Generating the documentation

To generate the documentation, you first have to build it. Several packages are necessary to build the doc, you can install them with the following command, at the root of the code repository:

```
pip install -e ".[docs]"
```

Then you need to install our special tool that builds the documentation:

```
pip install git+https://github.com/huggingface/doc-builder
```

NOTE

You only need to generate the documentation to inspect it locally (if you're planning changes and want to check how they look like before committing for instance). You don't have to commit the built documentation.

Building the documentation

Once you have setup the `doc-builder` and additional packages, you can generate the documentation by typing the following command:

```
doc-builder build transformers docs/source/ --build_dir ~/tmp/test-build
```

You can adapt the `--build_dir` to set any temporary folder that you prefer. This command will create it and generate the MDX files that will be rendered as the documentation on the main website. You can inspect them in your favorite Markdown editor.

NOTE

It's not possible to see locally how the final documentation will look like for now. Once you have opened a PR, you will see a bot add a comment to a link where the documentation with your changes lives.

Adding a new element to the navigation bar

Accepted files are Markdown (.md or .mdx).

Create a file with its extension and put it in the source directory. You can then link it to the toc-tree by putting the filename without the extension in the `_toctree.yml` file.

Renaming section headers and moving sections

It helps to keep the old links working when renaming section header and/or moving sections from one document to another. This is because the old links are likely to be used in Issues, Forums and Social media and it'd be make for a much more superior user experience if users reading those months later could still easily navigate to the originally intended information.

Therefore we simply keep a little map of moved sections at the end of the document where the original section was. The key is to preserve the original anchor.

So if you renamed a section from: "Section A" to "Section B", then you can add at the end of the file:

```
Sections that were moved:

[ <a href="#section-b">Section A</a><a id="section-a"></a> ]
```

and of course if you moved it to another file, then:

```
Sections that were moved:

[ <a href="../new-file#section-b">Section A</a><a id="section-a"></a> ]
```

Use the relative style to link to the new file so that the versioned docs continue to work.

For an example of a rich moved sections set please see the very end of the Trainer doc.

Writing Documentation - Specification

The `huggingface/transformers` documentation follows the Google documentation style for docstrings, although we can write them directly in Markdown.

Adding a new tutorial

Adding a new tutorial or section is done in two steps:

- Add a new file under `./source`. This file can either be ReStructuredText (.rst) or Markdown (.md).
- Link that file in `./source/_toctree.yml` on the correct toc-tree.

Make sure to put your new file under the proper section. It's unlikely to go in the first section (*Get Started*), so depending on the intended targets (beginners, more advanced users or researchers) it should go in section two, three or four.

Adding a new model

When adding a new model:

- Create a file `xxx.mdx` or under `./source/model_doc` (don't hesitate to copy an existing file as template).
- Link that file in `./source/_toctree.yml`.
- Write a short overview of the model:
  - Overview with paper & authors
  - Paper abstract
  - Tips and tricks and how to use it best
- Add the classes that should be linked in the model. This generally includes the configuration, the tokenizer, and every model of that class (the base model, alongside models with additional heads), both in PyTorch and TensorFlow. The order is generally:
  - Configuration,
  - Tokenizer
  - PyTorch base model
  - PyTorch head models
  - TensorFlow base model
  - TensorFlow head models
  - Flax base model

- Flax head models

These classes should be added using our Markdown syntax. Usually as follows:

```
## XXXConfig

[[autodoc]] XXXConfig
```

This will include every public method of the configuration that is documented. If for some reason you wish for a method not to be displayed in the documentation, you can do so by specifying which methods should be in the docs:

```
## XXXTokenizer

[[autodoc]] XXXTokenizer
    - build_inputs_with_special_tokens
    - get_special_tokens_mask
    - create_token_type_ids_from_sequences
    - save_vocabulary
```

If you just want to add a method that is not documented (for instance magic method like `__call__` are not documented byt default) you can put the list of methods to add in a list that contains `all` :

```
## XXXTokenizer

[[autodoc]] XXXTokenizer
    - all
    - __call__
```

Writing source documentation

Values that should be put in `code` should either be surrounded by backticks: \`like so\`. Note that argument names and objects like True, None or any strings should usually be put in `code` .

When mentioning a class, function or method, it is recommended to use our syntax for internal links so that our tool adds a link to its documentation with this syntax: [\`XXXClass\`] or [\`function\`]. This requires the class or function to be in the main package.

If you want to create a link to some internal class or function, you need to provide its path. For instance: [\`utils.ModelOutput\`]. This will be converted into a link with `utils.ModelOutput` in the description. To get rid of the path and only keep the name of the object you are linking to in the description, add a ~~: [~~\`utils.ModelOutput\`] will generate a link with `ModelOutput` in the description.

The same works for methods so you can either use [\`XXXClass.method\`] or [~\`XXXClass.method\`].

Defining arguments in a method

Arguments should be defined with the `Args:` (or `Arguments:` or `Parameters:`) prefix, followed by a line return and an indentation. The argument should be followed by its type, with its shape if it is a tensor, a colon and its description:

```
    Args:
        n_layers (`int`): The number of layers of the model.
```

If the description is too long to fit in one line, another indentation is necessary before writing the description after th argument.

Here's an example showcasing everything so far:

```
    Args:
        input_ids (`torch.LongTensor` of shape `(batch_size, sequence_length)`):
            Indices of input sequence tokens in the vocabulary.

            Indices can be obtained using [`AlbertTokenizer`]. See
[`~PreTrainedTokenizer.encode`] and
            [`~PreTrainedTokenizer.__call__`] for details.

            [What are input IDs?](../glossary#input-ids)
```

For optional arguments or arguments with defaults we follow the following syntax: imagine we have a function with the following signature:

```
def my_function(x: str = None, a: float = 1):
```

then its documentation should look like this:

```
    Args:
        x (`str`, *optional*):
            This argument controls ...
        a (`float`, *optional*, defaults to 1):
            This argument is used to ...
```

Note that we always omit the "defaults to `None`" when None is the default for any argument. Also note that even if the first line describing your argument type and its default gets long, you can't break it on several lines. You can however write as many lines as you want in the indented description (see the example above with `input_ids`).

Writing a multi-line code block

Multi-line code blocks can be useful for displaying examples. They are done between two lines of three backticks as usual in Markdown:

~~~
```
# first line of code
# second line
# etc
```
~~~

We follow the doctest syntax for the examples to automatically test the results stay consistent with the library.

Writing a return block

The return block should be introduced with the `Returns:` prefix, followed by a line return and an indentation. The first line should be the type of the return, followed by a line return. No need to indent further for the elements building the return.

Here's an example for a single value return:

```
    Returns:
        `List[int]`: A list of integers in the range [0, 1] --- 1 for a special token,
0 for a sequence token.
```

Here's an example for tuple return, comprising several objects:

```
    Returns:
        `tuple(torch.FloatTensor)` comprising various elements depending on the
configuration ([`BertConfig`]) and inputs:
        - ** loss** (*optional*, returned when `masked_lm_labels` is provided)
`torch.FloatTensor` of shape `(1,)` --
          Total loss as the sum of the masked language modeling loss and the next
sequence prediction (classification) loss.
        - **prediction_scores** (`torch.FloatTensor` of shape `(batch_size,
sequence_length, config.vocab_size)`) --
          Prediction scores of the language modeling head (scores for each vocabulary
token before SoftMax).
```

Adding an image

Due to the rapidly growing repository, it is important to make sure that no files that would significantly weigh down the repository are added. This includes images, videos and other non-text files. We prefer to leverage a hf.co hosted `dataset` like the ones hosted on `hf-internal-testing` in which to place these files and reference them by URL. We recommend putting them in the following dataset: huggingface/documentation-images. If an external contribution, feel free to add the images to your PR and ask a Hugging Face member to migrate your images to this dataset.

Styling the docstring

We have an automatic script running with the `make style` comment that will make sure that:

- the docstrings fully take advantage of the line width
- all code examples are formatted using black, like the code of the Transformers library

This script may have some weird failures if you made a syntax mistake or if you uncover a bug. Therefore, it's recommended to commit your changes before running `make style`, so you can revert the changes done by that script easily.

Testing documentation examples

Good documentation oftens comes with an example of how a specific function or class should be used. Each model class should contain at least one example showcasing how to use this model class in inference. *E.g.* the class Wav2Vec2ForCTC includes an example of how to transcribe speech to text in the docstring of its forward function.

Writing documenation examples

The syntax for Example docstrings can look as follows:

~~~
    Example:

    ```python
~~~

```
    >>> from transformers import Wav2Vec2Processor, Wav2Vec2ForCTC
    >>> from datasets import load_dataset
    >>> import torch

    >>> dataset = load_dataset("hf-internal-testing/librispeech_asr_demo", "clean",
split="validation")
    >>> dataset = dataset.sort("id")
    >>> sampling_rate = dataset.features["audio"].sampling_rate

    >>> processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-base-960h")
    >>> model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-base-960h")

    >>> # audio file is decoded on the fly
    >>> inputs = processor(dataset[0]["audio"]["array"], sampling_rate=sampling_rate,
return_tensors="pt")
    >>> with torch.no_grad():
    ...     logits = model(**inputs).logits
    >>> predicted_ids = torch.argmax(logits, dim=-1)

    >>> # transcribe speech
    >>> transcription = processor.batch_decode(predicted_ids)
    >>> transcription[0]
    'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE CLASSES AND WE ARE GLAD TO WELCOME
HIS GOSPEL'
    ```
```

The docstring should give a minimal, clear example of how the respective model is to be used in inference and also include the expected (ideally sensible) output. Often, readers will try out the example before even going through the function or class definitions. Therefore it is of utmost importance that the example works as expected.

Docstring testing

To do so each example should be included in the doctests. We use pytests' doctest integration to verify that all of our examples run correctly. For Transformers, the doctests are run on a daily basis via GitHub Actions as can be seen here.

To include your example in the daily doctests, you need add the filename that contains the example docstring to the documentation_tests.txt.

For Python files

You will first need to run the following command (from the root of the repository) to prepare the doc file (doc-testing needs to add additional lines that we don't include in the doc source files):

```
python utils/prepare_for_doc_test.py src docs
```

If you work on a specific python module, say `modeling_wav2vec2.py`, you can run the command as follows (to avoid the unnecessary temporary changes in irrelevant files):

```
python utils/prepare_for_doc_test.py src/transformers/utils/doc.py
src/transformers/models/wav2vec2/modeling_wav2vec2.py
```
```

( `utils/doc.py` should always be included)

Then you can run all the tests in the docstrings of a given file with the following command, here is how we test the modeling file of Wav2Vec2 for instance:

```
pytest --doctest-modules src/transformers/models/wav2vec2/modeling_wav2vec2.py -sv -
-doctest-continue-on-failure
```

If you want to isolate a specific docstring, just add `::` after the file name then type the whole path of the function/class/method whose docstring you want to test. For instance, here is how to just test the forward method of `Wav2Vec2ForCTC`:

```
pytest --doctest-modules
src/transformers/models/wav2vec2/modeling_wav2vec2.py::transformers.models.wav2vec2.mo
 -sv --doctest-continue-on-failure
```

Once you're done, you can run the following command (still from the root of the repository) to undo the changes made by the first command before committing:

```
python utils/prepare_for_doc_test.py src docs --remove_new_line
```

For Markdown files

You will first need to run the following command (from the root of the repository) to prepare the doc file (doc-testing needs to add additional lines that we don't include in the doc source files):

```
python utils/prepare_for_doc_test.py src docs
```

Then you can test locally a given file with this command (here testing the quicktour):

```
pytest --doctest-modules docs/source/quicktour.mdx -sv --doctest-continue-on-failure
--doctest-glob="*.mdx"
```

Once you're done, you can run the following command (still from the root of the repository) to undo the changes made by the first command before committing:

```
python utils/prepare_for_doc_test.py src docs --remove_new_line
```

Writing doctests

Here are a few tips to help you debug the doctests and make them pass:

- The outputs of the code need to match the expected output **exactly**, so make sure you have the same outputs. In particular doctest will see a difference between single quotes and double quotes, or a missing parenthesis. The only exceptions to that rule are:
  - whitespace: one give whitespace (space, tabulation, new line) is equivalent to any number of whitespace, so you can add new lines where there are spaces to make your output more readable.
  - numerical values: you should never put more than 4 or 5 digits to expected results as different setups or library versions might get you slightly different results. `doctest` is configure to ignore

    any difference lower than the precision to which you wrote (so 1e-4 if you write 4 digits).

- Don't leave a block of code that is very long to execute. If you can't make it fast, you can either not use the doctest syntax on it (so that it's ignored), or if you want to use the doctest syntax to show the results, you can add a comment `# doctest: +SKIP` at the end of the lines of code too long to execute
- Each line of code that produces a result needs to have that result written below. You can ignore an output if you don't want to show it in your code example by adding a comment `# doctest: +IGNORE_RESULT` at the end of the line of code produing it.