just make sure:

- to have gnuplot installed with lua support. With nix you can get it with:
  `nix-shell -p '(gnuplot.override { withLua = true; })'`
- Add `\usepackage{gnuplot-lua-tikz}` after making sure to run
  `gnuplot -e "set terminal tikz createstyle"` once to create the `.sty` file.
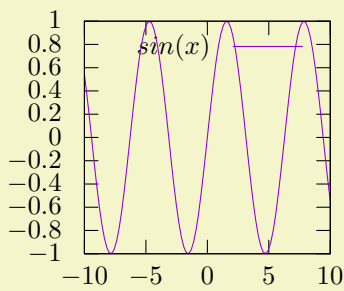
```
% \usepackage{gnuplot-lua-tikz} %% Generate with gnuplot -e "set terminal tikz createstyle"
\begin{CacheMeCode}{gnuplot,tikz terminal={size 5cm,4cm}}
plot sin(x)
\end{CacheMeCode}
```

(Note that if you do not provide any option to `tikz terminal`, it will just use the default value in gnuplot, you can also add any other option to the terminal here.) By default, we only cache the gnuplot compilation, but the `\input` is not cached (so gnuplot will not be run again the next time, but tikz will run every time). But it can also be cached, just add `cache tikz` for that:
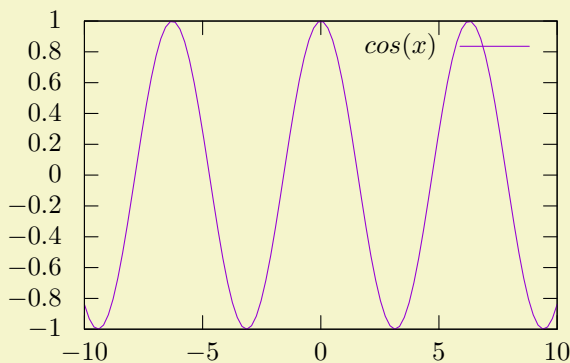

```
\robExtConfigure{
  add to preset={tikz}{
    % Run gnuplot -e "set terminal tikz createstyle" in the robustExternalize folder first
    % to create this file:
    add to preamble={\usepackage{gnuplot-lua-tikz}},
  },
  add to preset={gnuplot}{
    tikz terminal/.append style={
      cache tikz,
    },
  },
}
\begin{CacheMeCode}{gnuplot, tikz terminal={size 8cm,5cm}}
plot cos(x)
\end{CacheMeCode}
```

We also automatically add `__ROBEXT_WAY_BACK__` to the `loadpath`, this way you do NOT need to type `plot "__ROBEXT_WAY_BACK__yourfile.dat"` if the data is in the main folder. Note however that you need to add this file to the list of dependencies, otherwise it will not be recompiled if the file changes:

84
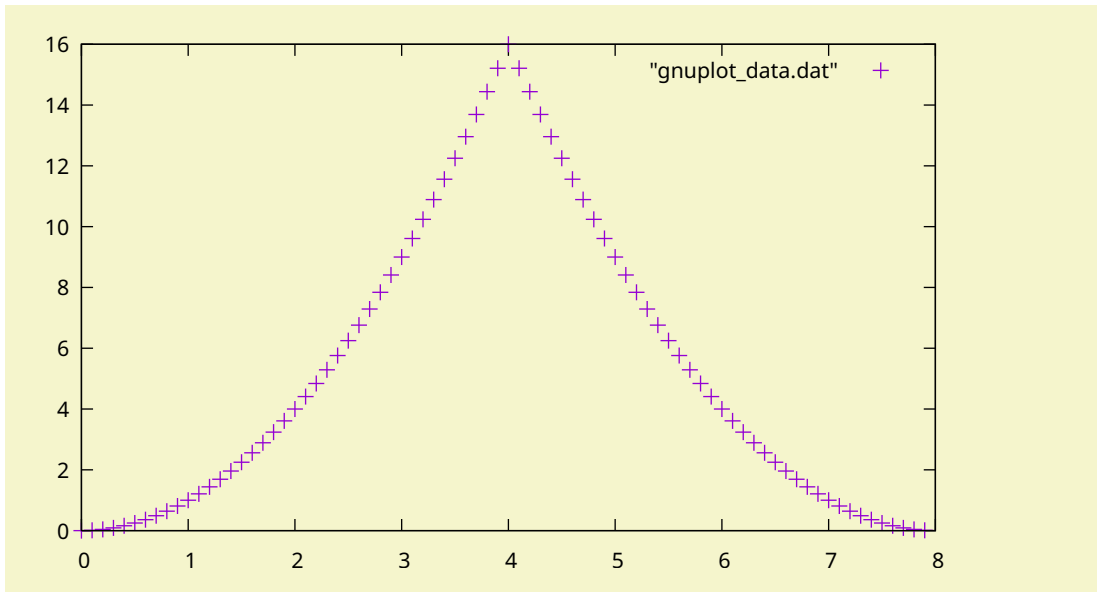
```
\begin{CacheMeCode}{gnuplot, dependencies={gnuplot_data.dat}}
plot "gnuplot_data.dat"
\end{CacheMeCode}
```

You can also use the current text width when using the tikz terminal using something like:
`tikz terminal/.expanded={size \lenToCm{\textwidth},\lenToCm{.3\textwidth}}`
Full example:
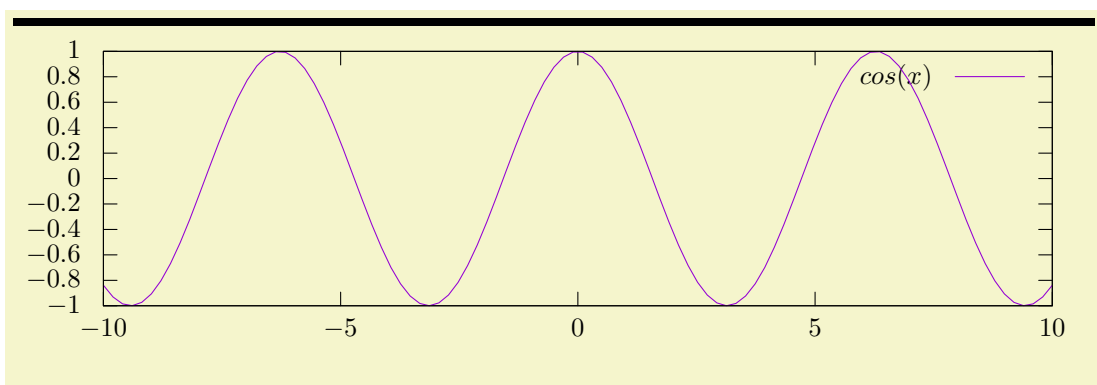

```
% Just to check size:
\noindent\rule{\textwidth}{1mm}\\%
\begin{CacheMeCode}{gnuplot, tikz terminal/.expanded={size \lenToCm{\textwidth},\lenToCm{.3\textwidth}}}
plot cos(x)
\end{CacheMeCode}
```

Similar to the tikz terminal, you can also use the `cairolatex` terminal like this:
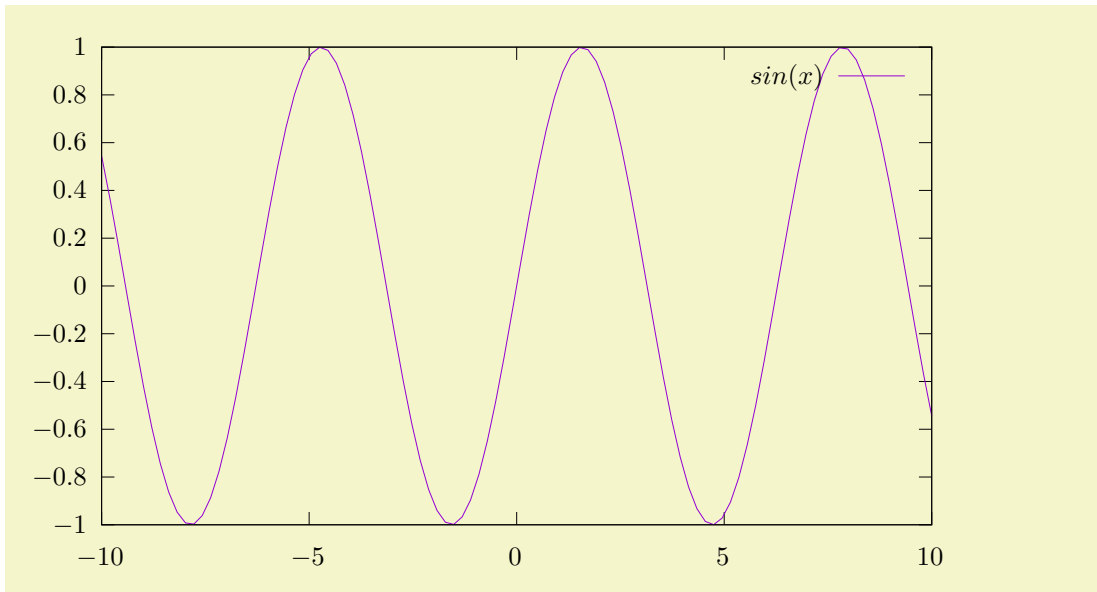
```
\begin{CacheMeCode}{gnuplot,cairolatex terminal}
plot sin(x)
\end{CacheMeCode}
```

As before, you can also specify the size, for instance it term of the current textwidth:
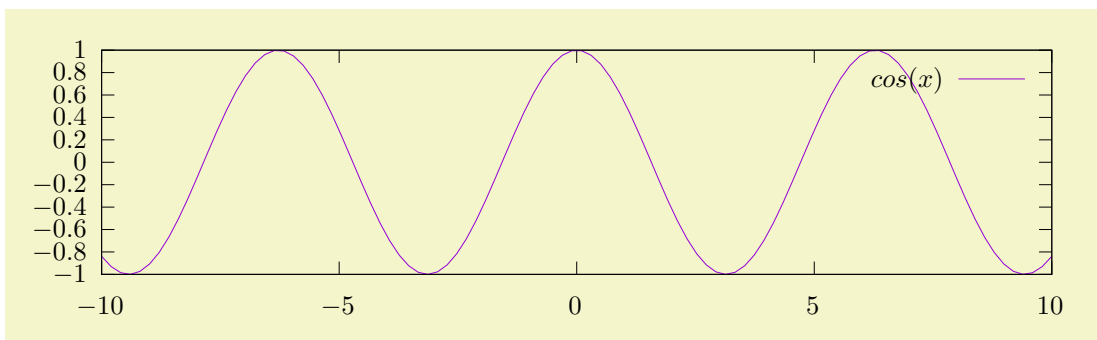

```
\begin{CacheMeCode}{gnuplot, cairolatex terminal/.expanded={size \lenToCm{\textwidth},\lenToCm{.3\textwidth}}}
plot cos(x)
\end{CacheMeCode}
```

\robExtGpgetvar{⟨*variable name*⟩}

Like \gpgetvar{} from gnuplot-lua-tikz but does not produce an error if the variable does not exist yet (mostly useful if you compile in parallel as shown here[9]). The first time, you will get a message like:

Please recompile to load variable: test_variable

and later this will compile fine:

---

[9] https://github.com/leo-colisson/robust-externalize/issues/17

is 5.
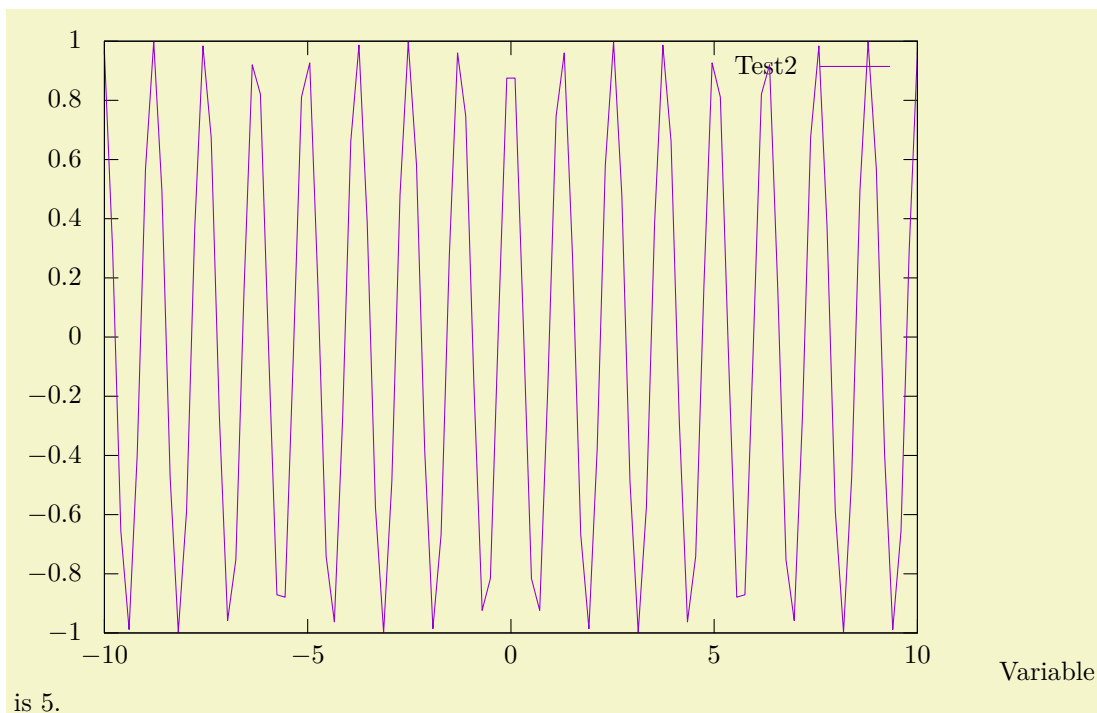
```
\begin{CacheMeCode}{gnuplot, tikz terminal={providevars test_variable}}
test_variable = 5
plot cos(test_variable * x) title "Test2"
\end{CacheMeCode}
Variable is \robExtGpgetvar{test_variable}.
```

4.4.7 Custom languages

To add support to new languages, see section 3.13.2 for an example.

4.5 List of special placeholders and presets

This library defines a number of pre-existing placeholders, or placeholders playing a special role. We list some of them in this section. All placeholders created by this library start with `__ROBEXT_`. Note that you can list all predefined placeholders (at least those globally defined) using `\printImportedPlaceholdersExceptDefaults` (note that some other placeholders might be created directly in the style set right before the command, and may not appear in this list if you call it before setting the style).

4.5.1 Generic placeholders

We define two special placeholders that should be defined by the user (possibly indirectly, using presets offered by this library):

- `__ROBEXT_TEMPLATE__` is a placeholder that should contain the code of the file to compile.

- `__ROBEXT_MAIN_CONTENT_ORIG__` is a placeholder containing the text typed by the user, automatically set by `CacheMe`, `CacheMeCode` etc. You rarely need to deal with this placeholder directly since `__ROBEXT_MAIN_CONTENT__` will typically point to it and add some necessary wrapping.

- `__ROBEXT_MAIN_CONTENT__` is a placeholder that might be used inside `__ROBEXT_TEMPLATE__`, that points by default to `__ROBEXT_MAIN_CONTENT_ORIG__` and that contains the content that should be typed inside the document. For instance, this might be a tikz picture, a

python function without the import etc. Note that it is often used to wrap the text of the user `__ROBEXT_MAIN_CONTENT_ORIG__`: for instance, the `tikzpicture` preset adds the `\begin{tikzpicture}` around the user code automatically: this way we do not need to edit the command to disable externalization.

- `__ROBEXT_COMPILATION_COMMAND__` contains the compilation command to run to compile the file (assuming we are in the cache folder).

We also provide a number of predefined placeholders in order to get the name of the source file etc... Note that most of these placeholders are defined (and/or expanded inplace) late during the compilation stage as one needs first to obtain the hash of the file, and therefore all dependencies, the content of the template etc.

- `__ROBEXT_SOURCE_FILE__` contains the path of the file to compile (containing the content of `__ROBEXT_TEMPLATE__`) like `robExt-somehash.tex`, relative to the cache folder (since we always go to this folder before doing any action, you most likely want to use this directly in the compilation command).

- `__ROBEXT_OUTPUT_PDF__` contains the path of the pdf file produced after the compilation command relative to the cache folder (like `robExt-somehash.pdf`). Even if you do not plan to output a pdf file, you should still create that file at the end of the compilation so that this library can know whether the compilation succeeded.

- `__ROBEXT_OUTPUT_PREFIX__` contains the prefix that all newly created file should follow, like `robExt-somehash`. If you want to create additional files (e.g. a picture, a video, a console output etc...) make sure to make it start with this string. It will not only help to ensure purity, but it also allows us to garbage collect useless files easily.

- `__ROBEXT_WAY_BACK__` contains the path to go back to the main project from the cache folder, like `../` (internally it is equals to the expanded value of `\robExtPrefixPathWayBack`).

- `__ROBEXT_CACHE_FOLDER__` contains the path to the cache folder. Since most commands are run from the cache folder, this should not be really useful to the user.

You can also use these placeholders to customize the default include function:

- `__ROBEXT_INCLUDEGRAPHICS_OPTIONS__` contains the options given to `\includegraphics` when loading the pdf

- `__ROBEXT_INCLUDEGRAPHICS_FILE__` contains the file loaded by `\includegraphics`, defaults to `\robExtAddCachePathAndName{\robExtFinalHash.pdf}`, that is itself equivalent to `__ROBEXT_CACHE_FOLDER____ROBEXT_OUTPUT_PDF__` or `__ROBEXT_CACHE_FOLDER____ROBEXT_OUTPUT_PREFIX__.pdf`.

4.5.2 Placeholders related to LaTeX

Some placeholders are reserved only when dealing with LaTeX code:

- `__ROBEXT_LATEX__` is the main entrypoint, containing all the latex template. It internally calls other placeholders listed below.

- `__ROBEXT_LATEX_OPTIONS__`: contains the options to compile the document, like `a4paper`. Empty by default.

- `__ROBEXT_LATEX_DOCUMENT_CLASS__`: contains the class of the document. Defaults to `standalone`.

- `__ROBEXT_LATEX_PREAMBLE__`: contains the preamble. Is empty by default.

- `__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__`: content inside the `document` environment. It will wrap the actual content typed by the user `__ROBEXT_MAIN_CONTENT__` around a box to compute its depth. If you do not want this behavior, you can set `__ROBEXT_LATEX_MAIN_CONTENT_WRAPPED__` to be equal to `__ROBEXT_MAIN_CONTENT__`. It calls internally `__ROBEXT_LATEX_CREATE_OUT_FILE__` and `__ROBEXT_LATEX_WRITE_DEPTH_TO_OUT_FILE__` to do this computation.

- `__ROBEXT_LATEX_CREATE_OUT_FILE__` creates a new file called `\jobname-out.tex` and open it in the handle called `\writeRobExt`

- `__ROBEXT_LATEX_WRITE_DEPTH_TO_OUT_FILE__` writes the height, depth and width of the box `\boxRobExt` into the filed opened in `\writeRobExt`.

- `__ROBEXT_LATEX_COMPILATION_COMMAND__` is the command used to compile a LaTeX document. It uses internally other placeholders:

- `__ROBEXT_LATEX_ENGINE__` is the engine used to compile the document (defaults to `pdflatex`)

- `__ROBEXT_LATEX_COMPILATION_COMMAND_OPTIONS__` contains the options used to compile the document (defaults to `-shell-escape -halt-on-error`)

4.5.3 Placeholders related to python

- `__ROBEXT_PYTHON_EXEC__` contains the python executable (defaults to `python`) used to compile

- `__ROBEXT_PYTHON__` contains the python template

- `__ROBEXT_PYTHON_IMPORT__` can contain import statements

- `__ROBEXT_PYTHON_MAIN_CONTENT_WRAPPED__` is used to add all the above functions. You can set it to `__ROBEXT_MAIN_CONTENT__` if you do not want them

- `__ROBEXT_PYTHON_FINISHED_WITH_NO_ERROR__` is called at the end to create the pdf file even if it is not created, you can set it to the empty string if you do not want to do that.

4.5.4 Placeholders related to bash

- `__ROBEXT_BASH_TEMPLATE__` contains the bash template. By default, it sets `set -e`, creates `outputTxt`, `outputTex` and `outputPdf` pointing to the corresponding files, and it created the pdf file at the end.

- `__ROBEXT_SHELL__` contains the shell (defaults to `bash`).

4.6 Customize presets and create your own style

Note that you can define your own presets simply by creating a new pgf style (please refer to tikz-pgf's documentation for more details). For instance, we defined the `tikz` and `tikzpicture` presets using:

```
\robExtConfigure{
  new preset={tikz}{
    latex,
    add to preamble={\usepackage{tikz}},
  },
  new preset={tikzpicture}{
    tikz,
    set placeholder={__ROBEXT_MAIN_CONTENT__}{\begin{tikzpicture}__ROBEXT_MAIN_CONTENT_ORIG__\end{tikzpicture}},
  },
}
```

in order to automatically load `tikz` and add the surrounding `tikzpicture` when needed (note that the style is always loaded **after** the definition of `__ROBEXT_MAIN_CONTENT_ORIG__`, so in theory you could also modify it directly even if it is not recommended). You can also customize an existing style by adding stuff to it using `add to preset` (or `.append style` but make sure to double the hashes). For instance, here, we add the `shadows` library to the `tikz` preset by default:

See, tikz's style now packs the `shadows` library by default:
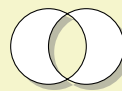
```
\robExtConfigure{
  add to preset={tikz}{
    add to preamble={\usetikzlibrary{shadows}},
  },
}
See, tikz's style now packs the |shadows| library by default: %
\begin{CacheMe}{tikzpicture}[[even odd rule]
  \filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{CacheMe}
```

4.7 Cache automatically a given environment

It might be handy to cache automatically a given environment: we already provide:
\cacheTikz
to cache all tikz pictures (unless externalization is disabled), but we also provide tools to handle arbitrary environments.

\robExtExternalizeAllTikzpicture[⟨*preset for tikz*⟩][⟨*preset for tikzpicture*⟩][⟨*delimiters*⟩]
\cacheTikz[⟨*preset for tikz*⟩][⟨*preset for tikzpicture*⟩][⟨*delimiters*⟩]
/robExt/cache tikz   (style, no value)
/robExt/cache tikz 2 args={⟨*tikz preset*⟩}{⟨*tikzpicture preset*⟩}   (style, no default)
/robExt/cache tikz 3 args={⟨*tikz preset*⟩}{⟨*tikzpicture preset*⟩}{⟨*delimiters cacheMe*
    options*⟩}   (style, no default)

(`cache tikz*` from 2.3, simply style alias) This will automatically cache all tikz pictures (both are alias, \cacheTikz is available from v2.0). Since v2.0, we added the delimiters options and we allow to specify custom presets for `tikz` and `tikzpicture`, and we parse \tikz as well (the `tikz` preset is now used by \tikz by default while the `tikzpicture` preset is used by `tikzpicture`). Note that we add an additional optional argument to the tikz picture via its first argument delimited by `delimiters` (defaults to `<>`) to specify preset options, which can for instance be practical to disable externalization on individual pictures. Cf. section 4.3.6 to see an example of use.

See that this syntax can be safely used. (cached)   Hello Alice! (cached)   Foo (not cached)

```
\cacheTikz

\def\whereIAm{(not cached)}
\robExtConfigure{
  add to preset={tikz}{
    add to preamble={
      \def\whereIAm{(cached)}
    },
  },
}

\tikz \node[fill=pink, rounded corners]{See that this syntax can be safely used. \whereIAm};

\tikz<add to preamble={\def\sayHello#1{Hello #1!}}> \node[fill=green, rounded
corners]{\sayHello{Alice} \whereIAm};

\tikz<disable externalization> \node[fill=green, rounded corners]{Foo \whereIAm};
```

Here is an example to specify arguments via parenthesis:

See that this syntax can be safely used. (cached)   Hello Alice! (cached)   Foo (not cached)

```
% the first two arguments are the default presets used by \tikz and \tikzpicture
\cacheTikz[tikz][tikzpicture][()]

\def\whereIAm{(not cached)}
\robExtConfigure{
  add to preset={tikz}{
    add to preamble={
      \def\whereIAm{(cached)}
    },
  },
}

\tikz \node[fill=pink, rounded corners]{See that this syntax can be safely used. \whereIAm};

\tikz(add to preamble={\def\sayHello#1{Hello #1!}}) \node[fill=green, rounded
corners]{\sayHello{Alice} \whereIAm};

\tikz(disable externalization) \node[fill=green, rounded corners]{Foo \whereIAm};
```

You can also change the default style loaded for tikz and tikzpicture (note that you might prefer to modify `tikz` directly using `add to preset`), but you need v2.1 (buggy in v2.0):

In tikz H   In tikzpicture HH

```
% You might prefer to modify tikz directly using "add to preset"
\cacheTikz[tikz, add to preamble={\def\hello{H}},][tikzpicture,
  add to preamble={\def\hello{HH}},
]

\tikz[baseline=(A.base)] \node(A){In tikz \hello}; %
\begin{tikzpicture}[baseline=(A.base)]
  \node(A){In tikzpicture \hello};
\end{tikzpicture}
```

Note that the `cache tikz` styles (the variation just provide default values) is useful for instance if you only want to locally enable `cache tikz` for the `\input` of a style (this way you can cache the output of a cached thing…). For instance, `gnuplot` with a tikz terminal will compile into a tex file that will be inputted. But unless `\cacheTikz` is enabled, the inputted picture will not be cached (i.e. tikz will need to run). If you still want to cache it without enabling `\cacheTikz` globally, you can load this style in the `gnuplot` style, or do something like:

\robExtDisableTikzpictureOverwrite

This is useful to temporarily reset the current environment to their original value (since v2.0 it also resets other environments, not just `tikzpicture`). This must typically be called at the beginning of `command if no externalization` to avoid infinite recursion if you redefine it, but expect for this case the user is not expected to use this option.

Let us say that you want to cache all elements of a given environment, like `minipage` or `zx-calculus` pictures (another package of mine):

\cacheEnvironment{⟨delimiters⟩}{⟨name environment⟩}{⟨default preset options⟩}

This will automatically cache the corresponding environment (but note that you still need to define it in the preamble of the cached files, for instance by loading the appropriate package):

A ZX figure and the same but aligned on the second line .
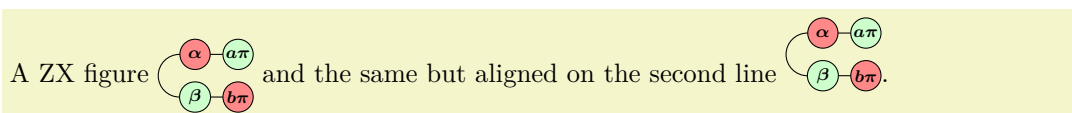
```
\cacheEnvironment{ZX}{latex, add to preamble={\usepackage{zx-calculus}}}
A ZX figure %
\begin{ZX}
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
\end{ZX} and the same but aligned on the second line %
\begin{ZX}[mbr=2]
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
\end{ZX}.
```

You can of course configure them using globally defined configuration, but you can also provide arguments to a single picture using the delimiters `delimiters` that default to `<...>` as the optional argument, like:
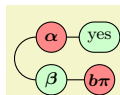

```
\cacheEnvironment{ZX}{latex, add to preamble={\usepackage{zx-calculus}}}
\begin{ZX}<add to preamble={\usepackage{amsmath}}> % amsmath provides \text
  \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{yes}} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
\end{ZX}
```

If you do not like `<>` or if your command already have this parameter, you can change it, for instance to get parens as delimiters, use:
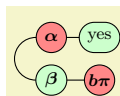

```
\cacheEnvironment[()]{ZX}{latex, add to preamble={\usepackage{zx-calculus}}}
\begin{ZX}(add to preamble={\usepackage{amsmath}}) % amsmath provides \text
  \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{yes}} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
\end{ZX}
```

otherwise you must provide some presets arguments, possibly empty:
`\begin{someweirdenv}<><args to someweirdenv>`

You can also use this argument to disable externalisation on a per-picture basis:
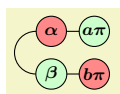

```
\cacheEnvironment{ZX}{latex, add to preamble={\usepackage{zx-calculus}}}
\begin{ZX}<disable externalization> % amsmath provides \text
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
\end{ZX}
```

\cacheCommand[⟨*delimiter*⟩]{⟨*name command*⟩}[⟨*xparse signature*⟩]{⟨*default preset options*⟩}

This will automatically cache the corresponding command, in a similar way to environment. It work exactly the same, except that we (usually) need to specify the signature of the command or it is impossible to know where the command stops. If the command is defined using xparse-compatible commands like `\NewDocumentCommand`, this detection is automatically done. Otherwise, you need to specify it in the xparse format: long story short, `O{foo}` is an optional argument with default value foo, `m` is a mandatory argument.

92

First, let us see an example with a command defined using xparse:

A ZX figure ⬡ and the same but aligned on the second line ⬡.
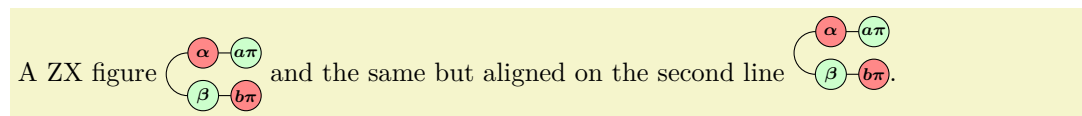
```
% We cache the command \zx that is defined with NewDocumentCommand: no need to specify the signature
\cacheCommand{zx}{latex, add to preamble={\usepackage{zx-calculus}}}
A ZX figure %
\zx{
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
} and the same but aligned on the second line %
\zx[mbr=2]{
  \zxX{\alpha} \rar \ar[d,C] & \zxZ*{a\pi} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
}.
```

You can of course configure them using globally defined configuration, but you can also provide arguments to a single picture using the delimiters that default to <> as the optional argument, like:
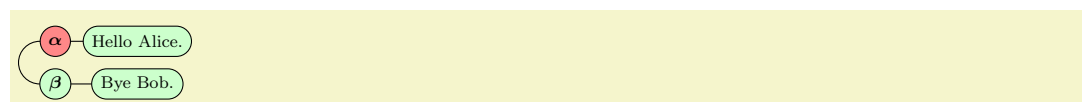
```
\cacheCommand{zx}{latex, add to preamble={\usepackage{zx-calculus}\def\hello#1{Hello #1.}}}
\zx<add to preamble={\usepackage{amsmath}\def\bye#1{Bye #1.}}>[mbr=1]{ % amsmath provides \text
  \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{\hello{Alice}}} \\
  \zxZ{\beta} \rar           & \zxZ{\text{\bye{Bob}}}
}
```

or with parens as delimiters:

```
\cacheCommand[()]{zx}{latex, add to preamble={\usepackage{zx-calculus}}}
\zx(add to preamble={\usepackage{amsmath}})[mbr=1]{ % amsmath provides \text
  \zxX{\alpha} \rar \ar[d,C] & \zxZ{\text{yes}} \\
  \zxZ{\beta} \rar           & \zxX*{b\pi}
}
```

Here is an example with a command not defined with \NewDocumentCommand (we also show how you can use the optional argument to disable externalisation on a per-picture basis):

Optional argument 1: default value, Argument 2: second arg, Argument 3: last arg (I am cached).
Optional argument 1: default value, Argument 2: second arg, Argument 3: last arg (I am not cached).

93

```
\def\fromWhere{I am not cached}
\newcommand{\myMacroNotDefinedWithNewDocumentCommand}[3][default value]{
  Optional argument 1: #1,
  Argument 2: #2,
  Argument 3: #3 (\fromWhere).
}
\cacheCommand{myMacroNotDefinedWithNewDocumentCommand}[O{default value}mm]{
  latex,
  add to preamble={%
    \def\fromWhere{I am cached}
    \newcommand{\myMacroNotDefinedWithNewDocumentCommand}[3][default value]{
      Optional argument 1: #1,
      Argument 2: #2,
      Argument 3: #3 (\fromWhere).
    }
  }
}

\myMacroNotDefinedWithNewDocumentCommand{second arg}{last arg}\\
\myMacroNotDefinedWithNewDocumentCommand<disable externalization>{second arg}{last arg}
```

Defining its own macro without repeating its definition. Note that when you define yourself a macro function, the above structure might not be optimal as you need to define the macro in the main document (in case you disable the externalization) and in the template. One option to avoid repeated code is to write the definition in a common input file (see section 3.4), but you might prefer to keep everything in a single file. In that case, you can simply wrap your function into `cacheMe` yourself. Say you want your function to draw a circle like, `\tikz \draw[fill=#2] (0,0) circle [radius=#1];`, then wrap in inside `CacheMe`:

You can externalize it ◯ or disable externalization ⬤ .

```
% D<>{} is an optional argument enclosed in <>, we use this to pass arguments to CacheMe
\NewDocumentCommand{\drawMyCircle}{D<>{}O{2mm}m}{
  \begin{CacheMe}{tikz, #1}
    \tikz \draw[fill=#3] (0,0) circle [radius=#2];
  \end{CacheMe}
}

You can externalize it \drawMyCircle{pink} %
or disable externalization \drawMyCircle<disable externalization>[4mm]{red}.
```

Manually wrap a command. In some more advanced cases, if you do not know the definition of the command and if `\cacheCommand` does not work (e.g. the number of mandatory argument depends on the first argument), you can wrap it manually. To do so, first copy first your command using `\DeclareCommandCopy{\robExtCommandOrigNAMEOFCOMMAND}{\NAMEOFCOMMAND}` and run `\robExtAddToCommandResetList{NAMEOFCOMMAND}` (this is needed to disable externalization), and override the command by wrapping it into a cache. Make sure to add `\def\robExtCommandOrigName{NAMEOFCOMMAND}` at the beginning of the function or `disable externalization` will not work:

Original command: ⍺–β. This is cached ⍺–β, you can add options like ⍺–Hey and we can disable the externalization: ⍺–Hey.

```
Original command: \zx{\zxZ{\alpha} \rar & \zxX{\beta}}. %
\DeclareCommandCopy{\robExtCommandOrigzx}{\zx}% used to recover the original function
\robExtAddToCommandResetList{zx}% let us know that this function should be reset when externalization is disabled
\DeclareDocumentCommand{\zx}{D<>{}O{}m}{%
  \def\robExtCommandOrigName{zx}%
  \begin{CacheMe}{latex, add to preamble={\usepackage{zx-calculus}}, #1}%
    \zx[#2]{#3}%
  \end{CacheMe}%
}
This is cached \zx<>{\zxZ{\alpha} \rar & \zxX{\beta}}, you can add options like %
\zx<add to preamble={\usepackage{amsmath}}>{\zxZ{\alpha} \rar & \zxX{\text{Hey}}} %
and we can disable the externalization: %
\zx<disable externalization>{\zxZ{\alpha} \rar & \zxX{\text{Hey}}}.
```

You can do a similar trick for environment, by using instead

```
\NewEnvironmentCopy{robExtEnvironmentOrigNAMEENV}{NAMEENV}
\robExtAddToEnvironmentResetList{NAMEENV}
\def\robExtEnvironmentOrigName{NAMEENV}
```

4.8   Operations on the cache

Every time we compile a document, we create automatically a bunch of files:

- the cache is located by default in the `robustExternalize` folder. Feel free to remove this folder if you want to completely clear the cache (but then you need to recompile everything). See below if you want to clean it in a better way.

- `\jobname-robExt-all-figures.txt` contains the list of all figures contained in the document. Mostly useful to help the script that remove other figures.

- `robExt-remove-old-figures.py` is a python script that will remove all cached files that are not used anymore. Just run `python robExt-remove-old-figures.py` to clean it. You will then see the list of files that the script wants to remove: make sure it does not remove any important data, and press "y". Note that it will search for all files that look like `*robExt-all-figures.txt` to see the list of pictures that are still in use, and by default it will only remove the images in the `robustExternalize` folder that start with `robExt-`. If you change the path of the cache or the prefix, edit the script (should not be hard to do).

- `\jobname-robExt-compile-missing-figures.sh` contains a list of commands that you need to run to compile the images not yet compiled in the cache (this list will only be created if you enable the manual compilation mode).

- `\jobname-robExt-tmp-file-you-can-remove.tmp` is a temporary file. Feel free to remove it.

We go over some of these scripts.

4.8.1   Cleaning the cache

You might want to clean the cache. Of course you can remove all generated files, but if you want to keep the picture in use in the latest version of the document, we provide a python script (automatically generated in the root folder) to do this. Just install python3 and run:

```
python3 robExt-remove-old-figures.py
```

(on windows, the executable might be called `python`) You will then be prompted for a confirmation after providing the list of files that will be removed.

4.8.2   Listing all figures in use

After the compilation of the document, a file `robExt-all-figures.txt` is created with the list of the `.tex` file of all figures used in the current document.

4.8.3 Manually compiling the figures

When enabling the manual mode (useful if we don't want to enable `-shell-escape`):

```
\robExtConfigure{
  enable manual mode
}
```

or the fallback to manual mode (it will only enable the manual mode if `-shell-escape` is disabled):

```
\robExtConfigure{
  enable fallback to manual mode,
}
```

the library creates a file `JOBNAME-robExt-compile-missing-figures.sh` that contains the instructions to build the figures that are not yet in the cache (each line contains the compilation command to run). On Linux (or on Windows with bash/cygwin/… installed, it possibly even work out of the box without) you can easily execute them using:

```
bash JOBNAME-robExt-compile-missing-figures.sh
```

4.9 How to debug

If for some reasons you are unable to understand why a build fails, first check if you compiled your document with `-shell-escape` (not that this must appear **before** the filename). Then, you can look at the log file to get more advices: when a cached document is compiled, we always write the full compilation command before compiling the file in the log file. This way, you can easily check the content of the file and see why it fails to compile. The compilation errors are also displayed directly in the output, and you can read the log file for details.

You might often get an error `! Missing $ inserted.`: this is typically when a placeholder was left unreplaced (e.g. you forgot to define it, import it, or you forgot to wrap a command in `\evalPlaceholder{}`): since LaTeX is asked to typeset `__something__`, it thinks that you are trying to write a subscript, and asks you to start first the math mode.

To get more advanced info during the compilation command, set the style `more logs` (that does `\def\robExtDebugMessage#1{\message{^^J[robExt] #1}}`) in order to print some messages on the replacement process (there is also `\robExtDebugWarning` for more important messages, but it prints the logs by default.

Then, it is often handy to check the content of the generated template. You can either get the name in the log, via `name output`, or print it directly in the file:

/robExt/debug   (style, no value)
/robExt/print command and source   (style, no value)

You can also use `debug` (or its alias `print command and source`) in order to disable the compilation and print instead the compilation command and the source file:

Command: (run in folder `robustExternalize/`)

```
pdflatex -halt-on-error "robExt-C258047503967D680D684293CEAD8CE4.tex"
```

Dependencies:

```
command,pdflatex -halt-on-error "__ROBEXT_SOURCE_FILE__"
50E2E35E8F3DC9D3A97DC41F843CF721,
```

Source (in `robustExternalize/robExt-C258047503967D680D684293CEAD8CE4.tex`):

```
\documentclass[,margin=30cm]{standalone}
 \usepackage {tikz}
% most packages must be loaded before hyperref
% so we typically want to load hyperref here

% some packages must be loaded after hyperref

\begin{document}%
%% We save the height/depth of the content by using a savebox:
\newwrite\writeRobExt%
\immediate\openout\writeRobExt=\jobname-out.tex%
%
\newsavebox\boxRobExt%
\begin{lrbox}{\boxRobExt}%
  \begin {tikzpicture}[baseline=(A.base)] \node [fill=red, rounded corners](A){My node tha
\end{lrbox}%
\usebox{\boxRobExt}%
\immediate\write\writeRobExt{%
  \string\def\string\robExtWidth{\the\wd\boxRobExt}%
  \string\def\string\robExtHeight{\the\ht\boxRobExt}%
  \string\def\string\robExtDepth{\the\dp\boxRobExt}%
}%
%

\end{document}
```

```
\begin{tikzpictureC}<debug>[baseline=(A.base)]
  \node[fill=red, rounded corners](A){My node that respects baseline \ding{164}.};
  \node[fill=red, rounded corners, opacity=.3,overlay] at (A.north east){I am an overlay text};
\end{tikzpictureC}
```

\robExtShowPlaceholder*{⟨*placeholder name*⟩}
/robExt/show placeholder={⟨*placeholder name*⟩}   (style, no default)

   \robExtShowPlaceholder{__ROBEXT_MAIN_CONTENT__} will show in the terminal the content
   of the placeholder. Use the start not to pause.

\robExtShowPlaceholders*
\robExtShowPlaceholdersContents*
/robExt/show placeholders   (style, no value)
/robExt/show placeholders contents   (style, no value)

   Show all imported placeholders and their content. Use the start not to pause.

4.10 Advanced optimizations

We made a number of optimizations to get near instant feedback on documents having hundreds of pictures, basically by saying which placeholder should be replaced, in which order, when to stop replacing placeholders, and possibly by compiling templates (see section 3.12). Most of them are invisible to the end user and pre-configured for presets like `latex` or when compiling a preset, but you might want to also use them on your own.

First note that the order used to import placeholders is important, as it will first replace the first imported placeholders: so if you imported the placeholders in the right order, a single loop should be enough to replace them all, otherwise you might need n loops if you need to replace n placeholders (the basic evaluation procedure just loops over placeholders and replace all occurrences in the string until there is either no remaining placeholders (see below), or if you obtain the same result after two loops).

`\onlyPlaceholdersInCompilationCommand{⟨`*list,of,placeholders*`⟩}`
`\firstPlaceholdersInCompilationCommand{⟨`*list,of,placeholders*`⟩}`
`\onlyPlaceholdersInTemplate{⟨`*list,of,placeholders*`⟩}`
`\firstPlaceholdersInTemplate{⟨`*list,of,placeholders*`⟩}`

> Using these commands (possibly inside `/utils/exec={\yourcommand{foo}}` in a style), you will tell to the system to evaluate the compilation/template placeholders (depending on the name of the command) only using the placeholders in the argument, or (for the commands starting with `first`), by starting with the placeholders in argument before continuing with others placeholders.

`/robExt/all placeholders have underscores`   (style, no value)
`/robExt/not all placeholders have underscores`   (style, no value)

> In order to know when to stop replacing placeholders earlier (i.e. without running until it converges), it is practical to know how placeholders look like in order to stop before. This (enabled by default) says to the library that all placeholders should start with two underscores `__`, allowing us to easily check if a string contains a placeholder. If you want to use placeholders with fancy names and no underscores, disable this feature (but it might be a bit slower).

`/robExt/do not load special placeholders`   (style, no value)
`/robExt/load special placeholders`   (style, no value)

> Similarly, to save some time, we do not import some special placeholders and only replace them manually at the end, the list being: `__ROBEXT_OUTPUT_PREFIX__`, `__ROBEXT_SOURCE_FILE__`, `__ROBEXT_OUTPUT_PDF__`, `__ROBEXT_WAY_BACK__`, `__ROBEXT_CACHE_FOLDER__`. If for some reasons you do want to import them earlier, you can do it this way.

`/robExt/disable placeholders`   (style, no value)
`/robExt/enable placeholders`   (style, no value)

> To save even more time, you can disable the placeholder system completely, and only replace the following placeholders: `__ROBEXT_MAIN_CONTENT__`, `__ROBEXT_LATEX_PREAMBLE__` and `__ROBEXT_MAIN_CONTENT_ORIG__`. This is done notably inside `new compiled preset` in order to speed up the compilation by maybe a factor of 1.5 or 2, but then you cannot use any other placeholders. Note that the default presets like `latex` enable placeholders, so you should be able to call `latex` inside a compiled template to come back to the uncompiled version.

`/robExt/disable import mechanism`   (style, no value)
`/robExt/enable optimizations`   (style, no value)

> If you disable the import mechanism (enabled by default), it will do an `\importAllPlaceholders` while evaluating the template. This is not recommended as it is quite inefficient, it is better to selectively import the group you need.

`/robExt/disable optimizations`   (style, no value)
`/robExt/enable optimizations`   (style, no value)

5 Frequent errors

5.1 Xelatex and tabs

It seems that xelatex (I never had issues with pdflatex and lualatex, but if you get this error on other engines you can try the same fix) has a hard time dealing with tabs[10] if you do not compile with:

```
xelatex -8bit yourfile.tex
```

If you see that tabs in your input file are turned into `^^I` in the source code, it might be worth adding the `-8bit` option.

6 TODO and known bugs:

- See how to deal with label, links and cite inside pictures (without disabling externalization). For label and links, I have a proof of concept, I "just" need to write it down. `https://tex.stackexchange.com/questions/695277/clickable-includegraphics-for-cross-reference-data`

- Deal with remember picture

- Since externalizing is all about speed, it would be nice to do more benchmarks. But overall, it seems fairly optimized, at least when using the compiled presets.

- Add a way to forward tikz styles and/or arbitrary code.

- Create a more efficient/precise way to check if a word is present than string "is in". Indeed, for now for each color/... to export, we check if the string contains this word, which scales like $O(sn)$ where s is the size of the string and n is the number of colors etc to export. Instead, I'd like to run it in $O(s)$ by extracting all words a single time, and then checking if this word exists or not. This is not an issue for `\newcommandAutoforward`, but could be an issue when exporting many colors or `\genericAutoForward`.

7 Acknowledgments

I am deeply indebted to many users on `tex.stackexchange.com` that made the writing of this library possible. I can't list you all, but thank you so much! A big thanks also to the project `https://github.com/sasozivanovic/memoize/` from which I borrowed most of the code to automatically wrap commands. Thanks to kiryph for providing useful feedbacks.

8 Changelog

- Future v2.3 (master on github, not released yet on CTAN):

    - fix bug in `more logs` and `less logs`
    - make `if matches word` more efficient, independent on the number of runs, allowing registration, namespaces, etc.
    - Added `\definecolorAutoForward` and `\colorletAutoForward`
    - Add support for gnuplot
    - Added `\runHereAndInPreambleOfCachedFiles`

- v2.2

    - v2.1 introduced two important bugs where hashes could not be used in a `\cacheMe` environment, and `auto forward` was not working if the string was containing an environment. This is fixed now.

---

[10]`https://tex.stackexchange.com/a/14776/116348`

- Fix issue with `compile in parallel` on MacOs (xargs there is not GNU xargs and is missing `-a` option)
- Added `*ExpandableDocumentCommandAutoForward`.
- Added `\genericAutoForward` to forward arbitrary code.
- Added `if windows` and `if unix` tests.
- Added details on the future difference between `if matches` and `if matches word`.

- v2.1:

  - Fix a bug when compiling the main document with lualatex or xelatex
  - Fix a bug when compiling a template that was not exporting dependencies
  - Add `recompile` to recompile a file even if no dependencies changed.
  - Add a hook `robust-externalize/just-before-writing-files`.
  - Fixed a bug in `\cacheTikz` that was not working with a custom argument for tikzpicture
  - Added options `forward*` to forward macros and counters
  - Added `auto forward` and co
  - Added `compile in parallel` and co

- v2.0:

  - **not backward compatible**: the preset `tikzpicture` should be used instead of `tikz`: `tikz` does not wrap anymore its content into `\begin{tikzpicture}` in order to be usable inside the command version `\tikz` (but `tikzpicture` does load `tikz` first). Moreover, all arguments to `tikzpictureC` or `tikzpicture` (if you loaded `cacheTikz`) should be specified in the first argument, using `<your options>`. This is used to provide a more uniform interface with the new `\cacheEnvironment` command. Hopefully this should **not cause too much trouble** since the first version of the library has been published on CTAN only a few days ago.
  - not really compatible: Renamed some latex placeholders to prefix all of them with `__ROBEXT_LATEX_`. Should not be a problem if you used the style made to change them.
  - Added an alias to `\robExtExternalizeAllTikzpictures` called `\cacheTikz`. Now, this also configures `\tikz`.
  - You can now use ampersands etc freely thanks to lrbox.
  - You can now automatically cache environments using `\cacheEnvironment` and commands using `\cacheCommand`.
  - We provide `new preset` and `add to preset` to avoid doubling the number of hashes.
  - We provide new functions like `\setPlaceholderRecReplaceFromList` to only expand a subset of placeholders.
  - An error message is given if you forget to set a template.
  - Add `enable fallback to manual mode` to fallback to manual mode only if shell escape is missing.
  - Allow restricted mode
  - `debug` mode available
  - Rename `set subfolder and way back` to `set cache folder and way back` (in a backward compatible way)
  - Waaaaaaay more efficient. Also added a method to compile presets.
  - I added the import system for efficiency reasons (not present in v1.0 at all). Note that I also added some other functions that I forgot to notify as new in the doc...

Index

This index only contains automatically generated entries. A good index should also contain carefully selected keywords. This index is not a good index.