faker-file Documentation

Release 0.17.11

Artur Barseghyan <artur.barseghyan@gmail.com>

Dec 21, 2023

CONTENTS

Create files with fake data. In many formats. With no efforts.

CHAPTER

ONE

PREREQUISITES

All of core dependencies of this package are *MIT* licensed. Most of optional dependencies of this package are *MIT* licensed, while a few are *BSD*-, *Apache 2*-, *GPL* or *HPND* licensed.

All licenses are mentioned below between the brackets.

- Core package requires Python 3.7, 3.8, 3.9, 3.10 or 3.11.
- Faker (*MIT*) is the only required dependency.
- Django (*BSD*) integration with factory_boy (*MIT*) has been tested with `Django` starting from version 2.2 to 4.2 (although only maintained versions of Django are currently being tested against).
- `BMP`, `GIF` and `TIFF` file support requires either just Pillow (*HPND*), or a combination of WeasyPrint (*BSD*), pdf2image (*MIT*), Pillow (*HPND*) and poppler (*GPLv2*).
- `DOCX` file support requires python-docx (*MIT*).
- `EPUB` file support requires xml2epub (*MIT*) and Jinja2 (*BSD*).
- `ICO`, `JPEG`, `PNG`, `SVG` and `WEBP` files support requires either just Pillow (*HPND*), or a combination of imgkit (*MIT*) and wkhtmltopdf (*LGPLv3*).
- `MP3` file support requires gTTS (*MIT*) or edge-tts (*GPLv3*).
- `PDF` file support requires either Pillow (*HPND*), or a combination of pdfkit (*MIT*) and wkhtmltopdf (*LGPLv3*), or reportlab (*BSD*).
- `PPTX` file support requires python-pptx (*MIT*).
- `ODP` and `ODT` file support requires odfpy (*Apache 2*).
- `ODS` file support requires tablib (*MIT*) and odfpy (*Apache 2*).
- `XLSX` file support requires tablib (*MIT*) and openpyxl (*MIT*).
- `PathyFileSystemStorage` storage support requires pathy (*Apache 2*).
- `AWSS3Storage` storage support requires pathy (*Apache 2*) and boto3 (*Apache 2*).
- `AzureCloudStorage` storage support requires pathy (*Apache 2*) and azure-storage-blob (*MIT*).
- `GoogleCloudStorage` storage support requires pathy (*Apache 2*) and google-cloud-storage (*Apache 2*).
- `SFTPStorage` storage support requires paramiko (*LGLPv2.1*).
- `AugmentFileFromDirProvider` provider requires either a combination of textaugment (*MIT*) and nltk (*Apache 2*) or a combination of nlpaug (*MIT*), PyTorch (*BSD*), transformers (*Apache 2*), numpy (*BSD*), pandas (*BSD*), tika (*Apache 2*) and Apache Tika (*Apache 2*).

CHAPTER

TWO

DOCUMENTATION

- Documentation is available on Read the Docs.
- For bootstrapping check the Quick start.
- For various ready to use code examples see the Recipes.
- For tips on `PDF` creation see Creating PDF.
- For tips on `DOCX` creation see Creating DOCX.
- For tips on `ODT` creation see Creating ODT.
- For tips on images creation see Creating images.
- For CLI options see the CLI.
- Read the Methodology.
- For guidelines on contributing check the Contributor guidelines.

CHAPTER

THREE

ONLINE DEMOS AND RELATED PROJECTS

Check the demo(s) and related projects below:

- REST API demo (based on faker-file-api REST API)
- UI frontend demo (based on faker-file-ui UI frontend)
- WASM frontend demo (based on faker-file-wasm WASM frontend)
- faker-file-qt GUI application (based on PyQT5).

CHAPTER

FOUR

INSTALLATION

4.1 Latest stable version from PyPI

WIth all dependencies

```
pip install faker-file[all]
```

Only core

```
pip install faker-file
```

With most common dependencies

Everything, except ML libraries which are required for data augmentation only

```
pip install faker-file[common]
```

With DOCX support

```
pip install faker-file[docx]
```

With EPUB support

```
pip install faker-file[epub]
```

With images support

```
pip install faker-file[images]
```

With PDF support

```
pip install faker-file[pdf]
```

With MP3 support

```
pip install faker-file[mp3]
```

With XLSX support

```
pip install faker-file[xlsx]
```

With ODS support

```
pip install faker-file[ods]
```

With ODT support

```
pip install faker-file[odt]
```

With data augmentation support

```
pip install faker-file[data-augmentation]
```

With GoogleCloudStorage support

```
pip install faker-file[gcs]
```

With AzureCloudStorage support

```
pip install faker-file[azure]
```

With AWSS3Storage support

```
pip install faker-file[s3]
```

4.2 Or development version from GitHub

```
pip install https://github.com/barseghyanartur/faker-file/archive/main.tar.gz
```

CHAPTER

FIVE

FEATURES

5.1 Supported file types

- BIN
- BMP
- CSV
- DOCX
- EML
- EPUB
- ICO
- GIF
- JPEG
- JSON
- MP3
- ODS
- ODT
- ODP
- PDF
- PNG
- RTF
- PPTX
- SVG
- TAR
- TIFF
- TXT
- WEBP
- XLSX
- XML
- ZIP

For all image formats (`BMP`, `ICO`, `GIF`, `JPEG`, `PNG`, `SVG`, `TIFF` and `WEBP`) and `PDF`, there are both graphic-only and mixed-content file providers (that also have text-to-image capabilities).

5.2 Additional providers

- `AugmentFileFromDirProvider`: Make an augmented copy of randomly picked file from given directory. The following types are supported : `DOCX`, `EML`, `EPUB`, `ODT`, `PDF`, `RTF` and `TXT`.
- `AugmentRandomImageFromDirProvider`: Augment a random image file from given directory. The following types are supported : `BMP`, `GIF`, `JPEG`, `PNG`, `TIFF` and `WEBP`.
- `AugmentImageFromPathProvider`: Augment an image file from given path. Supported file types are the same as for `AugmentRandomImageFromDirProvider` provider.
- `GenericFileProvider`: Create files in any format from raw bytes or a predefined template.
- `RandomFileFromDirProvider`: Pick a random file from given directory.
- `FileFromPathProvider`: File from given path.

5.3 Supported file storages

- Native file system storage
- AWS S3 storage
- Azure Cloud Storage
- Google Cloud Storage
- SFTP storage

CHAPTER

SIX

USAGE EXAMPLES

6.1 With Faker

Recommended way

```
from faker import Faker
# Import the file provider we want to use
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()  # Initialise Faker instance
FAKER.add_provider(TxtFileProvider)  # Register the TXT file provider

file = FAKER.txt_file()  # Generate a TXT file

# Meta-data is stored inside a ``data`` attribute (``dict``).
# The following line would produce something like /tmp/tmp/tmphzzb8mot.txt
print(file.data["filename"])
# The following line would produce a text generated by Faker, used as
# the content of the generated file.
print(file.data["content"])
```

Note: Note, that in this case `file` value is a `StringValue` instance, which inherits from `str` but contains meta-data such as absolute path to the generated file, and text used to generate the file, stored in `filename` and `content` keys of the `data` attribute respectively. See *Meta-data* for more information.

If you just need `bytes` back (instead of creating the file), provide the `raw=True` argument (works with all provider classes and inner functions):

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

raw = FAKER.txt_file(raw=True)
```

Note: Note, that in this case `file` value is a `BytesValue` instance, which inherits from `bytes` but contains meta-data such as absolute path to the generated file, and text used to generate the file, stored in `filename` and `content` keys of

the `data` attribute respectively. See *Meta-data* for more information.

But this works too

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file()
```

If you just need `bytes` back:

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()

raw = TxtFileProvider(FAKER).txt_file(raw=True)
```

6.2 With `factory_boy`

6.2.1 upload/models.py

```
from django.db import models

class Upload(models.Model):

    # ...
    file = models.FileField()
```

6.2.2 upload/factories.py

Note, that when using `faker-file` with `Django` and native file system storages, you need to pass your `MEDIA_ROOT` setting as `root_path` value to the chosen file storage as show below.

```
import factory
from django.conf import settings
from factory import Faker
from factory.django import DjangoModelFactory
from faker_file.providers.docx_file import DocxFileProvider
from faker_file.storages.filesystem import FileSystemStorage

from upload.models import Upload

FS_STORAGE = FileSystemStorage(
    root_path=settings.MEDIA_ROOT,
    rel_path="tmp"
)
```

(continues on next page)

(continued from previous page)

```
factory.Faker.add_provider(DocxFileProvider)

class UploadFactory(DjangoModelFactory):

    # ...
    file = Faker("docx_file", storage=FS_STORAGE)

    class Meta:
        model = Upload
```

CHAPTER

SEVEN

META-DATA

The return value of any file provider file generator function is either `StringValue` or `BytesValue`, which inherit from `str` and `bytes` respectively.

Both `StringValue` and `BytesValue` instances have a meta data attribute named `data` (type `dict`). Various file providers use `data` to store meta-data, such as `filename` (absolute path to the generated file; valid for all file providers), or `content` (text used when generating the file; valid for most file providers, except `FileFromPathProvider`, `RandomFileFromDirProvider`, `TarFileProvider` and `ZipFileProvider`).

All file providers store an absolute path to the generated file in `filename` key of the `data` attribute and instance of the storage used in `storage` key. See the table below.

Key name	File provider
filename	all
storage	all
content	all except FileFromPathProvider, RandomFileFromDirProvider, TarFileProvider, ZipFileProvider and all graphic file providers such as GraphicBmpFileProvider, GraphicGifFileProvider, GraphicIcoFileProvider, GraphicJpegFileProvider, GraphicPdfFileProvider, GraphicPngFileProvider, GraphicTiffFileProvider and GraphicWebpFileProvider
inner	only EmlFileProvider, TarFileProvider and ZipFileProvider

CHAPTER

EIGHT

FILE STORAGES

All file operations are delegated to a separate abstraction layer of storages.

The following storages are implemented:

- `FileSystemStorage`: Does not have additional requirements.
- `PathyFileSystemStorage`: Requires pathy.
- `AzureCloudStorage`: Requires pathy and *Azure* related dependencies.
- `GoogleCloudStorage`: Requires pathy and *Google Cloud* related dependencies.
- `AWSS3Storage`: Requires pathy and *AWS S3* related dependencies.
- `SFTPStorage`: Requires paramiko and related dependencies.

8.1 Usage example with storages

8.1.1 *FileSystemStorage* example

Native file system storage. Does not have dependencies.

- `root_path`: Path to the root directory. Given the example of Django, this would be the path to the `MEDIA_ROOT` directory. It's important to know, that `root_path` will not be embedded into the string representation of the file. Only `rel_path` will.
- `rel_path`: Relative path from the root directory. Given the example of Django, this would be the rest of the path to the file.

```
import tempfile
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FS_STORAGE = FileSystemStorage(
    root_path=tempfile.gettempdir(),  # Use settings.MEDIA_ROOT for Django
    rel_path="tmp",
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=FS_STORAGE)
```

(continues on next page)

(continued from previous page)

```
FS_STORAGE.exists(file)
```

8.1.2 *PathyFileSystemStorage* example

Native file system storage. Requires `pathy`.

```
import tempfile
from pathy import use_fs
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.cloud import PathyFileSystemStorage

use_fs(tempfile.gettempdir())
PATHY_FS_STORAGE = PathyFileSystemStorage(
    bucket_name="bucket_name",
    root_path="tmp",
    rel_path="sub-tmp",
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=PATHY_FS_STORAGE)

PATHY_FS_STORAGE.exists(file)
```

8.1.3 *AWSS3Storage* example

AWS S3 storage. Requires `pathy` and `boto3`.

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.aws_s3 import AWSS3Storage

S3_STORAGE = AWSS3Storage(
    bucket_name="bucket_name",
    root_path="tmp",  # Optional
    rel_path="sub-tmp",  # Optional
    # Credentials are optional too. If your AWS credentials are properly
    # set in the ~/.aws/credentials, you don't need to send them
    # explicitly.
    credentials={
        "key_id": "YOUR KEY ID",
        "key_secret": "YOUR KEY SECRET"
    },
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=S3_STORAGE)
```

(continues on next page)

(continued from previous page)

```
S3_STORAGE.exists(file)
```

CHAPTER

NINE

TESTING

Simply type:

```
pytest -vrx
```

Or use tox:

```
tox
```

Or use tox to check specific env:

```
tox -e py310-django41
```

CHAPTER

TEN

WRITING DOCUMENTATION

Keep the following hierarchy.

```
=====
title
=====

header
======

sub-header
----------

sub-sub-header
~~~~~~~~~~~~~~

sub-sub-sub-header
^^^^^^^^^^^^^^^^^^

sub-sub-sub-sub-header
++++++++++++++++++++++

sub-sub-sub-sub-sub-header
**************************
```

CHAPTER

ELEVEN

LICENSE

MIT

CHAPTER

TWELVE

SUPPORT

For security issues contact me at the e-mail given in the *Author* section.

For overall issues, go to GitHub.

CHAPTER

THIRTEEN

AUTHOR

Artur Barseghyan <artur.barseghyan@gmail.com>

CHAPTER

FOURTEEN

CITATION

Please, use the following entry when citing faker-file in your research:

```
@software{faker-file,
  author = {Artur Barseghyan},
  title = {faker-file: Create files with fake data. In many formats. With no efforts.},
  year = {2023},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {https://github.com/barseghyanartur/faker-file},
}
```

CHAPTER

FIFTEEN

PROJECT DOCUMENTATION

Contents:

Table of Contents

15.1 faker-file

Create files with fake data. In many formats. With no efforts.

15.1.1 Prerequisites

All of core dependencies of this package are *MIT* licensed. Most of optional dependencies of this package are *MIT* licensed, while a few are *BSD-*, *Apache 2-*, *GPL* or *HPND* licensed.

All licenses are mentioned below between the brackets.

- Core package requires Python 3.7, 3.8, 3.9, 3.10 or 3.11.
- Faker (*MIT*) is the only required dependency.
- Django (*BSD*) integration with factory_boy (*MIT*) has been tested with `Django` starting from version 2.2 to 4.2 (although only maintained versions of Django are currently being tested against).
- `BMP`, `GIF` and `TIFF` file support requires either just Pillow (*HPND*), or a combination of WeasyPrint (*BSD*), pdf2image (*MIT*), Pillow (*HPND*) and poppler (*GPLv2*).
- `DOCX` file support requires python-docx (*MIT*).
- `EPUB` file support requires xml2epub (*MIT*) and Jinja2 (*BSD*).
- `ICO`, `JPEG`, `PNG`, `SVG` and `WEBP` files support requires either just Pillow (*HPND*), or a combination of imgkit (*MIT*) and wkhtmltopdf (*LGPLv3*).
- `MP3` file support requires gTTS (*MIT*) or edge-tts (*GPLv3*).
- `PDF` file support requires either Pillow (*HPND*), or a combination of pdfkit (*MIT*) and wkhtmltopdf (*LGPLv3*), or reportlab (*BSD*).
- `PPTX` file support requires python-pptx (*MIT*).
- `ODP` and `ODT` file support requires odfpy (*Apache 2*).
- `ODS` file support requires tablib (*MIT*) and odfpy (*Apache 2*).
- `XLSX` file support requires tablib (*MIT*) and openpyxl (*MIT*).

- `PathyFileSystemStorage` storage support requires pathy (*Apache 2*).
- `AWSS3Storage` storage support requires pathy (*Apache 2*) and boto3 (*Apache 2*).
- `AzureCloudStorage` storage support requires pathy (*Apache 2*) and azure-storage-blob (*MIT*).
- `GoogleCloudStorage` storage support requires pathy (*Apache 2*) and google-cloud-storage (*Apache 2*).
- `SFTPStorage` storage support requires paramiko (*LGLPv2.1*).
- `AugmentFileFromDirProvider` provider requires either a combination of textaugment (*MIT*) and nltk (*Apache 2*) or a combination of nlpaug (*MIT*), PyTorch (*BSD*), transformers (*Apache 2*), numpy (*BSD*), pandas (*BSD*), tika (*Apache 2*) and Apache Tika (*Apache 2*).

15.1.2 Documentation

- Documentation is available on Read the Docs.
- For bootstrapping check the Quick start.
- For various ready to use code examples see the Recipes.
- For tips on `PDF` creation see Creating PDF.
- For tips on `DOCX` creation see Creating DOCX.
- For tips on `ODT` creation see Creating ODT.
- For tips on images creation see Creating images.
- For CLI options see the CLI.
- Read the Methodology.
- For guidelines on contributing check the Contributor guidelines.

15.1.3 Online demos and related projects

Check the demo(s) and related projects below:

- REST API demo (based on faker-file-api REST API)
- UI frontend demo (based on faker-file-ui UI frontend)
- WASM frontend demo (based on faker-file-wasm WASM frontend)
- faker-file-qt GUI application (based on PyQT5).

15.1.4 Installation

15.1.4.1 Latest stable version from PyPI

WIth all dependencies

```
pip install faker-file[all]
```

Only core

```
pip install faker-file
```

With most common dependencies

Everything, except ML libraries which are required for data augmentation only

```
pip install faker-file[common]
```

With DOCX support

```
pip install faker-file[docx]
```

With EPUB support

```
pip install faker-file[epub]
```

With images support

```
pip install faker-file[images]
```

With PDF support

```
pip install faker-file[pdf]
```

With MP3 support

```
pip install faker-file[mp3]
```

With XLSX support

```
pip install faker-file[xlsx]
```

With ODS support

```
pip install faker-file[ods]
```

With ODT support

```
pip install faker-file[odt]
```

With data augmentation support

```
pip install faker-file[data-augmentation]
```

With GoogleCloudStorage support

```
pip install faker-file[gcs]
```

With AzureCloudStorage support

```
pip install faker-file[azure]
```

With AWSS3Storage support

```
pip install faker-file[s3]
```

15.1.4.2 Or development version from GitHub

```
pip install https://github.com/barseghyanartur/faker-file/archive/main.tar.gz
```

15.1.5 Features

15.1.5.1 Supported file types

- BIN
- BMP
- CSV
- DOCX
- EML
- EPUB
- ICO
- GIF
- JPEG
- JSON
- MP3
- ODS
- ODT
- ODP
- PDF
- PNG
- RTF
- PPTX
- SVG
- TAR
- TIFF
- TXT
- WEBP
- XLSX
- XML
- ZIP

For all image formats (BMP, ICO, GIF, JPEG, PNG, SVG, TIFF and WEBP) and PDF, there are both graphic-only and mixed-content file providers (that also have text-to-image capabilities).

15.1.5.2 Additional providers

- `AugmentFileFromDirProvider`: Make an augmented copy of randomly picked file from given directory. The following types are supported : `DOCX`, `EML`, `EPUB`, `ODT`, `PDF`, `RTF` and `TXT`.
- `AugmentRandomImageFromDirProvider`: Augment a random image file from given directory. The following types are supported : `BMP`, `GIF`, `JPEG`, `PNG`, `TIFF` and `WEBP`.
- `AugmentImageFromPathProvider`: Augment an image file from given path. Supported file types are the same as for `AugmentRandomImageFromDirProvider` provider.
- `GenericFileProvider`: Create files in any format from raw bytes or a predefined template.
- `RandomFileFromDirProvider`: Pick a random file from given directory.
- `FileFromPathProvider`: File from given path.

15.1.5.3 Supported file storages

- Native file system storage
- AWS S3 storage
- Azure Cloud Storage
- Google Cloud Storage
- SFTP storage

15.1.6 Usage examples

15.1.6.1 With `Faker`

Recommended way

```
from faker import Faker
# Import the file provider we want to use
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()  # Initialise Faker instance
FAKER.add_provider(TxtFileProvider)  # Register the TXT file provider

file = FAKER.txt_file()  # Generate a TXT file

# Meta-data is stored inside a ``data`` attribute (``dict``).
# The following line would produce something like /tmp/tmp/tmphzzb8mot.txt
print(file.data["filename"])
# The following line would produce a text generated by Faker, used as
# the content of the generated file.
print(file.data["content"])
```

Note: Note, that in this case `file` value is a `StringValue` instance, which inherits from `str` but contains meta-data such as absolute path to the generated file, and text used to generate the file, stored in `filename` and `content` keys of the `data` attribute respectively. See *Meta-data* for more information.

If you just need `bytes` back (instead of creating the file), provide the `raw=True` argument (works with all provider classes and inner functions):

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

raw = FAKER.txt_file(raw=True)
```

Note: Note, that in this case `file` value is a `BytesValue` instance, which inherits from `bytes` but contains meta-data such as absolute path to the generated file, and text used to generate the file, stored in `filename` and `content` keys of the `data` attribute respectively. See *Meta-data* for more information.

But this works too

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file()
```

If you just need `bytes` back:

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()

raw = TxtFileProvider(FAKER).txt_file(raw=True)
```

15.1.6.2 With `factory_boy`

15.1.6.2.1 upload/models.py

```
from django.db import models

class Upload(models.Model):

    # ...
    file = models.FileField()
```

15.1.6.2.2 upload/factories.py

Note, that when using `faker-file` with `Django` and native file system storages, you need to pass your `MEDIA_ROOT` setting as `root_path` value to the chosen file storage as show below.

```
import factory
from django.conf import settings
from factory import Faker
from factory.django import DjangoModelFactory
from faker_file.providers.docx_file import DocxFileProvider
from faker_file.storages.filesystem import FileSystemStorage

from upload.models import Upload

FS_STORAGE = FileSystemStorage(
    root_path=settings.MEDIA_ROOT,
    rel_path="tmp"
)
factory.Faker.add_provider(DocxFileProvider)

class UploadFactory(DjangoModelFactory):

    # ...
    file = Faker("docx_file", storage=FS_STORAGE)

    class Meta:
        model = Upload
```

15.1.7 Meta-data

The return value of any file provider file generator function is either `StringValue` or `BytesValue`, which inherit from `str` and `bytes` respectively.

Both `StringValue` and `BytesValue` instances have a meta data attribute named `data` (type `dict`). Various file providers use `data` to store meta-data, such as `filename` (absolute path to the generated file; valid for all file providers), or `content` (text used when generating the file; valid for most file providers, except `FileFromPathProvider`, `RandomFileFromDirProvider`, `TarFileProvider` and `ZipFileProvider`).

All file providers store an absolute path to the generated file in `filename` key of the `data` attribute and instance of the storage used in `storage` key. See the table below.

Key name	File provider
filename	all
storage	all
content	all except FileFromPathProvider, RandomFileFromDirProvider, TarFileProvider, ZipFileProvider and all graphic file providers such as GraphicBmpFileProvider, GraphicGifFileProvider, GraphicIcoFileProvider, GraphicJpegFileProvider, GraphicPdfFileProvider, GraphicPngFileProvider, GraphicTiffFileProvider and GraphicWebpFileProvider
inner	only EmlFileProvider, TarFileProvider and ZipFileProvider

15.1.8 File storages

All file operations are delegated to a separate abstraction layer of storages.

The following storages are implemented:

- `FileSystemStorage`: Does not have additional requirements.
- `PathyFileSystemStorage`: Requires pathy.
- `AzureCloudStorage`: Requires pathy and *Azure* related dependencies.
- `GoogleCloudStorage`: Requires pathy and *Google Cloud* related dependencies.
- `AWSS3Storage`: Requires pathy and *AWS S3* related dependencies.
- `SFTPStorage`: Requires paramiko and related dependencies.

15.1.8.1 Usage example with storages

15.1.8.1.1 *FileSystemStorage* example

Native file system storage. Does not have dependencies.

- `root_path`: Path to the root directory. Given the example of Django, this would be the path to the `MEDIA_ROOT` directory. It's important to know, that `root_path` will not be embedded into the string representation of the file. Only `rel_path` will.
- `rel_path`: Relative path from the root directory. Given the example of Django, this would be the rest of the path to the file.

```
import tempfile
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FS_STORAGE = FileSystemStorage(
    root_path=tempfile.gettempdir(),  # Use settings.MEDIA_ROOT for Django
    rel_path="tmp",
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=FS_STORAGE)

FS_STORAGE.exists(file)
```

15.1.8.1.2 *PathyFileSystemStorage* example

Native file system storage. Requires `pathy`.

```
import tempfile
from pathy import use_fs
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.cloud import PathyFileSystemStorage

use_fs(tempfile.gettempdir())
PATHY_FS_STORAGE = PathyFileSystemStorage(
    bucket_name="bucket_name",
    root_path="tmp",
    rel_path="sub-tmp",
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=PATHY_FS_STORAGE)

PATHY_FS_STORAGE.exists(file)
```

15.1.8.1.3 *AWSS3Storage* example

AWS S3 storage. Requires `pathy` and `boto3`.

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.aws_s3 import AWSS3Storage

S3_STORAGE = AWSS3Storage(
    bucket_name="bucket_name",
    root_path="tmp",  # Optional
    rel_path="sub-tmp",  # Optional
    # Credentials are optional too. If your AWS credentials are properly
    # set in the ~/.aws/credentials, you don't need to send them
    # explicitly.
    credentials={
        "key_id": "YOUR KEY ID",
        "key_secret": "YOUR KEY SECRET"
    },
)

FAKER = Faker()

file = TxtFileProvider(FAKER).txt_file(storage=S3_STORAGE)

S3_STORAGE.exists(file)
```

15.1.9 Testing

Simply type:

```
pytest -vrx
```

Or use tox:

```
tox
```

Or use tox to check specific env:

```
tox -e py310-django41
```

15.1.10 Writing documentation

Keep the following hierarchy.

```
=====
title
=====

header
======

sub-header
----------

sub-sub-header
~~~~~~~~~~~~~~

sub-sub-sub-header
^^^^^^^^^^^^^^^^^^

sub-sub-sub-sub-header
++++++++++++++++++++++

sub-sub-sub-sub-sub-header
**************************
```

15.1.11 License

MIT

15.1.12 Support

For security issues contact me at the e-mail given in the *Author* section.

For overall issues, go to GitHub.

15.1.13 Author

Artur Barseghyan <artur.barseghyan@gmail.com>

15.1.14 Citation

Please, use the following entry when citing faker-file in your research:

```
@software{faker-file,
  author = {Artur Barseghyan},
  title = {faker-file: Create files with fake data. In many formats. With no efforts.},
  year = {2023},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {https://github.com/barseghyanartur/faker-file},
}
```

15.2 Quick start

15.2.1 Installation

```
pip install faker-file[all]
```

15.2.2 Usage

15.2.2.1 With Faker

Imports and initialization

```
from faker import Faker
from faker_file.providers.augment_file_from_dir import (
    AugmentFileFromDirProvider,
)
from faker_file.providers.bin_file import BinFileProvider
from faker_file.providers.bmp_file import BmpFileProvider
from faker_file.providers.csv_file import CsvFileProvider
from faker_file.providers.docx_file import DocxFileProvider
from faker_file.providers.eml_file import EmlFileProvider
from faker_file.providers.epub_file import EpubFileProvider
from faker_file.providers.gif_file import GifFileProvider
from faker_file.providers.ico_file import (
    GraphicIcoFileProvider,
```

(continues on next page)

(continued from previous page)

```
    IcoFileProvider,
)
from faker_file.providers.jpeg_file import (
    GraphicJpegFileProvider,
    JpegFileProvider,
)
from faker_file.providers.mp3_file import Mp3FileProvider
from faker_file.providers.odp_file import OdpFileProvider
from faker_file.providers.ods_file import OdsFileProvider
from faker_file.providers.odt_file import OdtFileProvider
from faker_file.providers.pdf_file import (
    GraphicPdfFileProvider,
    PdfFileProvider,
)
from faker_file.providers.png_file import (
    GraphicPngFileProvider,
    PngFileProvider,
)
from faker_file.providers.pptx_file import PptxFileProvider
from faker_file.providers.random_file_from_dir import RandomFileFromDirProvider
from faker_file.providers.rtf_file import RtfFileProvider
from faker_file.providers.svg_file import SvgFileProvider
from faker_file.providers.tar_file import TarFileProvider
from faker_file.providers.tiff_file import TiffFileProvider
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.providers.webp_file import (
    GraphicWebpFileProvider,
    WebpFileProvider,
)
from faker_file.providers.xlsx_file import XlsxFileProvider
from faker_file.providers.zip_file import ZipFileProvider

FAKER = Faker()
FAKER.add_provider(AugmentFileFromDirProvider)
FAKER.add_provider(BinFileProvider)
FAKER.add_provider(BmpFileProvider)
FAKER.add_provider(CsvFileProvider)
FAKER.add_provider(DocxFileProvider)
FAKER.add_provider(EmlFileProvider)
FAKER.add_provider(EpubFileProvider)
FAKER.add_provider(GifFileProvider)
FAKER.add_provider(GraphicIcoFileProvider)
FAKER.add_provider(GraphicJpegFileProvider)
FAKER.add_provider(GraphicPdfFileProvider)
FAKER.add_provider(GraphicPngFileProvider)
FAKER.add_provider(GraphicWebpFileProvider)
FAKER.add_provider(IcoFileProvider)
FAKER.add_provider(JpegFileProvider)
FAKER.add_provider(Mp3FileProvider)
FAKER.add_provider(OdpFileProvider)
FAKER.add_provider(OdsFileProvider)
FAKER.add_provider(OdtFileProvider)
```

(continues on next page)

(continued from previous page)

```
FAKER.add_provider(PdfFileProvider)
FAKER.add_provider(PngFileProvider)
FAKER.add_provider(PptxFileProvider)
FAKER.add_provider(RandomFileFromDirProvider)
FAKER.add_provider(RtfFileProvider)
FAKER.add_provider(SvgFileProvider)
FAKER.add_provider(TarFileProvider)
FAKER.add_provider(TiffFileProvider)
FAKER.add_provider(TxtFileProvider)
FAKER.add_provider(WebpFileProvider)
FAKER.add_provider(XlsxFileProvider)
FAKER.add_provider(ZipFileProvider)
```

Usage examples

```
# augmented_file = FAKER.augment_file_from_dir(
#     source_dir_path="/tmp/tmp/",
# )
bin_file = FAKER.bin_file()
bmp_file = FAKER.bmp_file()
csv_file = FAKER.csv_file()
docx_file = FAKER.docx_file()
eml_file = FAKER.eml_file()
epub_file = FAKER.epub_file()
gif_file = FAKER.gif_file()
graphic_ico_file = FAKER.graphic_ico_file()
graphic_jpeg_file = FAKER.graphic_jpeg_file()
graphic_pdf_file = FAKER.graphic_pdf_file()
graphic_png_file = FAKER.graphic_png_file()
graphic_webp_file = FAKER.graphic_webp_file()
ico_file = FAKER.ico_file()
jpeg_file = FAKER.jpeg_file()
mp3_file = FAKER.mp3_file()
odp_file = FAKER.odp_file()
ods_file = FAKER.ods_file()
odt_file = FAKER.odt_file()
pdf_file = FAKER.pdf_file()
png_file = FAKER.png_file()
pptx_file = FAKER.pptx_file()
random_file = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
)
rtf_file = FAKER.rtf_file()
svg_file = FAKER.svg_file()
tar_file = FAKER.tar_file()
tiff_file = FAKER.tiff_file()
txt_file = FAKER.txt_file()
# webp_file = FAKER.webp_file()
xlsx_file = FAKER.xlsx_file()
zip_file = FAKER.zip_file()
```

If you just need bytes back (instead of creating the file), provide the `raw=True` argument (works with all provider classes and inner functions):

```
# augmented_raw = FAKER.augment_file_from_dir(
#     source_dir_path="/tmp/tmp/",
#     raw=True,
# )
bin_raw = FAKER.bin_file(raw=True)
bmp_raw = FAKER.bmp_file(raw=True)
csv_raw = FAKER.csv_file(raw=True)
docx_raw = FAKER.docx_file(raw=True)
eml_raw = FAKER.eml_file(raw=True)
epub_raw = FAKER.epub_file(raw=True)
gif_raw = FAKER.gif_file(raw=True)
ico_raw = FAKER.ico_file(raw=True)
jpeg_raw = FAKER.jpeg_file(raw=True)
mp3_raw = FAKER.mp3_file(raw=True)
odp_raw = FAKER.odp_file(raw=True)
ods_raw = FAKER.ods_file(raw=True)
odt_raw = FAKER.odt_file(raw=True)
pdf_raw = FAKER.pdf_file(raw=True)
png_raw = FAKER.png_file(raw=True)
pptx_raw = FAKER.pptx_file(raw=True)
random_raw = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
    raw=True,
)
rtf_raw = FAKER.rtf_file(raw=True)
svg_raw = FAKER.svg_file(raw=True)
tar_raw = FAKER.tar_file(raw=True)
tiff_raw = FAKER.tiff_file(raw=True)
txt_raw = FAKER.txt_file(raw=True)
# webp_raw = FAKER.webp_file(raw=True)
xlsx_raw = FAKER.xlsx_file(raw=True)
zip_raw = FAKER.zip_file(raw=True)
```

See the full example here

15.2.2.2 With factory_boy

Imports and initialization

```
from factory import Faker, Trait
from faker_file.providers.augment_file_from_dir import (
    AugmentFileFromDirProvider,
)
from faker_file.providers.bin_file import BinFileProvider
from faker_file.providers.bmp_file import BmpFileProvider
from faker_file.providers.csv_file import CsvFileProvider
from faker_file.providers.docx_file import DocxFileProvider
from faker_file.providers.eml_file import EmlFileProvider
from faker_file.providers.epub_file import EpubFileProvider
from faker_file.providers.ico_file import (
```

(continues on next page)

(continued from previous page)

```
    GraphicIcoFileProvider,
    IcoFileProvider,
)
from faker_file.providers.jpeg_file import (
    GraphicJpegFileProvider,
    JpegFileProvider,
)
from faker_file.providers.mp3_file import Mp3FileProvider
from faker_file.providers.odp_file import OdpFileProvider
from faker_file.providers.ods_file import OdsFileProvider
from faker_file.providers.odt_file import OdtFileProvider
from faker_file.providers.pdf_file import (
    GraphicPdfFileProvider,
    PdfFileProvider,
)
from faker_file.providers.png_file import (
    GraphicPngFileProvider,
    PngFileProvider,
)
from faker_file.providers.pptx_file import PptxFileProvider
from faker_file.providers.random_file_from_dir import RandomFileFromDirProvider
from faker_file.providers.rtf_file import RtfFileProvider
from faker_file.providers.svg_file import SvgFileProvider
from faker_file.providers.tar_file import TarFileProvider
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.providers.webp_file import (
    GraphicWebpFileProvider,
    WebpFileProvider,
)
from faker_file.providers.xlsx_file import XlsxFileProvider
from faker_file.providers.zip_file import ZipFileProvider

Faker.add_provider(AugmentFileFromDirProvider)
Faker.add_provider(BinFileProvider)
Faker.add_provider(BmpFileProvider)
Faker.add_provider(CsvFileProvider)
Faker.add_provider(DocxFileProvider)
Faker.add_provider(EmlFileProvider)
Faker.add_provider(EpubFileProvider)
Faker.add_provider(GraphicIcoFileProvider)
Faker.add_provider(GraphicJpegFileProvider)
Faker.add_provider(GraphicPdfFileProvider)
Faker.add_provider(GraphicPngFileProvider)
Faker.add_provider(GraphicWebpFileProvider)
Faker.add_provider(IcoFileProvider)
Faker.add_provider(JpegFileProvider)
Faker.add_provider(Mp3FileProvider)
Faker.add_provider(OdpFileProvider)
Faker.add_provider(OdsFileProvider)
Faker.add_provider(OdtFileProvider)
Faker.add_provider(PdfFileProvider)
Faker.add_provider(PngFileProvider)
```

(continues on next page)

(continued from previous page)

```
Faker.add_provider(PptxFileProvider)
Faker.add_provider(RandomFileFromDirProvider)
Faker.add_provider(RtfFileProvider)
Faker.add_provider(SvgFileProvider)
Faker.add_provider(TarFileProvider)
Faker.add_provider(TxtFileProvider)
Faker.add_provider(WebpFileProvider)
Faker.add_provider(XlsxFileProvider)
Faker.add_provider(ZipFileProvider)
```

upload/models.py

```
from django.db import models

class Upload(models.Model):
    """Upload model."""

    name = models.CharField(max_length=255, unique=True)
    description = models.TextField(null=True, blank=True)

    # File
    file = models.FileField(null=True)
```

See the full example here

upload/factories.py

```
from django.conf import settings
from factory import Faker, Trait
from factory.django import DjangoModelFactory
from faker_file.storages.filesystem import FileSystemStorage

from upload.models import Upload

# Define a file storage, because we need to customize things in
# order for it to work with Django.
STORAGE = FileSystemStorage(root_path=settings.MEDIA_ROOT, rel_path="tmp")

# Factories


class UploadFactory(DjangoModelFactory):
    """Upload factory."""

    name = Faker("text", max_nb_chars=100)
    description = Faker("text", max_nb_chars=1000)

    class Meta:
        model = Upload

    class Params:
        bin_file = Trait(file=Faker("bin_file", storage=STORAGE))
        bmp_file = Trait(file=Faker("bmp_file", storage=STORAGE))
```

(continues on next page)

(continued from previous page)

```
        csv_file = Trait(file=Faker("csv_file", storage=STORAGE))
        docx_file = Trait(file=Faker("docx_file", storage=STORAGE))
        eml_file = Trait(file=Faker("eml_file", storage=STORAGE))
        epub_file = Trait(file=Faker("epub_file", storage=STORAGE))
        ico_file = Trait(file=Faker("ico_file", storage=STORAGE))
        jpeg_file = Trait(file=Faker("jpeg_file", storage=STORAGE))
        mp3_file = Trait(file=Faker("mp3_file", storage=STORAGE))
        odp_file = Trait(file=Faker("odp_file", storage=STORAGE))
        ods_file = Trait(file=Faker("ods_file", storage=STORAGE))
        odt_file = Trait(file=Faker("odt_file", storage=STORAGE))
        pdf_file = Trait(file=Faker("pdf_file", storage=STORAGE))
        png_file = Trait(file=Faker("png_file", storage=STORAGE))
        pptx_file = Trait(file=Faker("pptx_file", storage=STORAGE))
        rtf_file = Trait(file=Faker("rtf_file", storage=STORAGE))
        svg_file = Trait(file=Faker("svg_file", storage=STORAGE))
        tar_file = Trait(file=Faker("tar_file", storage=STORAGE))
        txt_file = Trait(file=Faker("txt_file", storage=STORAGE))
        webp_file = Trait(file=Faker("webp_file", storage=STORAGE))
        xlsx_file = Trait(file=Faker("xlsx_file", storage=STORAGE))
        zip_file = Trait(file=Faker("zip_file", storage=STORAGE))
```

Usage example

```
UploadFactory(bin_file=True)  # Upload with BIN file
UploadFactory(docx_file=True)  # Upload with DOCX file
UploadFactory(jpeg_file=True)  # Upload with JPEG file
UploadFactory(zip_file=True)  # Upload with ZIP file
```

See the full example here

15.3 Recipes

15.3.1 When using with `Faker`

When using with `Faker`, there are two ways of using the providers.

15.3.1.1 Imports and initializations

Recommended way

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

# Usage example
txt_file = FAKER.txt_file(content="Lorem ipsum")
```

See the full example here

But this works too

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()

# Usage example
txt_file = TxtFileProvider(FAKER).txt_file(content="Lorem ipsum")
```

See the full example here

Throughout documentation we will be mixing these approaches.

15.3.1.2 Create a TXT file with static content

- Content of the file is `Lorem ipsum`.

```
txt_file = FAKER.txt_file(content="Lorem ipsum")
```

See the full example here

15.3.1.3 Create a DOCX file with dynamically generated content

- Content is generated dynamically.
- Content is limited to 1024 chars.
- Wrap lines after 80 chars.
- Prefix the filename with `zzz`.

```
docx_file = FAKER.docx_file(
    prefix="zzz",
    max_nb_chars=1_024,
    wrap_chars_after=80,
)
```

See the full example here

15.3.1.4 Create a ZIP file consisting of TXT files with static content

- 5 TXT files in the ZIP archive (default value is 5).
- Content of all files is `Lorem ipsum`.

```
zip_file = FAKER.zip_file(
    options={"create_inner_file_args": {"content": "Lorem ipsum"}}
)
```

See the full example here

15.3.1.5 Create a ZIP file consisting of 3 DOCX files with dynamically generated content

- 3 DOCX files in the ZIP archive.
- Content is generated dynamically.
- Content is limited to 1024 chars.
- Prefix the filenames in archive with `xxx_`.
- Prefix the filename of the archive itself with `zzz`.
- Inside the ZIP, put all files in directory `yyy`.

```
from faker_file.providers.helpers.inner import create_inner_docx_file

zip_file = FAKER.zip_file(
    prefix="zzz",
    options={
        "count": 3,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "xxx_",
            "max_nb_chars": 1_024,
        },
        "directory": "yyy",
    },
)
```

See the full example here

15.3.1.6 Create a ZIP file of 9 DOCX files with content generated from template

- 9 DOCX files in the ZIP archive.
- Content is generated dynamically from given template.

```
from faker_file.providers.helpers.inner import create_inner_docx_file

TEMPLATE = "Hey {{name}},\n{{text}},\nBest regards\n{{name}}"

zip_file = FAKER.zip_file(
    options={
        "count": 9,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "content": TEMPLATE,
        },
    }
)
```

See the full example here

15.3.1.7 Create a nested ZIP file

Create a ZIP file which contains 5 ZIP files which contain 5 ZIP files which contain 5 DOCX files.

- 5 ZIP files in the ZIP archive.
- Content is generated dynamically.
- Prefix the filenames in archive with `nested_level_1_`.
- Prefix the filename of the archive itself with `nested_level_0_`.
- Each of the ZIP files inside the ZIP file in their turn contains 5 other ZIP files, prefixed with `nested_level_2_`, which in their turn contain 5 DOCX files.

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_zip_file,
)

zip_file = FAKER.zip_file(
    prefix="nested_level_0_",
    options={
        "create_inner_file_func": create_inner_zip_file,
        "create_inner_file_args": {
            "prefix": "nested_level_1_",
            "options": {
                "create_inner_file_func": create_inner_zip_file,
                "create_inner_file_args": {
                    "prefix": "nested_level_2_",
                    "options": {
                        "create_inner_file_func": create_inner_docx_file,
                    },
                },
            },
        },
    },
)
```

See the full example `here`

15.3.1.8 Create a ZIP file with variety of different file types within

- 50 files in the ZIP archive (limited to DOCX, EPUB and TXT types).
- Content is generated dynamically.
- Prefix the filename of the archive itself with `zzz_archive_`.
- Inside the ZIP, put all files in directory `zzz`.

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)
```

(continues on next page)

(continued from previous page)

```
from faker_file.storages.filesystem import FileSystemStorage

STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}

zip_file = FAKER.zip_file(
    prefix="zzz_archive_",
    options={
        "count": 50,
        "create_inner_file_func": fuzzy_choice_create_inner_file,
        "create_inner_file_args": {
            "func_choices": [
                (create_inner_docx_file, kwargs),
                (create_inner_epub_file, kwargs),
                (create_inner_txt_file, kwargs),
            ],
        },
        "directory": "zzz",
    },
)
```

See the full example here

15.3.1.9 Another way to create a ZIP file with variety of different file types within

- 3 files in the ZIP archive (1 DOCX, and 2 XML types).
- Content is generated dynamically.
- Filename of the archive itself is `alice-looking-through-the-glass.zip`.
- Files inside the archive have fixed name (passed with `basename` argument).

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_xml_file,
    list_create_inner_file,
)

zip_file = FAKER.zip_file(
    basename="alice-looking-through-the-glass",
    options={
        "create_inner_file_func": list_create_inner_file,
        "create_inner_file_args": {
            "func_list": [
                (create_inner_docx_file, {"basename": "doc"}),
                (create_inner_xml_file, {"basename": "doc_metadata"}),
                (create_inner_xml_file, {"basename": "doc_isbn"}),
            ],
        },
    },
)
```

See the full example here

Note, that `count` argument (not shown in the example, but commonly accepted by inner functions) will be simply ignored here.

15.3.1.10 Create an EML file consisting of TXT files with static content

- 5 TXT files in the EML email (default value is 5).
- Content of all files is `Lorem ipsum`.

```
from faker_file.providers.eml_file import EmlFileProvider

FAKER.add_provider(EmlFileProvider)

eml_file = FAKER.eml_file(
    options={"create_inner_file_args": {"content": "Lorem ipsum"}}
)
```

See the full example here

15.3.1.11 Create a EML file consisting of 3 DOCX files with dynamically generated content

- 3 DOCX files in the EML email.
- Content is generated dynamically.
- Content is limited to 1024 chars.
- Prefix the filenames in email with `xxx_`.
- Prefix the filename of the email itself with `zzz`.

```
from faker_file.providers.helpers.inner import create_inner_docx_file

eml_file = FAKER.eml_file(
    prefix="zzz",
    options={
        "count": 3,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "xxx_",
            "max_nb_chars": 1_024,
        },
    },
)
```

See the full example here

15.3.1.12 Create a nested EML file

Create a EML file which contains 5 EML files which contain 5 EML files which contain 5 DOCX files.

- 5 EML files in the EML file.
- Content is generated dynamically.
- Prefix the filenames in EML email with `nested_level_1_`.
- Prefix the filename of the EML email itself with `nested_level_0_`.
- Each of the EML files inside the EML file in their turn contains 5 other EML files, prefixed with `nested_level_2_`, which in their turn contain 5 DOCX files.

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_eml_file,
)

eml_file = FAKER.eml_file(
    prefix="nested_level_0_",
    options={
        "create_inner_file_func": create_inner_eml_file,
        "create_inner_file_args": {
            "prefix": "nested_level_1_",
            "options": {
                "create_inner_file_func": create_inner_eml_file,
                "create_inner_file_args": {
                    "prefix": "nested_level_2_",
                    "options": {
                        "create_inner_file_func": create_inner_docx_file,
                    },
                },
            },
        },
    },
)
```

See the full example `here`

15.3.1.13 Create an EML file with variety of different file types within

- 10 files in the EML file (limited to DOCX, EPUB and TXT types).
- Content is generated dynamically.
- Prefix the filename of the EML itself with `zzz`.

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)
```

(continues on next page)

(continued from previous page)

```
eml_file = FAKER.eml_file(
    prefix="zzz",
    options={
        "count": 10,
        "create_inner_file_func": fuzzy_choice_create_inner_file,
        "create_inner_file_args": {
            "func_choices": [
                (create_inner_docx_file, kwargs),
                (create_inner_epub_file, kwargs),
                (create_inner_txt_file, kwargs),
            ],
        },
    },
)
```

See the full example here

15.3.1.14 Create a PDF file with predefined template containing dynamic fixtures

- Content template is predefined and contains dynamic fixtures.
- Wrap lines after 80 chars.

```
from faker_file.providers.pdf_file import PdfFileProvider

FAKER.add_provider(PdfFileProvider)

TEMPLATE = """
{{date}} {{city}}, {{country}}

Hello {{name}},

{{text}} {{text}} {{text}}

{{text}} {{text}} {{text}}

{{text}} {{text}} {{text}}

Address: {{address}}

Best regards,

{{name}}
{{address}}
{{phone_number}}
"""

pdf_file = FAKER.pdf_file(content=TEMPLATE, wrap_chars_after=80)
```

See the full example here

15.3.1.15 Create a DOCX file with table and image using `DynamicTemplate`

When pre-defined templating and dynamic fixtures are not enough and full control is needed, you can use `DynamicTemplate` wrapper. It takes a list of content modifiers (tuples): `(func: Callable, kwargs: dict)`. Each callable should accept the following arguments:

- *provider*: Faker `Generator` instance or `Faker` instance.
- *document*: Document instance. Implementation specific.
- *data*: Dictionary. Used primarily for observability.
- *counter*: Integer. Index number of the content modifier.
- ***kwargs*: Dictionary. Useful to pass implementation-specific arguments.

The following example shows how to generate a DOCX file with paragraph, table and image.

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.docx_file import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a DOCX file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
docx_file = FAKER.docx_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    )
)
```

See the full example `here`

15.3.1.16 Create a ODT file with table and image using `DynamicTemplate`

Similarly to previous section, the following example shows how to generate an ODT file with table and image.

```
from faker_file.contrib.odt_file import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)
```

(continues on next page)

(continued from previous page)

```
from faker_file.providers.odt_file import OdtFileProvider

FAKER.add_provider(OdtFileProvider)  # Register OdtFileProvider

# Create a ODT file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
odt_file = FAKER.odt_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    )
)
```

See the full example here

15.3.1.17 Create a PDF using *reportlab* generator

```
from faker_file.providers.pdf_file.generators.reportlab_generator import (
    ReportlabPdfGenerator,
)

pdf_file = FAKER.pdf_file(pdf_generator_cls=ReportlabPdfGenerator)
```

See the full example here

15.3.1.18 Create a PDF using *pdfkit* generator

Note, that at the moment, `pdfkit` is the default generator. However, you could set it explicitly as follows:

```
from faker_file.providers.pdf_file.generators.pdfkit_generator import (
    PdfkitPdfGenerator,
)

pdf_file = FAKER.pdf_file(pdf_generator_cls=PdfkitPdfGenerator)
```

See the full example here

15.3.1.19 Create a graphic PDF file using *Pillow*

Graphic PDF file does not contain text. Don't use it when you need text based content. However, sometimes you just need a valid file in PDF format, without caring much about the content. That's where a GraphicPdfFileProvider comes to rescue:

```
from faker_file.providers.pdf_file import GraphicPdfFileProvider

FAKER.add_provider(GraphicPdfFileProvider)

pdf_file = FAKER.graphic_pdf_file()
```

See the full example here

The generated file will contain a random graphic (consisting of lines and shapes of different colours). One of the most useful arguments supported is `size`.

```
pdf_file = FAKER.graphic_pdf_file(size=(800, 800))
```

See the full example here

15.3.1.20 Graphic providers

Graphic file providers does not contain text. Don't use it when you need text based content. However, sometimes you just need a valid image file with graphics of a certain size. That's where graphic file providers help.

Supported files formats are: *BMP*, *GIF*, *ICO*, *JPEG*, *PDF*, *PNG*, *SVG TIFF* and *WEBP*.

15.3.1.20.1 Create an ICO file

```
from faker_file.providers.ico_file import GraphicIcoFileProvider

FAKER.add_provider(GraphicIcoFileProvider)

ico_file = FAKER.graphic_ico_file(size=(800, 800))
```

See the full example here

15.3.1.20.2 Create a JPEG file

```
from faker_file.providers.jpeg_file import GraphicJpegFileProvider

FAKER.add_provider(GraphicJpegFileProvider)

jpeg_file = FAKER.graphic_jpeg_file(size=(800, 800))
```

See the full example here

15.3.1.20.3 Create a PNG file

```
from faker_file.providers.png_file import GraphicPngFileProvider

FAKER.add_provider(GraphicPngFileProvider)

png_file = FAKER.graphic_png_file(size=(800, 800))
```

See the full example here

15.3.1.20.4 Create a WEBP file

```
from faker_file.providers.webp_file import GraphicWebpFileProvider

FAKER.add_provider(GraphicWebpFileProvider)

webp_file = FAKER.graphic_webp_file(size=(800, 800))
```

See the full example here

15.3.1.21 Create a MP3 file

```
from faker_file.providers.mp3_file import Mp3FileProvider

FAKER.add_provider(Mp3FileProvider)

mp3_file = FAKER.mp3_file()
```

See the full example here

15.3.1.22 Create a MP3 file by explicitly specifying MP3 generator class

15.3.1.22.1 Google Text-to-Speech

```
from faker_file.providers.mp3_file.generators.gtts_generator import (
    GttsMp3Generator,
)

mp3_file = FAKER.mp3_file(mp3_generator_cls=GttsMp3Generator)
```

See the full example here

You can tune arguments too:

```
mp3_file = FAKER.mp3_file(
    mp3_generator_cls=GttsMp3Generator,
    mp3_generator_kwargs={
        "lang": "en",
```

(continues on next page)

(continued from previous page)

```
        "tld": "co.uk",
    },
)
```

See the full example here

Refer to https://gtts.readthedocs.io/en/latest/module.html#languages-gtts-lang for list of accepted values for `lang` argument.

Refer to https://gtts.readthedocs.io/en/latest/module.html#localized-accents for list of accepted values for `tld` argument.

15.3.1.22.2 Microsoft Edge Text-to-Speech

```
from faker_file.providers.mp3_file.generators.edge_tts_generator import (
    EdgeTtsMp3Generator,
)

mp3_file = FAKER.mp3_file(mp3_generator_cls=EdgeTtsMp3Generator)
```

See the full example here

You can tune arguments too:

```
mp3_file = FAKER.mp3_file(
    mp3_generator_cls=EdgeTtsMp3Generator,
    mp3_generator_kwargs={
        "voice": "en-GB-LibbyNeural",
    },
)
```

See the full example here

Run `edge-tts -l` from terminal for list of available voices.

15.3.1.23 Create a MP3 file with custom MP3 generator

Default MP3 generator class is `GttsMp3Generator` which uses Google Text-to-Speech services to generate an MP3 file from given or randomly generated text. It does not require additional services to run and the only dependency here is the `gtts` package. You can however implement your own custom MP3 generator class and pass it to te `mp3_file` method in `mp3_generator_cls` argument instead of the default `GttsMp3Generator`. Read about quotas of Google Text-to-Speech services here.

Usage with custom MP3 generator class.

```
from faker_file.providers.base.mp3_generator import BaseMp3Generator
from marytts import MaryTTS  # Imaginary `marytts` Python library

# Define custom MP3 generator
class MaryTtsMp3Generator(BaseMp3Generator):
    locale: str = "cmu-rms-hsmm"
```

(continues on next page)

(continued from previous page)

```
    voice: str = "en_US"

    def handle_kwargs(self, **kwargs) -> None:
        # Since it's impossible to unify all TTS systems it's allowed
        # to pass arbitrary arguments to the `BaseMp3Generator`
        # constructor. Each implementation class contains its own
        # additional tuning arguments. Check the source code of the
        # implemented MP3 generators as an example.
        if "locale" in kwargs:
            self.locale = kwargs["locale"]
        if "voice" in kwargs:
            self.voice = kwargs["voice"]

    def generate(self) -> bytes:
        # Your implementation here. Note, that `self.content`
        # in this context is the text to make MP3 from.
        # `self.generator` would be the `Faker` or `Generator`
        # instance from which you could extract information on
        # active locale.
        # What comes below is pseudo implementation.
        mary_tts = MaryTTS(locale=self.locale, voice=self.voice)
        return mary_tts.synth_mp3(self.content)


# Generate MP3 file from random text
mp3_file = FAKER.mp3_file(
    mp3_generator_cls=MaryTtsMp3Generator,
)
```

See the full example here

See exact implementation of marytts_mp3_generator in the examples.

15.3.1.24 Pick a random file from a directory given

- Create an exact copy of the randomly picked file under a different name.
- Prefix of the destination file would be `zzz`.
- `source_dir_path` is the absolute path to the directory to pick files from.

```
from faker_file.providers.random_file_from_dir import RandomFileFromDirProvider

FAKER.add_provider(RandomFileFromDirProvider)

# We assume that directory "/tmp/tmp/" exists and contains files.
random_file = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
    prefix="zzz",
)
```

See the full example here

15.3.1.25 File from path given

- Create an exact copy of a file under a different name.
- Prefix of the destination file would be `zzz`.
- `path` is the absolute path to the file to copy.

```
from faker_file.providers.file_from_path import FileFromPathProvider

FAKER.add_provider(FileFromPathProvider)

# We assume that the file "/tmp/tmp/file.docx" exists.
docx_file = FAKER.file_from_path(
    path="/tmp/tmp/file.docx",
    prefix="zzz",
)
```

See the full example `here`

15.3.1.26 Generate a file of a certain size

The only two file types for which it is easy to foresee the file size are BIN and TXT. Note, that size of BIN files is always exact, while for TXT it is approximate.

15.3.1.26.1 BIN

```
from faker_file.providers.bin_file import BinFileProvider

FAKER.add_provider(BinFileProvider)

bin_file = FAKER.bin_file(length=1024**2)  # 1 Mb
bin_file = FAKER.bin_file(length=3 * 1024**2)  # 3 Mb
bin_file = FAKER.bin_file(length=10 * 1024**2)  # 10 Mb

bin_file = FAKER.bin_file(length=1024)  # 1 Kb
bin_file = FAKER.bin_file(length=3 * 1024)  # 3 Kb
bin_file = FAKER.bin_file(length=10 * 1024)  # 10 Kb
```

See the full example `here`

15.3.1.26.2 TXT

```
from faker_file.providers.txt_file import TxtFileProvider

FAKER.add_provider(TxtFileProvider)

txt_file = FAKER.txt_file(max_nb_chars=1024**2)  # 1 Mb
txt_file = FAKER.txt_file(max_nb_chars=3 * 1024**2)  # 3 Mb
txt_file = FAKER.txt_file(max_nb_chars=10 * 1024**2)  # 10 Mb
```

(continues on next page)

(continued from previous page)

```
txt_file = FAKER.txt_file(max_nb_chars=1024)  # 1 Kb
txt_file = FAKER.txt_file(max_nb_chars=3 * 1024)  # 3 Kb
txt_file = FAKER.txt_file(max_nb_chars=10 * 1024)  # 10 Kb
```

See the full example here

15.3.1.27 Generate a files using multiprocessing

15.3.1.27.1 Generate 10 DOCX files

- Use template.
- Generate 10 DOCX files.

```
from multiprocessing import Pool
from faker_file.providers.helpers.inner import create_inner_docx_file
from faker_file.storages.filesystem import FileSystemStorage

STORAGE = FileSystemStorage()

# Document template
TEMPLATE = "Hey {{name}},\n{{text}},\nBest regards\n{{name}}"

with Pool(processes=2) as pool:
    for _ in range(10):  # Number of times we want to run our function
        pool.apply_async(
            create_inner_docx_file,
            # Apply async doesn't support kwargs. We have to pass all
            # arguments.
            [STORAGE, "mp", FAKER, None, None, TEMPLATE],
        )
    pool.close()
    pool.join()
```

See the full example here

15.3.1.27.2 Randomize the file format

```
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_pdf_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)

kwargs = {"storage": STORAGE, "generator": FAKER, "content": TEMPLATE}

with Pool(processes=2) as pool:
    for _ in range(10):  # Number of times we want to run our function
```

(continues on next page)

(continued from previous page)

```
        pool.apply_async(
            fuzzy_choice_create_inner_file,
            [
                [
                    (create_inner_docx_file, kwargs),
                    (create_inner_epub_file, kwargs),
                    (create_inner_pdf_file, kwargs),
                    (create_inner_txt_file, kwargs),
                ]
            ],
        )
    pool.close()
    pool.join()
```

See the full example here

15.3.1.28 Generating files from existing documents using NLP augmentation

See the following example:

```
from faker_file.providers.augment_file_from_dir import (
    AugmentFileFromDirProvider,
)

FAKER.add_provider(AugmentFileFromDirProvider)

# We assume that directory "/tmp/tmp/" exists and contains
# files of `DOCX`, `EML`, `EPUB`, `ODT`, `PDF`, `RTF` or `TXT`
# formats.
augmented_file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/",
)
```

See the full example here

Generated file will resemble text of the original document, but will not be the same. This is useful when you don't want to test on text generated by `Faker`, but rather something that makes more sense for your use case, still want to ensure uniqueness of the documents.

The following file types are supported:

- DOCX
- EML
- EPUB
- ODT
- PDF
- RTF
- TXT

By default, all supported files are eligible for random selection. You could however narrow that list by providing `extensions` argument:

```
# We assume that directory "/tmp/tmp/" exists and contains
# files of `DOCX` and `ODT` formats.
augmented_file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/",
    extensions={"docx", "odt"},  # Pick only DOCX or ODT
)
```

See the full example `here`

Actual augmentation of texts is delegated to an abstraction layer of text augmenters. Currently, two augmenters are implemented. Default one is based on textaugment (which is in its' turn based on nltk) is very lightweight and speedy, but produces less accurate results. Another one is based on nlpaug, which is way more sophisticated, but at the cost of speed.

15.3.1.29 nlpaug augmenter

By default `bert-base-multilingual-cased` model is used, which is pretrained on the top 104 languages with the largest Wikipedia using a masked language modeling (MLM) objective. If you want to use a different model, specify the proper identifier in the `model_path` argument. Some well working options for `model_path` are:

- `bert-base-multilingual-cased`
- `bert-base-multilingual-uncased`
- `bert-base-cased`
- `bert-base-uncased`
- `bert-base-german-cased`
- `GroNLP/bert-base-dutch-cased`

```
from faker_file.providers.augment_file_from_dir.augmenters import (
    nlpaug_augmenter,
)

# We assume that directory "/tmp/tmp/" exists and contains
# files of `DOCX`, `EML`, `EPUB`, `ODT`, `PDF`, `RTF` or `TXT`
# formats.
augmented_file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/",
    text_augmenter_cls=nlpaug_augmenter.ContextualWordEmbeddingsAugmenter,
    text_augmenter_kwargs={
        "model_path": "bert-base-cased",
        "action": "substitute",  # or "insert"
    },
)
```

See the full example `here`

Refer to `nlpaug` docs and check *Textual augmenters* examples.

15.3.1.30 textaugment augmenter

```
from faker_file.providers.augment_file_from_dir.augmenters import (
    textaugment_augmenter,
)

# We assume that directory "/tmp/tmp/" exists and contains
# files of `DOCX`, `EML`, `EPUB`, `ODT`, `PDF`, `RTF` or `TXT`
# formats. Valid values for `action` are: "random_deletion",
# "random_insertion", "random_swap" and "synonym_replacement" (default).
augmented_file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/",
    text_augmenter_cls=textaugment_augmenter.EDATextaugmentAugmenter,
    text_augmenter_kwargs={
        "action": "synonym_replacement",
    },
)
```

See the full example here

15.3.1.31 Using *raw=True* features in tests

If you pass `raw=True` argument to any provider or inner function, instead of creating a file, you will get `bytes` back (or to be totally correct, `bytes`-like object `BytesValue`, which is basically bytes enriched with meta-data). You could then use the `bytes` content of the file to build a test payload as shown in the example test below:

```
class UploadTestCase(TestCase):
    """Upload test case."""

    def test_create_docx_upload(self) -> None:
        """Test create an Upload."""
        url = reverse("api:upload-list")

        raw = FAKER.docx_file(raw=True)
        test_file = BytesIO(raw)
        test_file.name = os.path.basename(raw.data["filename"])

        payload = {
            "name": FAKER.word(),
            "description": FAKER.paragraph(),
            "file": test_file,
        }

        response = self.client.post(url, payload, format="json")

        # Test if request is handled properly (HTTP 201)
        self.assertEqual(response.status_code, HTTP_201_CREATED)

        test_upload = Upload.objects.get(id=response.data["id"])

        # Test if the name is properly recorded
        self.assertEqual(str(test_upload.name), payload["name"])
```

(continues on next page)

(continued from previous page)

```
        # Test if file name recorded properly
        self.assertEqual(str(test_upload.file.name), test_file.name)
```

See the full example here

15.3.1.32 Create a HTML file from predefined template

If you want to generate a file in a format that is not (yet) supported, you can try to use `GenericFileProvider`. In the following example, an HTML file is generated from a template.

```
from faker_file.providers.generic_file import GenericFileProvider

FAKER.add_provider(GenericFileProvider)

generic_file = FAKER.generic_file(
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
)
```

See the full example here

15.3.1.33 Working with storages

15.3.1.33.1 AWS S3 storage

```
from faker_file.storages.aws_s3 import AWSS3Storage

AWS_S3_STORAGE = AWSS3Storage(
    bucket_name="your-bucket-name",
    root_path="",
    rel_path="",
)

txt_file = FAKER.txt_file(storage=AWS_S3_STORAGE)
```

See the full example here

Depending on the ORM or framework you're using, you might want to tweak the `root_path` and `rel_path` values. Especially if you store files in directories (like `your-bucket-name/path/to/the/file.ext`).

For instance, if you use `Django` and `django-storages`, and want to store the files inside `/user/uploads` directory the following would be correct:

```
AWS_S3_STORAGE = AWSS3Storage(
    bucket_name="your-bucket-name",
    root_path="",
    rel_path="user/uploads",
)
```

See the full example here

15.3.1.33.2 Google Cloud Storage

```
from faker_file.storages.google_cloud_storage import GoogleCloudStorage

GC_STORAGE = GoogleCloudStorage(
    bucket_name="your-bucket-name",
    root_path="",
    rel_path="",
)

# txt_file = FAKER.txt_file(storage=GC_STORAGE)
```

See the full example here

Similarly to `AWSS3Storage`, if you use `Django` and `django-storages`, and want to store the files inside `/user/uploads` directory the following would be correct:

```
GC_STORAGE = GoogleCloudStorage(
    bucket_name="your-bucket-name",
    root_path="",
    rel_path="user/uploads",
)
```

See the full example here

15.3.1.33.3 SFTP storage

```
from faker_file.storages.sftp_storage import SFTPStorage

SFTP_STORAGE = SFTPStorage(
    host="your-sftp-host.domain",
    port=22,
    username="your-sftp-username",
    password="your-sftp-password",
    root_path="/dir-name",
)

# txt_file = FAKER.txt_file(storage=SFTP_STORAGE)
```

See the full example here

15.3.2 When using with `Django` (and `factory_boy`)

When used with Django (to generate fake data with `factory_boy` factories), the `root_path` argument of the correspondent file storage shall be provided. Otherwise (although no errors will be triggered) the generated files will reside outside the `MEDIA_ROOT` directory (by default in `/tmp/` on Linux) and further operations with those files through Django will cause `SuspiciousOperation` exception.

15.3.2.1 Basic example

15.3.2.1.1 Imaginary `Django` model

```
class Upload(models.Model):
    """Upload model."""

    name = models.CharField(max_length=255, unique=True)
    description = models.TextField(null=True, blank=True)

    # File
    file = models.FileField(null=True)
```

See the full example here

15.3.2.1.2 Correspondent `factory_boy` factory

```
from django.conf import settings
from factory import Faker, Trait
from factory.django import DjangoModelFactory

# Import all providers we want to use
from faker_file.providers.bin_file import BinFileProvider
from faker_file.providers.csv_file import CsvFileProvider
from faker_file.providers.docx_file import DocxFileProvider
from faker_file.providers.eml_file import EmlFileProvider
from faker_file.providers.epub_file import EpubFileProvider
from faker_file.providers.ico_file import IcoFileProvider
from faker_file.providers.jpeg_file import JpegFileProvider
from faker_file.providers.mp3_file import Mp3FileProvider
from faker_file.providers.ods_file import OdsFileProvider
from faker_file.providers.odt_file import OdtFileProvider
from faker_file.providers.pdf_file import PdfFileProvider
from faker_file.providers.png_file import PngFileProvider
from faker_file.providers.pptx_file import PptxFileProvider
from faker_file.providers.rtf_file import RtfFileProvider
from faker_file.providers.svg_file import SvgFileProvider
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.providers.webp_file import WebpFileProvider
from faker_file.providers.xlsx_file import XlsxFileProvider
from faker_file.providers.zip_file import ZipFileProvider

# Import file storage, because we need to customize things in order for it
# to work with Django.
from faker_file.storages.filesystem import FileSystemStorage

from upload.models import Upload

# Add all providers we want to use
Faker.add_provider(BinFileProvider)
Faker.add_provider(CsvFileProvider)
```

(continues on next page)

(continued from previous page)

```
Faker.add_provider(DocxFileProvider)
Faker.add_provider(EmlFileProvider)
Faker.add_provider(EpubFileProvider)
Faker.add_provider(IcoFileProvider)
Faker.add_provider(JpegFileProvider)
Faker.add_provider(Mp3FileProvider)
Faker.add_provider(OdsFileProvider)
Faker.add_provider(OdtFileProvider)
Faker.add_provider(PdfFileProvider)
Faker.add_provider(PngFileProvider)
Faker.add_provider(PptxFileProvider)
Faker.add_provider(RtfFileProvider)
Faker.add_provider(SvgFileProvider)
Faker.add_provider(TxtFileProvider)
Faker.add_provider(WebpFileProvider)
Faker.add_provider(XlsxFileProvider)
Faker.add_provider(ZipFileProvider)

# Define a file storage. When working with Django and FileSystemStorage
# you need to set the value of `root_path` argument to
# `settings.MEDIA_ROOT`.
STORAGE = FileSystemStorage(root_path=settings.MEDIA_ROOT, rel_path="tmp")


class UploadFactory(DjangoModelFactory):
    """Upload factory."""

    name = Faker("text", max_nb_chars=100)
    description = Faker("text", max_nb_chars=1000)

    class Meta:
        model = Upload

    class Params:
        bin_file = Trait(file=Faker("bin_file", storage=STORAGE))
        csv_file = Trait(file=Faker("csv_file", storage=STORAGE))
        docx_file = Trait(file=Faker("docx_file", storage=STORAGE))
        eml_file = Trait(file=Faker("eml_file", storage=STORAGE))
        epub_file = Trait(file=Faker("epub_file", storage=STORAGE))
        ico_file = Trait(file=Faker("ico_file", storage=STORAGE))
        jpeg_file = Trait(file=Faker("jpeg_file", storage=STORAGE))
        mp3_file = Trait(file=Faker("mp3_file", storage=STORAGE))
        ods_file = Trait(file=Faker("ods_file", storage=STORAGE))
        odt_file = Trait(file=Faker("odt_file", storage=STORAGE))
        pdf_file = Trait(file=Faker("pdf_file", storage=STORAGE))
        png_file = Trait(file=Faker("png_file", storage=STORAGE))
        pptx_file = Trait(file=Faker("pptx_file", storage=STORAGE))
        rtf_file = Trait(file=Faker("rtf_file", storage=STORAGE))
        svg_file = Trait(file=Faker("svg_file", storage=STORAGE))
        txt_file = Trait(file=Faker("txt_file", storage=STORAGE))
        webp_file = Trait(file=Faker("webp_file", storage=STORAGE))
        xlsx_file = Trait(file=Faker("xlsx_file", storage=STORAGE))
```

(continues on next page)

(continued from previous page)

```
        zip_file = Trait(file=Faker("zip_file", storage=STORAGE))
```

And then somewhere in your code:

```
UploadFactory(bin_file=True)  # Upload with BIN file
UploadFactory(docx_file=True)  # Upload with DOCX file
UploadFactory(jpeg_file=True)  # Upload with JPEG file
UploadFactory(zip_file=True)  # Upload with ZIP file
```

See the full example here

15.3.2.2 Randomize provider choice

```
from random import choice
from factory import Faker, LazyAttribute, Trait
from faker import Faker as OriginalFaker

FAKER = OriginalFaker()
FAKER.add_provider(BinFileProvider)
FAKER.add_provider(CsvFileProvider)
FAKER.add_provider(DocxFileProvider)
FAKER.add_provider(EmlFileProvider)
FAKER.add_provider(EpubFileProvider)
FAKER.add_provider(IcoFileProvider)
FAKER.add_provider(JpegFileProvider)
FAKER.add_provider(Mp3FileProvider)
FAKER.add_provider(OdsFileProvider)
FAKER.add_provider(OdtFileProvider)
FAKER.add_provider(PdfFileProvider)
FAKER.add_provider(PngFileProvider)
FAKER.add_provider(PptxFileProvider)
FAKER.add_provider(RtfFileProvider)
FAKER.add_provider(SvgFileProvider)
FAKER.add_provider(TxtFileProvider)
FAKER.add_provider(WebpFileProvider)
FAKER.add_provider(XlsxFileProvider)
FAKER.add_provider(ZipFileProvider)


def random_file_generator(*args, **kwargs):
    random_provider = choice(
        [
            "bin_file",
            "csv_file",
            "docx_file",
            "eml_file",
            "epub_file",
            "ico_file",
            "jpeg_file",
```

(continues on next page)

(continued from previous page)

```
            "mp3_file",
            "ods_file",
            "odt_file",
            "pdf_file",
            "png_file",
            "pptx_file",
            "rtf_file",
            "svg_file",
            "txt_file",
            "webp_file",
            "xlsx_file",
            "zip_file",
        ]
    )
    func = getattr(FAKER, random_provider)
    return func(storage=STORAGE)


class UploadFactory(DjangoModelFactory):
    """Upload factory."""

    name = Faker("text", max_nb_chars=100)
    description = Faker("text", max_nb_chars=1000)

    class Meta:
        model = Upload

    class Params:
        random_file = Trait(file=LazyAttribute(random_file_generator))
```

And then somewhere in your code:

```
# Upload with randon file
upload = UploadFactory(random_file=True)
```

See the full example here

15.3.2.3 Use a different locale

```
    Faker.add_provider(OdtFileProvider)
    upload = UploadFactory()
```

See the full example here

15.3.2.4 Other Django usage examples

Faker example with AWS S3 storage

```
from faker import Faker
from faker_file.storages.aws_s3 import AWSS3Storage

STORAGE = AWSS3Storage(
    bucket_name=settings.AWS_STORAGE_BUCKET_NAME,
    root_path="",
    rel_path="",
)

FAKER = Faker()
FAKER.add_provider(PdfFileProvider)

pdf_file = FAKER.pdf_file(storage=STORAGE)
```

See the full example here

---

factory-boy example with AWS S3 storage

```
from factory import Faker
from faker_file.providers.pdf_file import PdfFileProvider
from faker_file.storages.aws_s3 import AWSS3Storage

STORAGE = AWSS3Storage(
    bucket_name=settings.AWS_STORAGE_BUCKET_NAME,
    root_path="",
    rel_path="",
)

Faker.add_provider(PdfFileProvider)


class UploadFactory(DjangoModelFactory):
    name = Faker("word")
    description = Faker("text")
    file = Faker("pdf_file", storage=STORAGE)

    class Meta:
        model = Upload


# Usage example
upload = UploadFactory()
```

See the full example here

Flexible storage selection

```
from django.core.files.storage import default_storage
from faker_file.storages.aws_s3 import AWSS3Storage
from faker_file.storages.filesystem import FileSystemStorage
from storages.backends.s3boto3 import S3Boto3Storage

# Faker doesn't know anything about Django. That's why, if we want to
# support remote storages, we need to manually check which file storage
# backend is used. If `Boto3` storage backend (of the `django-storages`
# package) is used we use the correspondent `AWSS3Storage` class of the
# `faker-file`.
# Otherwise, fall back to native file system storage (`FileSystemStorage`)
# of the `faker-file`.
if isinstance(default_storage, S3Boto3Storage):
    STORAGE = AWSS3Storage(
        bucket_name=settings.AWS_STORAGE_BUCKET_NAME,
        credentials={
            "key_id": settings.AWS_ACCESS_KEY_ID,
            "key_secret": settings.AWS_SECRET_ACCESS_KEY,
        },
        root_path="",
        rel_path="tmp",
    )
else:
    STORAGE = FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    )
```

See the full example here

15.4 Creating images

Creating images could be a challenging task. System dependencies on one side, large variety of many image formats on another.

Underlying creation of image files has been delegated to an abstraction layer of image generators. If you don't like how image files are generated or format you need isn't supported, you can create your own layer, using your favourite library.

Generally speaking, in faker-file each file provider represents a certain file type (with only a few exceptions). For generating a file in PNG format you should use *PngFileProvider*. For JPEG you would use *JpegFileProvider*.

15.4.1 Image providers

Currently, there are 3 types of image providers implemented:

- Graphic-only image providers.
- Mixed-content image providers.
- Image augmentation providers.

The graphic-only image providers are only capable of producing random graphics.

The mixed-content image providers can produce an image consisting of both text and graphics. Moreover, text comes in variety of different headings (such as h1, h2, h3, etc), paragraphs and tables.

Image augmentation providers simply augment existing images in a various, declaratively random, ways, such as: flip, resize, lighten, darken, grayscale and others.

15.4.2 Image generators

The following image generators are available.

- `PilImageGenerator`, built on top of the Pillow. It's the generator that will likely won't ask for any system dependencies that you don't yet have installed.
- `ImgkitImageGenerator` (default), built on top of the imgkit and wkhtmltopdf. Extremely easy to work with. Supports many formats.
- `WeasyPrintImageGenerator`, built on top of the WeasyPrint. Easy to work with. Supports formats that imgkit does not.

15.4.3 Building mixed-content images using imgkit

While imgkit generator is heavier and has wkhtmltopdf as a system dependency, it produces better quality images and has no issues with fonts or unicode characters.

See the following full functional snippet for generating images using imgkit.

```
from faker import Faker
from faker_file.providers.image.imgkit_generator import ImgkitImageGenerator
from faker_file.providers.png_file import PngFileProvider

FAKER = Faker()  # Initialize Faker
FAKER.add_provider(PngFileProvider)  # Register PngFileProvider

# Generate PNG file using `imgkit`
pdf_file = FAKER.png_file(image_generator_cls=ImgkitImageGenerator)
```

See the full example `here`

The generated PNG image will have 10,000 characters of text. The generated image will be as wide as needed to fit those 10,000 characters, but newlines are respected.

If you want image to be less wide, set value of `wrap_chars_after` to 80 characters (or any other number that fits your needs). See the example below:

```
# Generate an image file, wrapping each line after 80 characters
png_file = FAKER.png_file(
    image_generator_cls=ImgkitImageGenerator, wrap_chars_after=80
)
```

See the full example `here`

To have a longer text, increase the value of `max_nb_chars` accordingly. See the example below:

```
# Generate an image file of 20,000 characters
png_file = FAKER.png_file(
    image_generator_cls=ImgkitImageGenerator, max_nb_chars=20_000
)
```

See the full example here

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables and pretty much everything that you could think of, although currently only images, paragraphs and tables are supported out of the box. In order to customise the blocks image file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.image.imgkit_snippets import (
    add_paragraph,
    add_picture,
    add_table,
)

# Create an image file with a paragraph, a picture and a table.
# The ``DynamicTemplate`` simply accepts a list of callables (such
# as ``add_paragraph``, ``add_picture``) and dictionary to be later on
# fed to the callables as keyword arguments for customising the default
# values.
png_file = FAKER.png_file(
    image_generator_cls=ImgkitImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
png_file = FAKER.png_file(
    image_generator_cls=ImgkitImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.4.4 Building mixed-content images using WeasyPrint

While WeasyPrint generator isn't better or faster than the imgkit, it supports formats that imgkit doesn't (and vice-versa) and therefore is a good alternative to.

See the following snippet for generating images using WeasyPrint.

```
from faker_file.providers.image.weasyprint_generator import (
    WeasyPrintImageGenerator,
)

# Generate image file using `WeasyPrint`
png_file = FAKER.png_file(image_generator_cls=WeasyPrintImageGenerator)
```

See the full example here

All examples shown for imgkit apply for WeasyPrint generator, however when building images files from blocks (paragraphs, images and tables), the imports shall be adjusted:

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables and pretty much everything else that you could think of, although currently only images, paragraphs and tables are supported. In order to customise the blocks image file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.contrib.image.weasyprint_snippets import (
    add_paragraph,
    add_picture,
    add_table,
)

# Create an image file with paragraph, picture and table.
# The ``DynamicTemplate`` simply accepts a list of callables (such
# as ``add_paragraph``, ``add_picture``) and dictionary to be later on
# fed to the callables as keyword arguments for customising the default
# values.
png_file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
png_file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
```

(continues on next page)

(continued from previous page)

```
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.4.5 Building mixed-content images using Pillow

Usage example:

```
from faker_file.providers.image.pil_generator import PilImageGenerator

png_file = FAKER.png_file(image_generator_cls=PilImageGenerator)
```

See the full example here

With options:

```
png_file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    image_generator_kwargs={
        "encoding": "utf8",
        "font_size": 14,
        "page_width": 800,
        "page_height": 1200,
        "line_height": 16,
        "spacing": 5,
    },
    wrap_chars_after=100,
)
```

See the full example here

All examples shown for imgkit and WeasyPrint apply to Pillow generator, however when building image files from blocks (paragraphs, images and tables), the imports shall be adjusted. See the example below:

```
from faker_file.contrib.image.pil_snippets import (
    add_paragraph,
    add_picture,
    add_table,
)

# Create an image file with paragraph, picture and table.
# The ``DynamicTemplate`` simply accepts a list of callables (such as
# ``add_paragraph``, ``add_picture``) and dictionary to be later on fed
# to the callables as keyword arguments for customising the default
```

(continues on next page)

(continued from previous page)

```
# values.
png_file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
png_file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_picture, {}),  # Add picture
            (add_table, {}),  # Add table
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.4.6 Creating graphics-only images using Pillow

There are so called `graphic` image file providers available. Produced image files would not contain text, so don't use it when you need text based content. However, sometimes you just need a valid image file, without caring much about the content. That's where graphic image providers comes to rescue:

```
from faker_file.providers.png_file import GraphicPngFileProvider

FAKER.add_provider(GraphicPngFileProvider)  # Register provider

png_file = FAKER.graphic_png_file()
```

See the full example here

The generated file will contain a random graphic (consisting of lines and shapes of different colours).

One of the most useful arguments supported is `size`.

```
png_file = FAKER.graphic_png_file(size=(800, 800))
```

See the full example here

15.4.7 Augment existing images

Augment the input image with a series of random augmentation methods.

```
from faker_file.providers.augment_image_from_path import (
    AugmentImageFromPathProvider,
)
from faker_file.providers.augment_random_image_from_dir import (
    AugmentRandomImageFromDirProvider,
)
from faker_file.providers.image.augment import (
    add_brightness,
    decrease_contrast,
    flip_horizontal,
    flip_vertical,
    resize_height,
    resize_width,
)

FAKER.add_provider(AugmentImageFromPathProvider)
FAKER.add_provider(AugmentRandomImageFromDirProvider)

# We assumed that directory "/tmp/tmp/" exists and contains
# image files, among which "01.png". Augmentations will be applied
# sequentially, one by one until all fulfilled. If you wish to apply only
# a random number of augmentations, but not all, pass the `num_steps`
# argument, with value less than the number of `augmentations` provided.
augmented_image_file = FAKER.augment_image_from_path(
    path="/tmp/tmp/01.png",
    augmentations=[
        (flip_horizontal, {}),
        (flip_vertical, {}),
        (decrease_contrast, {}),
        (add_brightness, {}),
        (resize_width, {"lower": 0.9, "upper": 1.1}),
        (resize_height, {"lower": 0.9, "upper": 1.1}),
    ],
    prefix="augmented_image_01_",
    # num_steps=3,
)

augmented_random_image_file = FAKER.augment_random_image_from_dir(
    source_dir_path="/tmp/tmp/",
    augmentations=[
        (flip_horizontal, {}),
        (flip_vertical, {}),
        (decrease_contrast, {}),
        (add_brightness, {}),
        (resize_width, {"lower": 0.9, "upper": 1.1}),
        (resize_height, {"lower": 0.9, "upper": 1.1}),
    ],
    prefix="augmented_random_image_",
    # num_steps=3,
)
```

See the full example `here`

15.5 Creating PDF

PDF is certainly one of the most complicated formats out there. And certainly one of the formats most of the developers will be having trouble with, as there are many versions and dialects. That makes it almost impossible and highly challenging to have **just one right way** of creating PDF files. That's why, creation of PDF files has been delegated to an abstraction layer of PDF generators. If you don't like how PDF files are generated, you can create your own layer, using your favourite library.

Currently, there are three PDF generators implemented:

- `PdfkitPdfGenerator` (default), built on top of the pdfkit and wkhtmltopdf.
- `ReportlabPdfGenerator`, build on top of the famous reportlab.
- `PilPdfGenerator`, build on top of the Pillow. Produced PDFs would contain images only (even texts are stored as images), unlike pdfkit or reportlab based solutions, where PDFs would simply contain selectable text. However, it's the generator that will likely won't ask for any system dependencies that you don't yet have installed.

15.5.1 Building PDF with text using pdfkit

While pdfkit generator is heavier and has wkhtmltopdf as a system dependency, it produces better quality PDFs and has no issues with fonts or unicode characters.

See the following full functional snippet for generating PDF using pdfkit.

```
# Required imports
from faker import Faker
from faker_file.providers.pdf_file import PdfFileProvider
from faker_file.providers.pdf_file.generators.pdfkit_generator import (
    PdfkitPdfGenerator,
)

FAKER = Faker()  # Initialize Faker
FAKER.add_provider(PdfFileProvider)  # Register PdfFileProvider

# Generate PDF file using `pdfkit`
pdf_file = FAKER.pdf_file(pdf_generator_cls=PdfkitPdfGenerator)
```

See the full example `here`

The generated PDF will have 10,000 characters of text, which is about 2 pages.

If you want PDF with more pages, you could either:

- Increase the value of `max_nb_chars` accordingly.
- Set value of `wrap_chars_after` to 80 characters to force longer pages.
- Insert manual page breaks and other content.

See the example below for `max_nb_chars` tweak:

```
# Generate PDF file of 20,000 characters, using `pdfkit`
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PdfkitPdfGenerator, max_nb_chars=20_000
)
```

See the full example here

See the example below for `wrap_chars_after` tweak:

```
# Generate PDF file, wrapping each line after 80 characters, using `pdfkit`
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PdfkitPdfGenerator, wrap_chars_after=80
)
```

See the full example here

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables, manual text break and pretty much everything that is supported by PDF format specification, although currently only images, paragraphs, tables and manual text breaks are supported out of the box. In order to customise the blocks PDF file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.pdf_file.pdfkit_snippets import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a PDF file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PdfkitPdfGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
pdf_file = FAKER.pdf_file(
```

(continues on next page)

(continued from previous page)

```
    pdf_generator_cls=PdfkitPdfGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.5.2 Building PDFs with text using reportlab

While reportlab generator is much lighter than the pdfkit and does not have system dependencies, but might produce PDF files with questionable encoding when generating unicode text.

See the following full functional snippet for generating PDF using reportlab.

```
from faker_file.providers.pdf_file.generators.reportlab_generator import (
    ReportlabPdfGenerator,
)

# Generate PDF file using `reportlab`
pdf_file = FAKER.pdf_file(pdf_generator_cls=ReportlabPdfGenerator)
```

See the full example here

All examples shown for pdfkit apply for reportlab generator, however when building PDF files from blocks (paragraphs, images, tables and page breaks), the imports shall be adjusted.

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables, manual text break and pretty much everything that is supported by PDF format specification, although currently only images, paragraphs, tables and manual text breaks are supported. In order to customise the blocks PDF file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.contrib.pdf_file.reportlab_snippets import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a PDF file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
```

(continues on next page)

(continued from previous page)

```
# as keyword arguments for customising the default values.
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=ReportlabPdfGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=ReportlabPdfGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.5.3 Building PDFs with text using Pillow

Usage example:

```
from faker_file.providers.pdf_file.generators.pil_generator import (
    PilPdfGenerator,
)

pdf_file = FAKER.pdf_file(pdf_generator_cls=PilPdfGenerator)
```

See the full example here

With options:

```
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PilPdfGenerator,
```

(continues on next page)

(continued from previous page)

```
    pdf_generator_kwargs={
        "encoding": "utf8",
        "font_size": 14,
        "page_width": 800,
        "page_height": 1200,
        "line_height": 16,
        "spacing": 5,
    },
    wrap_chars_after=100,
)
```

See the full example here

All examples shown for pdfkit and reportlab apply to Pillow generator, however when building PDF files from blocks (paragraphs, images, tables and page breaks), the imports shall be adjusted.

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables, manual text break and pretty much everything that is supported by PDF format specification, although currently only images, paragraphs, tables and manual text breaks are supported. In order to customise the blocks PDF file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.contrib.pdf_file.pil_snippets import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a PDF file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PilPdfGenerator,
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    ),
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
pdf_file = FAKER.pdf_file(
    pdf_generator_cls=PilPdfGenerator,
    content=DynamicTemplate(
```

(continues on next page)

(continued from previous page)

```
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
        * 5  # Will repeat your config 5 times
    ),
)
```

See the full example here

15.5.4 Creating PDFs with graphics using Pillow

There's a so called *graphic* PDF file provider available. Produced PDF files would not contain text, so don't use it when you need text based content. However, sometimes you just need a valid file in PDF format, without caring much about the content. That's where a GraphicPdfFileProvider comes to rescue:

```
from faker_file.providers.pdf_file import GraphicPdfFileProvider

pdf_file = FAKER.graphic_pdf_file()
```

See the full example here

The generated file will contain a random graphic (consisting of lines and shapes of different colours).

One of the most useful arguments supported is `size`.

```
pdf_file = FAKER.graphic_pdf_file(size=(800, 800))
```

See the full example here

15.6 Creating DOCX

See the following full functional snippet for generating DOCX.

```
# Required imports
from faker import Faker
from faker_file.providers.docx_file import DocxFileProvider

FAKER = Faker()  # Initialize Faker
FAKER.add_provider(DocxFileProvider)  # Register DocxFileProvider

# Generate DOCX file
docx_file = FAKER.docx_file()
```

See the full example here

The generated DOCX will have 10,000 characters of text, which is about 5 pages.

If you want DOCX with more pages, you could either:

- Increase the value of `max_nb_chars` accordingly.
- Set value of `wrap_chars_after` to 80 characters to force longer pages.
- Insert manual page breaks and other content.

See the example below for `max_nb_chars` tweak:

```
# Generate DOCX file of 20,000 characters
docx_file = FAKER.docx_file(max_nb_chars=20_000)
```

See the full example `here`

See the example below for `wrap_chars_after` tweak:

```
# Generate DOCX file, wrapping each line after 80 characters
docx_file = FAKER.docx_file(wrap_chars_after=80)
```

See the full example `here`

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables, manual text break and pretty much everything that is supported by DOCX format specification, although currently only images, paragraphs, tables and manual text breaks are supported out of the box. In order to customise the blocks DOCX file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.docx_file import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a DOCX file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
docx_file = FAKER.docx_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    )
)
```

(continues on next page)

(continued from previous page)

```
# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
docx_file = FAKER.docx_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
        * 5  # Will repeat your config 5 times
    )
)
```

See the full example here

15.7 Creating ODT

See the following full functional snippet for generating ODT.

```
# Required imports
from faker import Faker
from faker_file.providers.odt_file import OdtFileProvider

FAKER = Faker()  # Initialize Faker
FAKER.add_provider(OdtFileProvider)  # Register OdtFileProvider

# Generate ODT file
odt_file = FAKER.odt_file()
```

See the full example here

The generated ODT will have 10,000 characters of text, which is about 5 pages.

If you want ODT with more pages, you could either:

- Increase the value of `max_nb_chars` accordingly.
- Set value of `wrap_chars_after` to 80 characters to force longer pages.
- Insert manual page breaks and other content.

See the example below for `max_nb_chars` tweak:

```
# Generate ODT file of 20,000 characters
odt_file = FAKER.odt_file(max_nb_chars=20_000)
```

See the full example here

See the example below for `wrap_chars_after` tweak:

```
# Generate ODT file, wrapping each line after 80 characters
odt_file = FAKER.odt_file(wrap_chars_after=80)
```

See the full example here

As mentioned above, it's possible to diversify the generated context with images, paragraphs, tables, manual text break and pretty much everything that is supported by ODT format specification, although currently only images, paragraphs, tables and manual text breaks are supported out of the box. In order to customise the blocks ODT file is built from, the `DynamicTemplate` class is used. See the example below for usage examples:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.odt_file import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a ODT file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
odt_file = FAKER.odt_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    )
)

# You could make the list as long as you like or simply multiply for
# easier repetition as follows:
odt_file = FAKER.odt_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
        * 5  # Will repeat your config 5 times
    )
)
```

See the full example here

15.8 CLI

It's possible to generate files from CLI.

Note: For using CLI you should install all common dependencies (including underlying system dependencies).

Install using pipx (recommended):

```
pipx install faker-file[common]
```

Install using `pip`.

```
pip install faker-file[common] --user
```

15.8.1 List available provider options

```
faker-file --help
```

Output:

```
usage: faker-file [-h]
  {generate-completion,version,bin_file,csv_file,docx_file,eml_file,
   epub_file,generic_file,graphic_ico_file,graphic_jpeg_file,
   graphic_pdf_file,graphic_png_file,graphic_webp_file,ico_file,
   jpeg_file,mp3_file,odp_file,ods_file,odt_file,pdf_file,png_file,
   pptx_file,rtf_file,svg_file,tar_file,txt_file,webp_file,xlsx_file,
   xml_file,zip_file}
      ...

CLI for the faker-file package.

positional arguments:
  {generate-completion,version,bin_file,csv_file,docx_file,eml_file,
   epub_file,generic_file,graphic_ico_file,graphic_jpeg_file,
   graphic_pdf_file,graphic_png_file,graphic_webp_file,ico_file,
   jpeg_file,mp3_file,odp_file,ods_file,odt_file,pdf_file,png_file,
   pptx_file,rtf_file,svg_file,tar_file,txt_file,webp_file,xlsx_file,
   xml_file,zip_file}

                        Available file providers.
    generate-completion Generate bash completion file.
    version             Print version.
    bin_file            Generate a bin file.
    csv_file            Generate a csv file.
    docx_file           Generate a docx file.
    eml_file            Generate a eml file.
    epub_file           Generate a epub file.
    generic_file        Generate a generic file.
    graphic_ico_file    Generate a graphic_ico file.
    graphic_jpeg_file   Generate a graphic_jpeg file.
```

(continues on next page)

(continued from previous page)

```
    graphic_pdf_file        Generate a graphic_pdf file.
    graphic_png_file        Generate a graphic_png file.
    graphic_webp_file       Generate a graphic_webp file.
    ico_file                Generate a ico file.
    jpeg_file               Generate a jpeg file.
    mp3_file                Generate a mp3 file.
    odp_file                Generate a odp file.
    ods_file                Generate a ods file.
    odt_file                Generate a odt file.
    pdf_file                Generate a pdf file.
    png_file                Generate a png file.
    pptx_file               Generate a pptx file.
    rtf_file                Generate a rtf file.
    svg_file                Generate a svg file.
    tar_file                Generate a tar file.
    txt_file                Generate a txt file.
    webp_file               Generate a webp file.
    xlsx_file               Generate a xlsx file.
    xml_file                Generate a xml file.
    zip_file                Generate a zip file.

options:
  -h, --help            show this help message and exit
```

15.8.2 List options for a certain provider

```
faker-file docx_file --help
```

Output:

```
usage: faker-file docx_file [-h] [--prefix PREFIX] [--max_nb_chars MAX_NB_CHARS] [--wrap_
↪chars_after WRAP_CHARS_AFTER] [--content CONTENT] [--nb_files NB_FILES]

options:
  -h, --help            show this help message and exit
  --prefix PREFIX       prefix (default: None)
  --max_nb_chars MAX_NB_CHARS
                        max_nb_chars (default: 10000)
  --wrap_chars_after WRAP_CHARS_AFTER
                        wrap_chars_after (default: None)
  --content CONTENT     content (default: None)
  --nb_files NB_FILES   number of files to generate (default: 1)
```

15.8.3 Generate a file using certain provider

```
faker-file docx_file
```

Output:

```
Generated docx_file file: tmp/tmpva0mp3lp.docx
```

15.8.4 Shell auto-completion

First, generate shell auto-completion file.

```
faker-file generate-completion
```

Then, source the generated file:

```
source ~/faker_file_completion.sh
```

Now you can use auto-completion. Simply type faker-file [tab-tab] to see the list of available options:

```
$ faker-file
bin_file             graphic_jpeg_file    ods_file             txt_file
csv_file             graphic_pdf_file     odt_file             version
docx_file            graphic_png_file     pdf_file             webp_file
eml_file             graphic_webp_file    png_file             xlsx_file
epub_file            ico_file             pptx_file            xml_file
generate-completion  jpeg_file            rtf_file             zip_file
generic_file         mp3_file             svg_file
graphic_ico_file     odp_file             tar_file
```

It works with sub options too:

```
$ faker-file docx_file --
--content    --max_nb_chars    --prefix    --wrap_chars_after    --nb_files
```

To update the completion script, simply run the `generate-completion` command again and source the `~/faker_file_completion.sh` as already shown above.

15.9 Methodology

15.9.1 But why

Let's start with some hypothetical questions.

"But why generate testing files dynamically", - you may ask?

And the answer would be, - "for a number of reasons":

Because you do need files and managing test files is a pain nobody wants to have. You create testing files for one use case, then you need to support another, but you need to modify the original files or make modifications there. You either duplicate or make changes, then at some point, after a number of iterations, your test files collection grows so big, you can't easily find out how some of the test files different one from another or your test fail, you spend some time to investigate and find out that there has been a slight modification of one of the files, which made your pipeline to fail.

You fix the error and decide to document your collection (a good thing anyway). But then your collection grows even more. The burden of managing both test files, the documentation of the test files and the test code becomes unbearable.

Now imagine doing it not for one, but for a number of projects. You want to be smart and make a collection of files, document it properly and think you've done a good job, but then you start to realise that you do need to deviate or add new files to the collection to support new use cases. You want to be safe and decide to version control it. Your collection grows, you start ot accept PRs from other devs and go down the rabbit whole of owning another critical repository. Your documentation grows and so does the repository size (mostly binary content). Storing such a huge amount of files becomes a burden. It slows down everyone.

Not even talking about, that you might not be allowed to store some of the you're using for testing centrally, because you would then need to run obfuscation, anonymization to legally address concerns of privacy regulations.

15.9.2 When test files are generated dynamically

When test files are generated dynamically, you are relieved from most of the concerns mentioned above. There are a couple of drawbacks here too, such as tests execution time (because generating of the test files on the fly does require some computation resources and therefore - your CI execution time will grow).

15.9.3 Best practices

In some very specific use-cases, mimicking original files might be too difficult and you might want to still consider including some of the very specific and hard-to-recreate files in the project repository, but on much lower scale. Use faker-file for simple use cases and only use custom files when things get too complicated otherwise. The so-called hybrid approach.

15.9.3.1 Identify what kind of files do you need

faker-file supports a large variety of file types, but content of files can be generally broken down by 2 categories:

- Text based: Useful when testing OCR or text processing pipelines. ATM, most of the faker-file providers generate text-based content.
- Non-text based: Typically images and non-human readable formats such as BIN. Useful when you need to test validity of the uploaded file, but don't care much about what's inside.

Image providers:

File type	Graphic	Text	Generator
BMP	GraphicBmpFileProvider	BmpFileProvider	Pillow, WeasyPrint
GIF	GraphicGifFileProvider	GifFileProvider	Pillow, WeasyPrint
ICO	GraphicIcoFileProvider	IcoFileProvider	Pillow, Imagekit, WeasyPrint
JPEG	GraphicJpegFileProvider	JpegFileProvider	Pillow, Imagekit, WeasyPrint
PDF	GraphicPdfFileProvider	PdfFileProvider	Pillow, Imagekit, WeasyPrint
PNG	GraphicPngFileProvider	PngFileProvider	Pillow, Imagekit, WeasyPrint
SVG	(not supported)	SvgFileProvider	Imagekit
TIFF	GraphicTiffFileProvider	TiffFileProvider	Pillow, Imagekit*, WeasyPrint
WEBP	GraphicWebpFileProvider	WebpFileProvider	Pillow, Imagekit*, WeasyPrint

Note: Items marked with * may require xvfb to function properly.

At the moment, 2 of the 3 text-to-image providers require additional system dependencies (such as wkhtmltopdf for imgkit and poppler for WeasyPrint, both of which are available for most popular operating systems, including Windows, macOS and Linux).

A few formats, such as BMP, GIF and TIFF, which are not supported by imgkit and underlying wkhtmltopdf, rely on WeasyPrint, pdf2image and poppler through the `WeasyPrintImageGenerator`.

The lightest alternative to imgkit and WeasyPrint generators is the Pillow generator (`PilImageGenerator`), which is basic, but does not require additional system dependencies to be installed (most of the system dependencies for Pillow are likely already installed on your system: `libjpeg`, `zlib`, `libtiff`, `libfreetype6` and `libwebp`).

Graphic image providers on the other hand rely on Pillow and underlying system dependencies mentioned above.

Take a good look at the prerequisites to identify required dependencies.

TL;DR

For text-to-image file generation you could use Pillow based generators, which are basic, but do not require additional system dependencies. For advanced text-to-image file generation you could use either imgkit or WeasyPrint based generators, which require wkhtmltopdf and poppler respectively.

For graphic file generation, the only option is to use graphic file providers, which depend on *Pillow* (and underlying system dependencies) only.

15.9.3.2 Installation

When using faker-file for automated tests in a large project with a lot of dependencies, the recommended way to install it is to carefully pick the dependencies required and further use requirements management package, like pip-tools, to compile them into hashed set of packages working well together.

For instance, if we only need DOCX and PDF support, your `requirements.in` file could look as follows:

```
faker
faker-file
python-docx
reportlab
```

If you only plan to use faker-file as a CLI application, just install all common dependencies as follows:

```
pipx install "faker-file[common]"
```

15.9.3.3 Creating files

A couple of use-cases when faker-file can help you out:

15.9.3.3.1 Create a simple DOCX file

Let's imagine we need to generate a DOCX file with text 50 chars long (just for observability).

```
from faker import Faker
from faker_file.providers.docx_file import DocxFileProvider

FAKER = Faker()
FAKER.add_provider(DocxFileProvider)

```

(continues on next page)

(continued from previous page)

```
docx_file = FAKER.docx_file(max_nb_chars=50)
print(docx_file)  # Sample value: 'tmp/tmpgdctmfbp.docx'
print(docx_file.data["content"])  # Sample value: 'Learn where receive social.'
print(docx_file.data["filename"])  # Sample value: '/tmp/tmp/tmpgdctmfbp.docx'
```

See the full example here

15.9.3.3.2 Create a more structured DOCX file

Imagine, you need a letter sample. It contains

```
TEMPLATE = """
{{date}} {{city}}, {{country}}

Hello {{name}},

{{text}}

Address: {{address}}

Best regards,

{{name}}
{{address}}
{{phone_number}}
"""

docx_file = FAKER.docx_file(content=TEMPLATE)

print(docx_file)  # Sample value: 'tmp/tmpgdctmfbp.docx'
print(docx_file.data["content"])
# Sample value below:
#  2009-05-14 Pettyberg, Puerto Rico
#  Hello Lauren Williams,
#
#  Everyone bill I information. Put particularly note language support
#  green. Game free family probably case day vote.
#  Commercial especially game heart.
#
#  Address: 19017 Jennifer Drives
#  Jamesbury, MI 39121
#
#  Best regards,
#
#  Robin Jones
#  4650 Paul Extensions
#  Port Johnside, VI 78151
#  001-704-255-3093
```

See the full example here

15.9.3.3.3 Create even more structured DOCX file

Imagine, you need to generate a highly custom document with types of data, such as images, tables, manual page breaks, paragraphs, etc.

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.docx_file import (
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a DOCX file with paragraph, picture, table and manual page breaks
# in between the mentioned elements. The ``DynamicTemplate`` simply
# accepts a list of callables (such as ``add_paragraph``,
# ``add_page_break``) and dictionary to be later on fed to the callables
# as keyword arguments for customising the default values.
docx_file = FAKER.docx_file(
    content=DynamicTemplate(
        [
            (add_paragraph, {}),  # Add paragraph
            (add_page_break, {}),  # Add page break
            (add_picture, {}),  # Add picture
            (add_page_break, {}),  # Add page break
            (add_table, {}),  # Add table
            (add_page_break, {}),  # Add page break
        ]
    )
)
```

See the full example here

Note: All callables do accept arguments. You could provide `content=TEMPLATE` argument to the `add_paragraph` function and instead of just random text, you would get a more structured paragraph (from one of previous examples).

15.9.3.3.4 For when you think faker-file isn't enough

As previously mentioned, sometimes when test documents are too complex it might be hard to replicate them and you want to store just a few very specific documents in the project repository.

faker-file comes up with a couple of providers that might still help you in that case.

Both *FileFromPathProvider* and *RandomFileFromDirProvider* are created to support the hybrid approach.

15.9.3.3.4.1 FileFromPathProvider

Create a file by copying it from the given path.

- Create an exact copy of a file under a different name.
- Prefix of the destination file would be `zzz`.
- `path` is the absolute path to the file to copy.

```
from faker_file.providers.file_from_path import FileFromPathProvider

FAKER.add_provider(FileFromPathProvider)

# We assume that directory "/tmp/tmp/" exists and contains a file named
# "file.docx".
docx_file_copy = FAKER.file_from_path(
    path="/tmp/tmp/file.docx",
    prefix="zzz",
)
```

See the full example here

Now you don't have to copy-paste your file from one place to another. It will be done for you in a convenient way.

15.9.3.3.4.2 RandomFileFromDirProvider

Create a file by copying it randomly from the given directory.

- Create an exact copy of the randomly picked file under a different name.
- Prefix of the destination file would be `zzz`.
- `source_dir_path` is the absolute path to the directory to pick files from.

```
from faker_file.providers.random_file_from_dir import RandomFileFromDirProvider

FAKER.add_provider(RandomFileFromDirProvider)

# We assume that directory "/tmp/tmp/" exists and contains files with".docx"
# extension.
docx_file_copy = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
    prefix="zzz",
)
```

See the full example here

Now you don't have to copy-paste your file from one place to another. It will be done for you in a convenient way.

15.9.3.4 Clean up files

`FileSystemStorage` is the default storage and by default files are stored inside a `tmp` directory within the system's temporary directory, which is commonly cleaned up after system restart. However, there's a mechanism of cleaning up files after the tests run. At any time, to clean up all files created by that moment, call `clean_up` method of the `FileRegistry` class instance, as shown below:

```
# Import instance at once
from faker_file.registry import FILE_REGISTRY

# Trigger the clean-up
FILE_REGISTRY.clean_up()
```

See the full example `here`

Typically you would call the `clean_up` method in the `tearDown`.

To remove a single file, use `remove` method of `FileRegistry` instance.

```
# We assume that there's an initialized `txt_file` instance to remove.
FILE_REGISTRY.remove(txt_file)  # Where file is an instance of ``StringValue``
```

See the full example `here`

If you only have a string representation of the `StringValue`, try to search for its' correspondent `StringValue` instance first using `search` method.

```
# We assume that there's an initialized `filename` (str) to remove.
txt_file = FILE_REGISTRY.search(filename)
if txt_file:
    FILE_REGISTRY.remove(txt_file)
```

See the full example `here`

15.10 Testing files like a pro

Note: Talk from the PyGrunn conference in 2023.

Create files with fake data. In many formats. With no efforts.

15.10.1 Introduction

Thank you for choosing this talk and for being here.

There might be many reasons why you're here. Perhaps you haven't done any testing in Python that required files, so you're curious. Or maybe you have done it many times, but never really liked what you did because it was too verbose or intrusive.

Every time I had to deal with testing files, I had to invent things, reinvent things, recall things from the past, and each time, before diving into a rabbit hole of writing many lines of code or producing yet another collection of files stored

somewhere, I checked for available solutions that could simplify things for me, make it easier, less intrusive, and less work. I wanted to make it just fine and enjoyable to work with.

As of today, I have found a solution that works well for me, and that's what I want to share with you.

15.10.2 Why/motivation

But why, you may ask?

Because test files are often not available when you need them. At least, not at the right time for testing, because your customer or partner doesn't have them. And even if they do have the right files, there are dozens of reasons for never-ending delays, most of which are related to privacy regulations, such as NDAs to be signed, anonymization, and so on.

And yet, there are deadlines. You have to come up with something, every time. For every project you work on. For every file format you are expected to support.

Or maybe you do have a few test files, and you decide to test your pipeline with the 100 you have (if you're lucky to have that much) and it all works. Then you go live and discover that your system doesn't perform well enough to handle thousands of them.

But what are files really? Are they not just pieces of texts and images, sometimes tables, audios and videos, spreadsheets, presentations - all mostly originated from text. We can generate text!

Nowadays, we have concepts such as Synthetic Data and libraries like Faker to support these concepts.

15.10.3 Intermezzo

And if you have never heard of Faker or the term Synthetic Data, I'll make a quick recap for you.

Synthetic data, or fake data, is computer-generated data that is similar to real-world data. It's primary purpose is to increase the privacy and integrity of systems.

As everything else in life, it has pros, cons and alternatives.

15.10.3.1 The pros

- **Data privacy**: Because it's fake - there's no risk of exposing sensitive user data and no need to comply with data privacy regulations.
- **Scalability**: You can generate as much data as you need.
- **Controll**: You have full control over the data, so you can test specific rare edge cases.

15.10.3.2 The cons

- **Realism**: Because it's fake it does not always accurately represent real data or contain the same patterns and anomalies. That could lead to less accurate testing.
- **Generation complexity**: Creating realistic data can be complex and time-consuming, depending on the domain and the complexity of the data structures.
- **Maintenance**: Keeping the data generation logic up-to-date with evolving application requirements does take time.

15.10.3.3 The alternatives

- **Production data anonymization**: When you take a copy (or subset) of the real production data and anonymize it to remove or obfuscate sensitive information.
- **Manual test data creation**: When you manually create test data, usually done for smaller scale or more specific testing.
- **Data augmentation**: When you modify existing data to create new data.

All of the alternatives have their pros and cons too, but I'm not going to cover any of that in this presentation.

Faker is a Python package for generating synthetic text data. It's knows many patterns and locales. It can generate names, texts, addresses, zip codes, ISBN numbers and a lot more.

I started to use Faker around 2016. It was such a relief! You could just do things like this:

```
from faker import Faker
FAKER = Faker()
FAKER.first_name()
FAKER.last_name()
FAKER.address()
FAKER.zip_code()
FAKER.text()
FAKER.isbn13()
FAKER.email()
FAKER.company_email()
FAKER.company()
FAKER.date_between(start_date="-30y", end_date="+30y")
```

Before Faker there was *Lorem Ipsum* (or Lipsum), which was OK (or better than nothing), but didn't make much sense.

Then Faker (and *Faker*-like libraries for creating fake data) emerged to save us.

Then test cases became more complex. Primary data sources were often files. We needed to test data/ETL pipelines. Faker still helped a lot, but it was inconvenient to replicate your previous best approach for files and reinvent the wheel for each new project.

That's why faker-file was created. I wrote it mainly for myself, but you may find it useful too.

15.10.4 How does faker-file help to solve that problem?

In essence, faker-file is just a set of providers for the famous Faker library.

- You can use it with Faker and factory_boy (for ORM integration).
- It works with Django.
- It supports remote storages (AWS S3, Google Cloud Storage, Azure Cloud Storage).
- You are in control of the generated content. By default, for most basic cases, content it's generated using Faker's `text` method, but you could easily tweak that using the `content` argument.

You can use it to run a comprehensive integration test of your pipeline in your favorite cloud.

Some of the most commonly-used file formats are supported:

- *BIN*
- *CSV*
- *DOCX*

- *EML*
- *EPUB*
- *ICO*
- *JPEG*
- *MP3*
- *ODP*
- *ODS*
- *ODT*
- *PDF*
- *PNG*
- *RTF*
- *PPTX*
- *SVG*
- *TXT*
- *WEBP*
- *XLSX*
- *XML*
- *ZIP*

Installation

```
pip install faker-file[common]
```

Using it is as simple as follows.

15.10.4.1 Generate a *DOCX* file with fake content

- Generate 1 *DOCX* file with fake content (generated by Faker).

```
# Import the Faker class from faker package
from faker import Faker

# Import the file provider we want to use
from faker_file.providers.docx_file import DocxFileProvider

FAKER = Faker()  # Initialise Faker instance

FAKER.add_provider(DocxFileProvider)  # Register the DOCX file provider

file = FAKER.docx_file()  # Generate a DOCX file

# Note, that `file` is this case is an instance of either `StringValue`
# or `BytesValue` objects, which inherit from `str` and `bytes`
# respectively, but add meta data. Meta data is stored inside the `data`
# property (`Dict`). One of the common attributes of which (among all
```

(continues on next page)

(continued from previous page)

```
# file providers) is the `filename`, which holds an absolute path to the
# generated file.
print(file.data["filename"])

# Another common attribute (although it's not available for all providers)
# is `content`, which holds the text used to generate the file with.
print(file.data["content"])
```

15.10.4.2 Provide content manually

- Generate 1 *DOCX* file with developer defined content.

```
# The text we want have in our generated DOCX file
TEXT = """
"The Queen of Hearts, she made some tarts,
    All on a summer day:
The Knave of Hearts, he stole those tarts,
    And took them quite away."
"""

# Generate a DOCX file with the given text
file = FAKER.docx_file(content=TEXT)
```

- Similarly, generate 1 *PNG* file.

```
from faker_file.providers.png_file import PngFileProvider

FAKER.add_provider(PngFileProvider)

file = FAKER.png_file()
```

- Similarly, generate 1 *PDF* file. Limit the line width to 80 characters.

```
from faker_file.providers.pdf_file import PdfFileProvider

FAKER.add_provider(PdfFileProvider)

file = FAKER.pdf_file(wrap_chars_after=80)
```

15.10.4.3 Provide templated content

You can generate documents from pre-defined templates.

```
TEMPLATE = """
{{date}} {{city}}, {{country}}

Hello {{name}},

{{text}}
```

(continues on next page)

(continued from previous page)

```
Address: {{address}}

Best regards,

{{name}}
{{address}}
{{phone_number}}
"""

file = FAKER.pdf_file(content=TEMPLATE, wrap_chars_after=80)
```

15.10.4.4 Archive types

15.10.4.4.1 ZIP archive containing 5 TXT files

As you might have noticed, some archive types are also supported. The created archive will contain 5 files in TXT format (defaults).

```
from faker_file.providers.zip_file import ZipFileProvider

FAKER.add_provider(ZipFileProvider)

file = FAKER.zip_file()
```

15.10.4.4.2 ZIP archive containing 3 DOCX files with text generated from a template

```
from faker_file.providers.helpers.inner import create_inner_docx_file

file = FAKER.zip_file(
    prefix="zzz",
    options={
        "count": 3,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "xxx_",
            "content": TEMPLATE,
        },
        "directory": "yyy",
    }
)
```

15.10.4.4.3 Nested ZIP archive

And of course nested archives are supported too. Create a *ZIP* file which contains 5 *ZIP* files which contain 5 *ZIP* files which contain 2 *DOCX* files.

- 5 *ZIP* files in the *ZIP* archive.
- Content is generated dynamically.
- Prefix the filenames in archive with `nested_level_1_`.
- Prefix the filename of the archive itself with `nested_level_0_`.
- Each of the *ZIP* files inside the *ZIP* file in their turn contains 5 other *ZIP* files, prefixed with `nested_level_2_`, which in their turn contain 2 *DOCX* files.

```
from faker_file.providers.helpers.inner import create_inner_zip_file

file = FAKER.zip_file(
    prefix="nested_level_0_",
    options={
        "create_inner_file_func": create_inner_zip_file,
        "create_inner_file_args": {
            "prefix": "nested_level_1_",
            "options": {
                "create_inner_file_func": create_inner_zip_file,
                "create_inner_file_args": {
                    "prefix": "nested_level_2_",
                    "options": {
                        "count": 2,
                        "create_inner_file_func": create_inner_docx_file,
                        "create_inner_file_args": {
                            "content": TEXT + "\n\n{{date}}",
                        }
                    }
                },
            }
        },
    }
)
```

It works similarly for *EML* files (using `EmlFileProvider`).

```
from faker_file.providers.eml_file import EmlFileProvider
from faker_file.providers.helpers.inner import create_inner_docx_file

FAKER.add_provider(EmlFileProvider)

file = FAKER.eml_file(
    prefix="zzz",
    content=TEMPLATE,
    options={
        "count": 3,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "xxx_",
```

(continues on next page)

(continued from previous page)

```
            "content": TEXT + "\n\n{{date}}",
        },
    }
)
```

15.10.4.4.4 Create a ZIP file with variety of different file types within

- 50 files in the ZIP archive (limited to DOCX, EPUB and TXT types).
- Content is generated dynamically.
- Prefix the filename of the archive itself with *zzz_archive_*.
- Inside the ZIP, put all files in directory zzz.

```
from faker import Faker
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)
from faker_file.providers.zip_file import ZipFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FAKER = Faker()
STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}
file = ZipFileProvider(FAKER).zip_file(
    prefix="zzz_archive_",
    options={
        "count": 50,
        "create_inner_file_func": fuzzy_choice_create_inner_file,
        "create_inner_file_args": {
            "func_choices": [
                (create_inner_docx_file, kwargs),
                (create_inner_epub_file, kwargs),
                (create_inner_txt_file, kwargs),
            ],
        },
        "directory": "zzz",
    }
)
```

15.10.4.4.5 Another way to create a ZIP file with variety of different file types within

- 3 files in the ZIP archive (1 DOCX, and 2 XML types).
- Content is generated dynamically.
- Filename of the archive itself is *alice-looking-through-the-glass.zip*.
- Files inside the archive have fixed name (passed with basename argument).

```
from faker import Faker
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_xml_file,
    list_create_inner_file,
)
from faker_file.providers.zip_file import ZipFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FAKER = Faker()
STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}
file = ZipFileProvider(FAKER).zip_file(
    basename="alice-looking-through-the-glass",
    options={
        "create_inner_file_func": list_create_inner_file,
        "create_inner_file_args": {
            "func_list": [
                (create_inner_docx_file, {"basename": "doc"}),
                (create_inner_xml_file, {"basename": "doc_metadata"}),
                (create_inner_xml_file, {"basename": "doc_isbn"}),
            ],
        },
    }
)
```

15.10.4.5 Using raw=True features in tests

If you pass `raw=True` argument to any provider or inner function, instead of creating a file, you will get bytes back (or to be totally correct, bytes-like object `BytesValue`, which is basically bytes enriched with meta-data). You could then use the bytes content of the file to build a test payload as shown in the example test below:

```
import os
from io import BytesIO

from django.test import TestCase
from django.urls import reverse
from faker import Faker
from faker_file.providers.docx_file import DocxFileProvider
from rest_framework.status import HTTP_201_CREATED
from upload.models import Upload

FAKER = Faker()
```

(continues on next page)

(continued from previous page)

```
FAKER.add_provider(DocxFileProvider)

class UploadTestCase(TestCase):
    """Upload test case."""

    def test_create_docx_upload(self) -> None:
        """Test create an Upload."""
        url = reverse("api:upload-list")

        raw = FAKER.docx_file(raw=True)
        test_file = BytesIO(raw)
        test_file.name = os.path.basename(raw.data["filename"])

        payload = {
            "name": FAKER.word(),
            "description": FAKER.paragraph(),
            "file": test_file,
        }

        response = self.client.post(url, payload, format="json")

        # Test if request is handled properly (HTTP 201)
        self.assertEqual(response.status_code, HTTP_201_CREATED)

        test_upload = Upload.objects.get(id=response.data["id"])

        # Test if the name is properly recorded
        self.assertEqual(str(test_upload.name), payload["name"])

        # Test if file name recorded properly
        self.assertEqual(str(test_upload.file.name), test_file.name)
```

15.10.4.6 Create a HTML file predefined template

If you want to generate a file in a format that is not (yet) supported, you can try to use `GenericFileProvider`. In the following example, an HTML file is generated from a template.

```
from faker import Faker
from faker_file.providers.generic_file import GenericFileProvider

file = GenericFileProvider(Faker()).generic_file(
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
)
```

15.10.4.7 Storages

15.10.4.7.1 Example usage with *Django* (using local file system storage)

```
from django.conf import settings
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.filesystem import FileSystemStorage

STORAGE = FileSystemStorage(
    root_path=settings.MEDIA_ROOT,
    rel_path="tmp",
)

FAKER.add_provider(TxtFileProvider)

file = FAKER.txt_file(content=TEXT, storage=STORAGE)
```

15.10.4.7.2 Example usage with AWS S3 storage

```
from faker_file.storages.aws_s3 import AWSS3Storage

S3_STORAGE = AWSS3Storage(
    bucket_name="test-bucket",
    root_path="tmp",  # Optional
    rel_path="sub-tmp",  # Optional
    # Credentials are optional too. If your AWS credentials are properly
    # set in the ~/.aws/credentials, you don't need to send them
    # explicitly.
    # credentials={
    #     "key_id": "YOUR KEY ID",
    #     "key_secret": "YOUR KEY SECRET"
    # },
)

file = FAKER.txt_file(storage=S3_STORAGE)
```

15.10.4.8 Augment existing files

If you think Faker generated data doesn't make sense for you and you want your files to look like a collection of 100 files you already have, you could use augmentation features.

You will need additional requirements:

```
pip install faker-file[ml]
```

Usage example:

```
from faker_file.providers.augment_file_from_dir import (
    AugmentFileFromDirProvider,
)
```

(continues on next page)

(continued from previous page)

```
FAKER.add_provider(AugmentFileFromDirProvider)

file = FAKER.augment_file_from_dir(
    source_dir_path="/home/me/Documents/faker_file_source/",
    wrap_chars_after=120,
)
```

Generated file will resemble text of the original document, but will not be the same.

15.10.4.9 CLI

Even if you're not using automated testing, but still want to quickly generate a file with fake content, you could use faker-file:

```
faker-file generate-completion
source ~/faker_file_completion.sh
```

Generate an MP3 file:

```
faker-file mp3_file --prefix=my_file_
```

Generate 10 DOCX files:

```
faker-file docx_file --nb_files 10 --prefix=my_file_
```

15.10.5 Without faker-file

There are alternatives.

You could simply store a collection of test files somewhere. If you do so, make sure you "know" your collection. It should be obvious of how to use it. In other words - document it properly, alongside snippets to make most of it.

Then there comes a natural question - where to store? Should it be centrally hosted or per repository?

An obvious drawback of centrally hosted approach is that modifications become critical. A mistake may cause failure of your CI/CD pipeline. Also, you need to take care of the setup (for both CI/CD and development).

On the other hand, if you do it per project/repository basis, or even using a blue-print repository, you miss these direct contributions to the upstream.

BTW, consider storing your test files in GitLFS.

Besides, adding test files to the repository still feels a little bit strange to me. There's always a case when you need to have a variation and therefore you need to make another copy, sometimes a very long copy. And oh, refactoring and cleaning up becomes almost unmanageable.

Additionally, you could always go for a mixed approach, when some of the essentially needed files you still do store in the repository (and that can be project specific), while you still make use of the synthetic data for the cases when it's justified.

15.10.6 Recap/conclusion

- Most likely, combination of Faker, factory_boy and faker-file will do just fine for your MVP and even way beyond that (you have all in one: synthetic data + dynamic fixtures + generation of files). This approach also saves you from thinking about where to store your test data, and overall, makes your code more manageable and simplifies the development process.
- If you need to test files in your project, think upfront about the details, such as amount of test files you will need, where to store them, how to store them, etc.
- If some of your test cases are too specific to replicate with faker-file, consider using hybrid approach.

15.11 Security Policy

15.11.1 Reporting a Vulnerability

Do not report security issues on GitHub!

Please report security issues by emailing Artur Barseghyan <artur.barseghyan@gmail.com>.

15.11.2 Supported Versions

Make sure to use the latest version.

The two most recent `faker-file` release series receive security support.

For example, during the development cycle leading to the release of `faker-file` 0.17.x, support will be provided for `faker-file` 0.16.x.

Upon the release of `faker-file` 0.18.x, security support for `faker-file` 0.16.x will end.

Version	Supported
0.17.x	Yes
0.16.x	Yes
< 0.16	No

15.12 Contributor Covenant Code of Conduct

15.12.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

15.12.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

15.12.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

15.12.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

15.12.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at artur.barseghyan@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

15.12.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

15.12.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

15.12.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

15.12.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

15.12.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

15.12.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

15.13 Contributor guidelines

15.13.1 Developer prerequisites

15.13.1.1 pre-commit

Refer to pre-commit for installation instructions.

TL;DR:

```
pip install pipx --user  # Install pipx
pipx install pre-commit  # Install pre-commit
pre-commit install  # Install pre-commit hooks
```

Installing pre-commit will ensure you adhere to the project code quality standards.

15.13.2 Code standards

black, isort, ruff and doc8 will be automatically triggered by pre-commit. Still, if you want to run checks manually:

```
./scripts/black.sh
./scripts/doc8.sh
./scripts/isort.sh
./scripts/ruff.sh
```

15.13.3 Requirements

Requirements are compiled using pip-tools.

```
./scripts/compile_requirements.sh
```

15.13.4 Virtual environment

You are advised to work in virtual environment.

TL;DR:

```
python -m venv env
pip install -e .
pip install -r examples/requirements/django_3_2_and_flask.txt
```

15.13.5 Documentation

Check documentation.

15.13.6 Testing

Check testing.

If you introduce changes or fixes, make sure to test them locally using all supported environments. For that use tox.

```
tox
```

In any case, GitHub Actions will catch potential errors, but using tox speeds things up.

15.13.7 Pull requests

You can contribute to the project by making a pull request.

For example:

- To fix documentation typos.
- To improve documentation (for instance, to add new recipe or fix an existing recipe that doesn't seem to work).
- To introduce a new feature (for instance, add support for a non-supported file type).

Good to know:

- Test suite makes extensive use of parametrization. Make sure you have added your changes in the right place.

General list to go through:

- Does your change require documentation update?
- Does your change require update to tests?
- Did you test both Latin and Unicode characters?
- Does your change rely on third-party cloud based service? If so, please make sure it's added to tests that should be retried a couple of times. Example: `@pytest.mark.flaky(reruns=5)`.

When fixing bugs (in addition to the general list):

- Make sure to add regression tests.

When adding a new feature (in addition to the general list):

- Check the licenses of added dependencies carefully and make sure to list them in prerequisites.
- Make sure to update the documentation (check whether the installation, features, recipes and quick start require changes).

15.13.8 Questions

Questions can be asked on GitHub discussions.

15.13.9 Issues

For reporting a bug or filing a feature request use GitHub issues.

Do not report security issues on GitHub. Check the support section.

15.14 Release history and notes

Sequence based identifiers are used for versioning (schema follows below):

```
major.minor[.revision]
```

- It's always safe to upgrade within the same minor version (for example, from 0.3 to 0.3.4).
- Minor version changes might be backwards incompatible. Read the release notes carefully before upgrading (for example, when upgrading from 0.3.4 to 0.4).
- All backwards incompatible changes are mentioned in this document.

15.14.1 0.17.11

2023-11-20

- Minor documentation fixes.

15.14.2 0.17.10

2023-11-19

- Make `text_extractor_cls` and `text_augmenter_cls` arguments of the `AugmentFileFromDirProvider` provider access string values (to load requested class from path given).
- Add `TextaugmentAugmenter` based on textaugment package (very lightweight) and make it a default text augmenter.
- Minor documentation fixes.
- Optimized GitHub CI.

15.14.3 0.17.9

2023-10-10

- Improvements and fixes in the documentation.
- Announcing feature plans to change default PDF and Image generators to Pillow based ones, instead of wkhtmltopdf in version 0.18.

15.14.4 0.17.8

2023-09-21

Note: This release is dedicated to the victims of the war in Artsakh (Nagorno-Karabakh), a land now lost to its native inhabitants (Armenians). Following a grueling nine-month blockade, Azerbaijan initiated another military onslaught on September 19, 2023. The already weakened and outnumbered forces of Artsakh could no longer mount an effective resistance.

- Added support for `DynamicTemplate` to all non-graphic image providers. That means, that you can produce images with text, tables, various headings and other images. Correspondent snippets are implemented for all supported image generators; namely reportlab, WeasyPrint and Pillow.

15.14.5 0.17.7

2023-09-12

- Added `GTTS_MP3_GENERATOR` and `EDGE_TTS_MP3_GENERATOR` to the `mp3_file` provider import options.

15.14.6 0.17.6

2023-09-09

- Added `add_paragraph`, `add_picture`, `add_heading_h1` and other heading helpers to `pil_snippets` contrib module.

15.14.7 0.17.5

2023-08-22

Note: This release might introduces minor backwards incompatible changes only if you have written own- or customized existing- image providers and used them in combination with WeasyPrint-based image generator. A new property named `image_format` has been added to all image-based providers and the `WeasyPrintImageGenerator` is using that instead of formerly used `extension` property.

- Added `PilImageGenerator` (for text-to-image).
- Added `PilPdfGenerator` (for text-to-image).

15.14.8 0.17.4

2023-08-18

Note: Release is dedicated to the victims and de-facto hostages of the Blockade of the Republic of Artsakh. Have you ever heard of Armenian genocide? It's happening again. For more than 8 months, Azerbaijan has launched an illegal blockade of the Republic of Artsakh, including critical civilian infrastructure such as gas, electricity and roads connecting Armenia and Artaskh. Shortages of essential goods – including electricity, fuel, and water reserves – are widespread and emergency reserves are being rationed. The blockade has resulted in significant medical and food shortages in Artsakh, leading to increased health complications, as reported by Artsakh Healthcare ministry.

- Deaths due to cardiovascular diseases doubled in the first seven months of the year, with a particular surge in July-August.
- Deaths from malignant tumors rose by 15.9% over the same period due to lack of medications and medical aid.
- New cases of stroke and heart attacks increased by 26% and 9.7% respectively.
- Newly diagnosed cases of malignant tumors rose by 24.3%.
- Around 90% of monitored pregnant women developed anemia from poor nutrition and medication shortages.
- While overall abortion numbers remained stable, medically indicated abortions quadrupled in July due to factors like stress and inadequate nutrition.
- Reports of fainting surged by 91% in July-August.
- Emergency calls for high blood pressure saw a 5.6-fold increase in July-August.

The dire health outcomes are attributed to the blockade's impact, including medication shortages, stress, disrupted medical procedures, and restricted healthcare access. The Artsakh Health Ministry warns of further deterioration if the blockade continues, emphasizing the systemic challenges in healthcare delivery due to the blockade.

- Added `AugmentRandomImageFromDirProvider` and `AugmentImageFromPathProvider` providers for basic image augmentation.
- Added `storage` to metadata for all providers for easy clean-up of files.
- Added `unlink` method to all storages for easy clean-up of files.
- Added `FileRegistry` to keep track of all files created and introduce functionality for cleaning up the files.
- Stop testing against Python 3.7.

15.14.9 0.17.3

2023-08-02

Note: In memory of Sinead O'Connor.

- Allow to pass `image` argument (`bytes`) to the contrib `add_picture` functions.
- Documentation improvements.

15.14.10 0.17.2

2023-07-25

- Added `JSON` file provider.

15.14.11 0.17.1

2023-07-21

- Added `WeasyPrintImageGenerator` image generator class based on WeasyPrint and pdf2image packages.
- Added `BMP`, `TIFF` and `GIF` file providers (both text-to-image and graphic ones). Note, that above mentioned text-to-image providers are using `WeasyPrintImageGenerator` as a default image generator class, since `ImagekitImageGenerator` class isn't capable of supporting the above mentioned file formats.
- Added more helper functions for `DynamicTemplate` use for ODT, PDF and DOCX file providers to support h1, h2, h3, h4, h5 and h6 headings.

15.14.12 0.17

2023-07-12

Note: Release is dedicated to the victims and de-facto hostages of the Blockade of the Republic of Artsakh. Have you ever heard of Armenian genocide? It's happening again and the world silently watches.

- Introducing graphic image providers. Prior to this release, images have been created using text-to-image solutions. Sometimes it's just handy to have a graphic image. Therefore, a number of graphic image file providers have been created (including inner functions support). The following graphic file providers have been added: `GraphicIcoFileProvider`, `GraphicJpegFileProvider`, `GraphicPdfFileProvider`, `GraphicPngFileProvider` and `GraphicWebpFileProvider` to support creation of graphic `ICO`, `JPEG`, `PDF`, `PNG` and `WEBP` files.
- The previously mentioned text-to-image rendering has been delegated to image generators. Default generator is still based on the imgkit, but the change makes it possible to use custom generators.

15.14.13 0.16.4

2023-07-01

- Documentation improvements. Added a dedicated section for creating ODT files.
- Adding `add_paragraph` and `add_page_break` to `ODT` contrib module.

15.14.14 0.16.3

2023-06-30

- Documentation improvements. Added a dedicated section for creating PDF files. Added a dedicated section for creating DOCX files.
- Adding `add_paragraph` and `add_page_break` to `DOCX` contrib module.

15.14.15 0.16.2

2023-06-28

- Moving some of the snippets from tests to a `contrib` module to improve usability. The snippets are generic enough to be used in tests and if you don't like the way they work, you could always make a new one. New snippets to insert page breaks and paragraphs into PDF (using both pdfkit and reportlab generators) have been added.

15.14.16 0.16.1

2023-06-23

- Better error handling in CLI.

15.14.17 0.16

2023-06-21

Note: This release is dedicated to my beloved son - Tigran, who turned 11!

Note: This release introduces minor backwards incompatible changes.

- Minor improvements in PDF generation. If you have been using `DynamicTemplate` to generate complex PDFs, you are likely affected by the change. Make sure to at least add an additional argument named `generator` to the functions passed to the `DynamicTemplate` class. See the example below:

  Old:

  ```
  def add_pb(provider, story, data, counter, **kwargs):
  ```

  New:

  ```
  def add_pb(provider, generator, story, data, counter, **kwargs):
  ```

- Add code examples of how to generate a PDF with 100 pages with both `PdfkitPdfGenerator` and `ReportlabPdfGenerator` PDF generator classes.
- Add `version` CLI command.
- Add `generate-completion` and `version` commands to the CLI auto-completion.

15.14.18 0.15.5

2023-06-18

- Minor fixes and documentation improvements.

15.14.19 0.15.4

2023-06-15

- Improved `SFTPStorage` tests.
- Stop testing against Python 3.7.
- Stop testing against Django 4.0.

15.14.20 0.15.3

2023-06-14

- Add `SFTPStorage` and correspondent tests.

15.14.21 0.15.2

2023-06-08

- Add optional `subject` argument to the `EmlFileProvider`. Update tests accordingly.
- Add data integrity tests.

15.14.22 0.15.1

2023-06-06

- Added `FileFromPathProvider` provider, which simply picks a file from path given. Add correspondent `create_inner_file_from_path` inner function.

15.14.23 0.15

2023-06-05

- Added `format_func` argument to most of the providers. This allows to control which formatter function will be used as a default formatter. Previously it has been `faker.provider.Python.pystr_format`, which has been changed to `faker.provider.Python.parse`, since the latter is more convenient (as it does not transform characters like `?`, `!`, `#` into something else using `bothify` method). To revert this behaviour, make sure to pass a callable function `faker_file.base.pystr_format_func` in `format_func` argument to each correspondent provider or inner function.
- Added `create_inner_random_file_from_dir` inner function.
- Tested against Django 4.2.
- Stop testing against Django 2.2.

15.14.24 0.14.5

2023-05-11

- Minor fixes in `xml_file` provider.

15.14.25 0.14.4

2023-05-11

- Changed type of `data_columns` for `xml_file` provider from `Sequence[Tuple[str, str]]` to `Dict[str, str]`.
- In the `pdf_file` provider, changed default value of `pdf_generator_cls` from concrete `PdfkitPdfGenerator` value to its' string representation faker_file.providers.pdf_file.generators.pdfkit_generator.PdfkitPdfGenerator.
- In the `mp3_file` provider, changed default value of `mp3_generator_cls` from concrete `GttsMp3Generator` value to its' string representation faker_file.providers.mp3_file.generators.gtts_generator.GttsMp3Generator.

15.14.26 0.14.3

2023-05-10

- Minor fixes in the `GenericFileProvider`.

15.14.27 0.14.2

2023-05-09

- Add `create_inner_generic_file` inner function.
- Add `generic_file` support to CLI.

15.14.28 0.14.1

2023-05-08

- Add support for `list_create_inner_file`-like functions to the `EML` file provider. If you are using CLI and CLI-completion, make sure to re-generate the completion file.
- Add `GenericFileProvider` provider to support generic file types.

15.14.29 0.14

2023-05-07

Note: This release introduces minor backwards incompatible changes.

- A new argument `basename` has been added to all providers, inner functions and storage classes. If you have customized things or created your own providers, make sure to make appropriate changes in your code. See the source code for more implementation examples. If you are using CLI and CLI-completion, make sure to re-generate the completion file.

- A new inner function `list_create_inner_file` has been added, using which it's possible to create just a list of given files (ignoring `count` value) using given arguments. The amount of files is determined by the `func_list` (each pair `(Callable, kwargs)` corresponds to a single file. Both `ZipFileProvider` and `TarFileProvider` have been altered to reflect these changes.
- Added to support for `XML` files through `XmlFileProvider`.

15.14.30 0.13

2023-05-05

Note: This release introduces minor backwards incompatible changes.

- Display full path to the created file in the CLI.
- Added `DynamicTemplate` support for `PDF` file. The `generate` method of the `BasePdfGenerator` and classes derived from it, got two new arguments: `data` (`Dict[str, Any]`), and `provider` (`Union[Faker, Generator, Provider]`). If you have implemented custom generators for `PDF` (`pdf_file` provider), make sure to reflect mentioned changes in your code.

15.14.31 0.12.6

2023-05-02

- Added `DynamicTemplate` support for `DOCX` and `ODT` files.

15.14.32 0.12.5

2023-04-24

Note: In memory of the victims of the Armenian Genocide.

- Expose `mp3_generator_cls` and `pdf_generator_cls` CLI options for `mp3_file` and `pdf_file` respectively.
- Add `num_files` CLI option for all providers.

15.14.33 0.12.4

2023-04-22

- Make it possible to load classes from strings for passing as arguments to `mp3_file` and `pdf_file` providers.

15.14.34 0.12.3

2023-04-21

- Fixes in CLI options.

15.14.35 0.12.2

2023-04-20

- Fixes in CLI options.

15.14.36 0.12.1

2023-04-19

- Added CLI options.

15.14.37 0.12

2023-02-24

Note, that this release introduces breaking changes!

- Make it easy to use a different PDF library with `PdfFileProvider` by adding `pdf_generator_cls` and `pdf_generator_kwargs` optional arguments to the `pdf_file` method. Added `ReportlabPdfGenerator` class based on the famous `reportlab` library. Default is still `PdfkitPdfGenerator`. Since `encoding` was something specific for `pdfkit` library, it was moved from `pdf_file` method to `PdfkitPdfGenerator`, to which it can be passed in `pdf_generator_kwargs`. If you have passed the `encoding` argument explicitly, make sure to make correspondent changes. Note, that using the new `ReportlabPdfGenerator` class could speed-up PDF generation by about 40 times.

15.14.38 0.11.5

2023-02-20

- Fixes in typing of `CsvFileProvider`. `Tuple[str, str]` becomes `Tuple[str, ...]`.

15.14.39 0.11.4

2023-02-16

Note: Release dedicated to my dear valentine - Anahit.

- Added `filename` to `data` property of values returned by `Mp3FileProvider` provider (`StringValue`, `BytesValue`).

15.14.40 0.11.3

2023-02-10

- Moved several interface classes from one location to another. If you haven't implemented custom generators, this won't affect you. If you did, make sure to update your imports:
  - `BaseTextAugmenter` has been moved from `faker_file.providers.augment_file_from_dir.augmenters.base` to `faker_file.providers.base.text_augmenter`.
  - `BaseTextExtractor` has been moved from `faker_file.providers.augment_file_from_dir.extractors.base` to `faker_file.providers.base.text_extractor`.
  - `BaseMp3Generator` has been moved from `faker_file.providers.mp3_file.generators.base` to `faker_file.providers.base.mp3_generator`.

15.14.41 0.11.2

2023-02-07

- Add `filename` to `data` property of values returned by providers (`StringValue`, `BytesValue`).

15.14.42 0.11.1

2023-01-31

- Documentation improvements.
- MyPy fixes.

15.14.43 0.11

2023-01-25

- Allow returning binary contents of the file by providing the `raw=True` argument (`False` by default, works with all provider classes and inner functions). If you have subclassed or overriden provider classes or written custom inner functions, make sure to reflect the changes in your code.

15.14.44 0.10.12

2023-01-21

- Add `TarFileProvider` and `create_inner_tar_file` function.
- Add `OdpFileProvider` and `create_inner_odp_file` function.

15.14.45 0.10.11

2023-01-20

- Improve `EPUB` document layout.
- Improve `PDF` document layout.
- Minor documentation improvements.

15.14.46 0.10.10

2023-01-19

- Allow passing `model_name` and `action` arguments to the `ContextualWordEmbeddingsAugmenter`.
- Replace `bert-base-cased` with `bert-base-multilingual-cased` as a default model for `ContextualWordEmbeddingsAugmenter`.
- Improve `PPTX` document layout.
- Minor fixes in documentation.

15.14.47 0.10.9

2023-01-18

- Add an installation directive `[common]` to install everything except ML libraries.
- Added testing of UTF8 content.

15.14.48 0.10.8

2023-01-16

- Switch to PyPI releases of `gtts`.
- Stop testing against Django 3.0 and 3.1.
- Documentation improvements.
- Tests improvements.

15.14.49 0.10.7

2023-01-13

- Add `OdtFileProvider` and `create_inner_odt_file` function.
- Documentation improvements.
- Async related deprecation fixes in `EdgeTtsMp3Generator`.
- Optimize example factories.

15.14.50 0.10.6

2023-01-11

- Add `AugmentFileFromDirProvider` provider for making augmented copies of randomly picked files from given directory.
- Documentation improvements.
- Fixes in setup.

15.14.51 0.10.5

2023-01-09

- Add `fuzzy_choice_create_inner_file` inner function for easy diversion of files within archives (`ZIP`, `EML`).
- Documentation improvements.
- Add `MaryTTS` example (another MP3 generator for `Mp3FileProvider`).

15.14.52 0.10.4

2023-01-08

- Add missing `mp3_generator_kwargs` argument to the `create_inner_mp3_file` function.
- Clean-up.

15.14.53 0.10.3

2023-01-07

Improvements of the `Mp3FileProvider` module:

- Pass active generator to the `Mp3FileProvider` in the `generator` argument if `BaseMp3Generator` (and all implementations).
- Introduce `handle_kwargs` method in the `BaseMp3Generator` to handle arbitrary provider specific tuning.
- Add `EdgeTtsMp3Generator` MP3 generator.
- Add `mp3_generator_kwargs` argument to the `Mp3FileProvider.mp3_file` method.

15.14.54 0.10.2

2023-01-06

- Add `Mp3FileProvider`.
- Add `create_inner_mp3_file` inner function.

15.14.55 0.10.1

2023-01-05

- Fixes in `ZipFileProvider`.

15.14.56 0.10

2023-01-04

Note, that this release introduces breaking changes!

- Move all `create_inner_*_file` functions from `faker_file.providers.zip_file` to `faker_file.providers.helpers.inner` module. Adjust your imports accordingly.
- Add `EmlFileProvider`.
- Add `create_inner_eml_file` inner function.

15.14.57 0.9.3

2023-01-03

- Add `EpubFileProvider` provider.

15.14.58 0.9.2

2022-12-23

- Add `RrfFileProvider`.
- Added `SQLAlchemy` factory example.

15.14.59 0.9.1

2022-12-19

- Fixes in cloud storage.
- Documentation fixes.

15.14.60 0.9

2022-12-17

- Add optional `encoding` argument to `CsvFileProvider` and `PdfFileProvider` providers.
- Add `root_path` argument to cloud storages.
- Moved all image related code (`IcoFileProvider`, `JpegFileProvider`, `PngFileProvider`, `SvgFileProvider`, `WebpFileProvider`) to `ImageMixin`. Moved all tabular data related code (`OdsFileProvider`, `XlsxFileProvider`) to `TabularDataMixin`.
- Documentation improvements.

15.14.61 0.8

2022-12-16

Note, that this release introduces breaking changes!

- All file system based operations are moved to a separate abstraction layer of file storages. The following storages have been implemented: `FileSystemStorage`, `PathyFileSystemStorage`, `AWSS3Storage`, `GoogleCloudStorage` and `AzureStorage`. The `root_path` and `rel_path` params of the providers are deprecated in favour of storages. See the docs more usage examples.

15.14.62 0.7

2022-12-12

- Added `RandomFileFromDirProvider` which picks a random file from directory given.
- Improved docs.

15.14.63 0.6

2022-12-11

- Pass optional `generator` argument to inner functions of the `ZipFileProvider`.
- Added `create_inner_zip_file` inner function which allows to create nested ZIPs.
- Reached test coverage of 100%.

15.14.64 0.5

2022-12-10

Note, that this release introduces breaking changes!

- Added *ODS* file support.
- Switched to `tablib` for easy, non-variant support of various formats (*XLSX*, *ODS*).
- Silence `imgkit` logging output.
- *ZipFileProvider* allows to pass arbitrary arguments to inner functions. Put all your inner function arguments into a dictionary and pass it in *create_inner_file_args* key inside *options* argument. See the example below.

```
zip_file = ZipFileProvider(None).file(
    prefix="zzz_archive_",
    options={
        "count": 5,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "zzz_file_",
            "max_nb_chars": 1_024,
            "content": "{{date}}\r\n{{text}}\r\n{{name}}",
        },
        "directory": "zzz",
    }
)
```

15.14.65 0.4

2022-12-09

Note, that this release introduces breaking changes!

- Remove the concept of content generators (and the correspondent `content_generator` arguments in implemented providers). Instead, allow usage of dynamic fixtures in the provided `content` argument.
- Remove temporary files when creating ZIP archives.
- Various improvements and fixes in docs.

15.14.66 0.3

2022-12-08

- Add support for *BIN*, *CSV* and *XLSX* files.
- Better visual representation of generated images and PDFs.

15.14.67 0.2

2022-12-07

- Added support for *ICO*, *JPEG*, *PNG*, *SVG* and *WEBP* files.
- Documentation improvements.

15.14.68 0.1

2022-12-06

- Initial beta release.

15.15 Package

15.15.1 faker_file package

15.15.1.1 Subpackages

15.15.1.1.1 faker_file.cli package

15.15.1.1.1.1 Submodules

15.15.1.1.1.2 faker_file.cli.command module

faker_file.cli.command.**main**()

15.15.1.1.1.3 faker_file.cli.helpers module

faker_file.cli.helpers.**generate_completion_file**()

faker_file.cli.helpers.**generate_file**(*method_name: str*, ***kwargs*) → *StringValue*

faker_file.cli.helpers.**get_method_kwargs**(*cls: Type[*FileMixin*]*, *method_name: str*) → Tuple[Dict[str, Any], Dict[str, Any]]

faker_file.cli.helpers.**is_optional_type**(*t: Any*) → bool

15.15.1.1.1.4 Module contents

15.15.1.1.2 faker_file.contrib package

15.15.1.1.2.1 Subpackages

15.15.1.1.2.2 faker_file.contrib.pdf_file package

15.15.1.1.2.3 Submodules

15.15.1.1.2.4 faker_file.contrib.pdf_file.pdfkit_snippets module

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h1_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h1 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h2_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h2 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h3_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h3 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h4_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h4 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h5_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h5 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_h6_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h6 heading generation.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_heading**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for heading generation using pdfkit.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_page_break**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the page break insertion using pdfkit.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_paragraph**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for paragraph generation using pdfkit.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_picture**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the picture generation using pdfkit.

faker_file.contrib.pdf_file.pdfkit_snippets.**add_table**(*provider*, *generator*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the table generation using pdfkit.

15.15.1.1.2.5 faker_file.contrib.pdf_file.reportlab_snippets module

15.15.1.1.2.6 Module contents

15.15.1.1.2.7 Submodules

15.15.1.1.2.8 faker_file.contrib.docx_file module

faker_file.contrib.docx_file.**add_h1_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h1 heading generation.

faker_file.contrib.docx_file.**add_h2_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h2 heading generation.

faker_file.contrib.docx_file.**add_h3_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h3 heading generation.

faker_file.contrib.docx_file.**add_h4_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h4 heading generation.

faker_file.contrib.docx_file.**add_h5_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h5 heading generation.

faker_file.contrib.docx_file.**add_h6_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the h6 heading generation.

faker_file.contrib.docx_file.**add_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the heading generation.

faker_file.contrib.docx_file.**add_page_break**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for page break generation.

faker_file.contrib.docx_file.**add_paragraph**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the paragraph generation.

faker_file.contrib.docx_file.**add_picture**(*provider*, *document*, *data*, *counter*, ***kwargs*)

Callable responsible for the picture generation.

faker_file.contrib.docx_file.**add_table**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the table generation.

faker_file.contrib.docx_file.**add_title_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the title heading generation.

15.15.1.1.2.9 faker_file.contrib.odt_file module

faker_file.contrib.odt_file.**add_h1_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h1 heading generation.

faker_file.contrib.odt_file.**add_h2_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h2 heading generation.

faker_file.contrib.odt_file.**add_h3_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h3 heading generation.

faker_file.contrib.odt_file.**add_h4_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h4 heading generation.

faker_file.contrib.odt_file.**add_h5_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h5 heading generation.

faker_file.contrib.odt_file.**add_h6_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the h6 heading generation.

faker_file.contrib.odt_file.**add_heading**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the heading generation.

faker_file.contrib.odt_file.**add_page_break**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for page break generation.

faker_file.contrib.odt_file.**add_paragraph**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the paragraph generation.

faker_file.contrib.odt_file.**add_picture**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the picture generation.

faker_file.contrib.odt_file.**add_table**(*provider*, *document*, *data*, *counter*, ***kwargs*)
Callable responsible for the table generation.

15.15.1.1.2.10 Module contents

15.15.1.1.3 faker_file.providers package

15.15.1.1.3.1 Subpackages

15.15.1.1.3.2 faker_file.providers.augment_file_from_dir package

15.15.1.1.3.3 Subpackages

15.15.1.1.3.4 faker_file.providers.augment_file_from_dir.augmenters package

15.15.1.1.3.5 Submodules

15.15.1.1.3.6 faker_file.providers.augment_file_from_dir.augmenters.nlpaug_augmenter module

15.15.1.1.3.7 Module contents

15.15.1.1.3.8 faker_file.providers.augment_file_from_dir.extractors package

15.15.1.1.3.9 Submodules

15.15.1.1.3.10 faker_file.providers.augment_file_from_dir.extractors.tika_extractor module

15.15.1.1.3.11 Module contents

15.15.1.1.3.12 Module contents

class faker_file.providers.augment_file_from_dir.**AugmentFileFromDirProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

Augment file from given directory provider.

Usage example:

```
from faker import Faker
from faker_file.providers.augment_file_from_dir import (
    AugmentFileFromDirProvider,
)

FAKER = Faker()
FAKER.add_provider(AugmentFileFromDirProvider)

file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/",
)
```

Usage example with options:

```
file = FAKER.augment_file_from_dir(
    source_dir_path="/tmp/tmp/", prefix="zzz", extensions={"docx", "pdf"}

)
```

augment_file_from_dir(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *wrap_chars_after: Optional[int] = None*, *text_extractor_cls: Optional[Union[str, Type[BaseTextExtractor]]] = DEFAULT_EXTRACTOR*, *text_extractor_kwargs: Optional[Dict[str, Any]] = None*, *text_augmenter_cls: Optional[Union[str, Type[BaseTextAugmenter]]] = DEFAULT_AUGMENTER*, *text_augmenter_kwargs: Optional[Dict[str, Any]] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

augment_file_from_dir(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *wrap_chars_after: Optional[int] = None*, *text_extractor_cls: Optional[Union[str, Type[BaseTextExtractor]]] = DEFAULT_EXTRACTOR*, *text_extractor_kwargs: Optional[Dict[str, Any]] = None*, *text_augmenter_cls: Optional[Union[str, Type[BaseTextAugmenter]]] = DEFAULT_AUGMENTER*, *text_augmenter_kwargs: Optional[Dict[str, Any]] = None*, ***kwargs*) → *StringValue*

Augment a random file from given directory.

Parameters

- **source_dir_path** – Source files directory.
- **extensions** – Allowed extensions.
- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **text_extractor_cls** – Text extractor class.
- **text_extractor_kwargs** – Text extractor kwargs.
- **text_augmenter_cls** – Text augmenter class.
- **text_augmenter_kwargs** – Text augmenter kwargs.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = ''

15.15.1.1.3.13 faker_file.providers.base package

15.15.1.1.3.14 Submodules

15.15.1.1.3.15 faker_file.providers.base.image_generator module

class faker_file.providers.base.image_generator.**BaseImageGenerator**(***kwargs*)

Bases: object

Base image generator.

generate(*content: str*, *data: Dict[str, Any]*, *provider: Union[Faker, Generator, Provider]*) → bytes

handle_kwargs(***kwargs*)

Handle kwargs.

15.15.1.1.3.16 faker_file.providers.base.mp3_generator module

class faker_file.providers.base.mp3_generator.**BaseMp3Generator**(*content: str*, *generator: Union[Faker, Generator, Provider]*, ***kwargs*)

Bases: object

Base MP3 generator.

content: str

generate(***kwargs*) → bytes

generator: Union[Faker, Generator, Provider]

handle_kwargs(***kwargs*)

Handle kwargs.

15.15.1.1.3.17 faker_file.providers.base.pdf_generator module

class faker_file.providers.base.pdf_generator.**BasePdfGenerator**(***kwargs*)

Bases: object

Base PDF generator.

generate(*content: str*, *data: Dict[str, Any]*, *provider: Union[Faker, Generator, Provider]*) → bytes

handle_kwargs(***kwargs*)

Handle kwargs.

15.15.1.1.3.18 faker_file.providers.base.text_augmenter module

class faker_file.providers.base.text_augmenter.**BaseTextAugmenter**(***kwargs*)

Bases: object

Base text augmenter.

augment(*text: str*) → str

handle_kwargs(***kwargs*)

Handle kwargs.

15.15.1.1.3.19 faker_file.providers.base.text_extractor module

class faker_file.providers.base.text_extractor.**BaseTextExtractor**(***kwargs*)

Bases: object

Base text extractor.

extract(*source_file: Union[Path, str]*) → str

handle_kwargs(***kwargs*)

Handle kwargs.

path: str

15.15.1.1.3.20 Module contents

15.15.1.1.3.21 faker_file.providers.helpers package

15.15.1.1.3.22 Submodules

15.15.1.1.3.23 faker_file.providers.helpers.inner module

faker_file.providers.helpers.inner.**create_inner_augment_image_from_path**(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_augment_image_from_path**(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, ***kwargs*) → *StringValue*

Create inner augment_image_from_path file.

faker_file.providers.helpers.inner.**create_inner_augment_random_image_from_dir**(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_augment_random_image_from_dir**(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, ***kwargs*) → *StringValue*

Create inner augment_random_image_from_dir file.

faker_file.providers.helpers.inner.**create_inner_bin_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *length: int = 1 * 1024 * 1024*, *content: Optional[bytes] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_bin_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *length: int = 1 * 1024 * 1024*, *content: Optional[bytes] = None*, ***kwargs*) → *StringValue*

Create inner BIN file.

faker_file.providers.helpers.inner.**create_inner_csv_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *header: Optional[Sequence[str]] = None*, *data_columns: Tuple[str, str] = ('{{name}}', '{{address}}')*, *num_rows: int = 10*, *include_row_ids: bool = False*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_csv_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *header: Optional[Sequence[str]] = None*, *data_columns: Tuple[str, str] = ('{{name}}', '{{address}}')*, *num_rows: int = 10*, *include_row_ids: bool = False*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner CSV file.

faker_file.providers.helpers.inner.**create_inner_docx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_docx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner DOCX file.

faker_file.providers.helpers.inner.**create_inner_eml_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_eml_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner EML file.

faker_file.providers.helpers.inner.**create_inner_epub_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *title: Optional[str] = None*, *chapter_title: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_epub_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *title: Optional[str] = None*, *chapter_title: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner EPUB file.

faker_file.providers.helpers.inner.**create_inner_file_from_path**(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_file_from_path**(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, ***kwargs*) → *StringValue*

Create inner file from path.

faker_file.providers.helpers.inner.**create_inner_generic_file**(*content: Union[bytes, str]*, *extension: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_generic_file**(*content: Union[bytes, str]*, *extension: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner generic file.

faker_file.providers.helpers.inner.**create_inner_graphic_ico_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_graphic_ico_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner graphic ICO file.

faker_file.providers.helpers.inner.**create_inner_graphic_jpeg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_graphic_jpeg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner graphic JPEG file.

faker_file.providers.helpers.inner.**create_inner_graphic_pdf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_graphic_pdf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner graphic PDF file.

faker_file.providers.helpers.inner.**create_inner_graphic_png_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_graphic_png_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner graphic PNG file.

faker_file.providers.helpers.inner.**create_inner_graphic_webp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_graphic_webp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner graphic WEBP file.

faker_file.providers.helpers.inner.**create_inner_ico_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_ico_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner ICO file.

faker_file.providers.helpers.inner.**create_inner_jpeg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_jpeg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner JPEG file.

faker_file.providers.helpers.inner.**create_inner_json_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[List] = None*, *num_rows: int = 10*, *indent: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_json_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[List] = None*, *num_rows: int = 10*, *indent: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner JSON file.

faker_file.providers.helpers.inner.**create_inner_mp3_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_AUDIO_MAX_NB_CHARS*, *content: Optional[str] = None*, *mp3_generator_cls: Optional[Union[str, Type[BaseMp3Generator]]] = None*, *mp3_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_mp3_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_AUDIO_MAX_NB_CHARS*, *content: Optional[str] = None*, *mp3_generator_cls: Optional[Union[str, Type[BaseMp3Generator]]] = None*, *mp3_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner ODS file.

faker_file.providers.helpers.inner.**create_inner_odp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_odp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → Union[*BytesValue*, *StringValue*]

Create inner ODP file.

faker_file.providers.helpers.inner.**create_inner_ods_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_ods_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner ODS file.

faker_file.providers.helpers.inner.**create_inner_odt_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_odt_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner ODT file.

faker_file.providers.helpers.inner.**create_inner_pdf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *pdf_generator_cls: Optional[Union[str, Type[BasePdfGenerator]]] = None*, *pdf_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_pdf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *pdf_generator_cls: Optional[Union[str, Type[BasePdfGenerator]]] = None*, *pdf_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner PDF file.

faker_file.providers.helpers.inner.**create_inner_png_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_png_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner PNG file.

faker_file.providers.helpers.inner.**create_inner_pptx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_pptx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner PPTX file.

faker_file.providers.helpers.inner.**create_inner_random_file_from_dir**(*source_dir_path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_random_file_from_dir**(*source_dir_path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, ***kwargs*) → *StringValue*

Create inner random_file_from_dir file.

faker_file.providers.helpers.inner.**create_inner_rtf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_rtf_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner RTF file.

faker_file.providers.helpers.inner.**create_inner_svg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_svg_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner SVG file.

faker_file.providers.helpers.inner.**create_inner_tar_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, *compression: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_tar_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, *compression: Optional[str] = None*, ***kwargs*) → *StringValue*

Create inner TAR file.

faker_file.providers.helpers.inner.**create_inner_txt_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_txt_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner TXT file.

faker_file.providers.helpers.inner.**create_inner_webp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_webp_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner WEBP file.

faker_file.providers.helpers.inner.**create_inner_xlsx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_xlsx_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner XLSX file.

faker_file.providers.helpers.inner.**create_inner_xml_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *root_element: str = 'root'*, *row_element: str = 'row'*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_xml_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *root_element: str = 'root'*, *row_element: str = 'row'*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Create inner XML file.

faker_file.providers.helpers.inner.**create_inner_zip_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

faker_file.providers.helpers.inner.**create_inner_zip_file**(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *generator: Optional[Union[Faker, Generator, Provider]] = None*, *options: Optional[Dict[str, Any]] = None*, ***kwargs*) → *StringValue*

Create inner ZIP file.

faker_file.providers.helpers.inner.**fuzzy_choice_create_inner_file**(*func_choices: List[Tuple[Callable, Dict[str, Any]]]*, ***kwargs*) → Union[*BytesValue*, *StringValue*]

Create inner file from given list of function choices.

Parameters

func_choices – List of functions to choose from.

Returns
StringValue.

Usage example:

```
from faker import Faker
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)
from faker_file.storages.filesystem import FileSystemStorage

FAKER = Faker()
STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}
file = fuzzy_choice_create_inner_file(
    [
        (create_inner_docx_file, kwargs),
        (create_inner_epub_file, kwargs),
        (create_inner_txt_file, kwargs),
    ]
)
```

You could use it in archives to make a variety of different file types within the archive.

```
from faker import Faker
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_epub_file,
    create_inner_txt_file,
    fuzzy_choice_create_inner_file,
)
from faker_file.providers.zip_file import ZipFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FAKER = Faker()
FAKER.add_provider(ZipFileProvider)

STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}
file = FAKER.zip_file(
    prefix="zzz_archive_",
    options={
        "count": 50,
        "create_inner_file_func": fuzzy_choice_create_inner_file,
        "create_inner_file_args": {
            "func_choices": [
                (create_inner_docx_file, kwargs),
                (create_inner_epub_file, kwargs),
```

(continues on next page)

(continued from previous page)

```
                (create_inner_txt_file, kwargs),
            ],
        },
        "directory": "zzz",
    }
)
```

faker_file.providers.helpers.inner.**list_create_inner_file**(*func_list: List[Tuple[Callable, Dict[str, Any]]]*, ***kwargs*) → List[Union[*BytesValue*, *StringValue*]]

Generates multiple files based on the provided list of functions and arguments.

Parameters
func_list – List of tuples, each containing a function to generate a file and its arguments.

Returns
List of generated file names.

Usage example:

```
from faker import Faker
from faker_file.providers.helpers.inner import (
    create_inner_docx_file,
    create_inner_xml_file,
    list_create_inner_file,
)
from faker_file.providers.zip_file import ZipFileProvider
from faker_file.storages.filesystem import FileSystemStorage

FAKER = Faker()
FAKER.add_provider(ZipFileProvider)
STORAGE = FileSystemStorage()

kwargs = {"storage": STORAGE, "generator": FAKER}
file = FAKER.zip_file(
    basename="alice-looking-through-the-glass",
    options={
        "create_inner_file_func": list_create_inner_file,
        "create_inner_file_args": {
            "func_list": [
                (create_inner_docx_file, {"basename": "doc"}),
                (create_inner_xml_file, {"basename": "doc_metadata"}),
                (create_inner_xml_file, {"basename": "doc_isbn"}),
            ],
        },
    }
)
```

Note, that while all other inner functions return back *Union[BytesValue, StringValue]* value, *list_create_inner_file* returns back a *List[Union[BytesValue, StringValue]]* value.

Notably, all inner functions were designed to support archives (such as ZIP, TAR and EML, but the list may grow in the future). If the inner function passed in the *create_inner_file_func* argument returns a List of *Union[BytesValue, StringValue]* values, the *option* argument is being ignored and generated files are simply limited to what has been passed in the *func_list* list of tuples.

15.15.1.1.3.24 Module contents

15.15.1.1.3.25 faker_file.providers.image package

15.15.1.1.3.26 Submodules

15.15.1.1.3.27 faker_file.providers.image.augment module

faker_file.providers.image.augment.**add_brightness**(*img: Image*, *lower: float = 1*, *upper: float = 2*) → Image

Increase the image's brightness by a random factor.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement. Default is 0.5.
- **upper** – Upper bound for the random enhancement. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**add_contrast**(*img: Image*, *lower: float = 1*, *upper: float = 2*) → Image

Enhance the image's contrast by a random factor.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement. Default is 0.5.
- **upper** – Upper bound for the random enhancement. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**add_darkness**(*img: Image*, *lower: float = 0.5*, *upper: float = 1*) → Image

Decrease the image's brightness by a random factor.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement. Default is 0.5.
- **upper** – Upper bound for the random enhancement. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**add_saturation**(*img: Image*, *lower: float = 1*, *upper: float = 2*) → Image

Enhance the image's color saturation by a random factor.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement. Default is 0.5.

- **upper** – Upper bound for the random enhancement. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**augment_image**(*image_bytes: bytes*, *augmentations: ~typing.Optional[~typing.List[~typing.Tuple[~typing.Callable*, *~typing.Dict[str*, *~typing.Any]]]] = None*, *num_steps: ~typing.Optional[int] = None*, *pop_func: ~typing.Callable = <function random_pop>*) → bytes

Augment the input image with a series of random augmentation methods.

Read an image provided in bytes format, applies a specified number of random augmentation methods from a given list, and then returns the augmented image in bytes format. If no list of methods is provided, a default list is used. If no number of steps (methods) is specified, all methods will be applied.

Parameters

- **image_bytes** – Input image in bytes format.
- **augmentations** – List of tuples of callable augmentation functions and their respective keyword arguments. If not provided, the default augmentation functions will be used.
- **num_steps** – Number of augmentation steps (functions) to be applied. If not specified, the length of the *augmentations* list will be used.
- **pop_func** – Callable to pop items from *augmentations* list. By default, the *random_pop* is used, which pops items in random order. If you want the order of augmentations to be constant and as given, replace it with *list.pop* (*pop_func=list.pop*).

Returns
Augmented image in bytes format.

faker_file.providers.image.augment.**augment_image_file**(*image_path: str*, *augmentations: ~typing.Optional[~typing.List[~typing.Tuple[~typing.Callable*, *~typing.Dict[str*, *~typing.Any]]]] = None*, *num_steps: ~typing.Optional[int] = None*, *pop_func: ~typing.Callable = <function random_pop>*) → bytes

Augment image from path.

Augment the input image with a series of random augmentation functions.

faker_file.providers.image.augment.**color_jitter**(*img: Image*, *lower: float = 0.5*, *upper: float = 1.5*) → Image

Randomly adjust the image's brightness, contrast, saturation, and hue.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement multiplier. Default is 0.5.
- **upper** – Upper bound for the random enhancement multiplier. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**decrease_contrast**(*img: Image*, *lower: float = 0.5*, *upper: float = 1*) → Image

Reduce the image's contrast by a random factor.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random enhancement. Default is 0.5.
- **upper** – Upper bound for the random enhancement. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**equalize**(*img: Image*) → Image

Equalize the image's histogram.

faker_file.providers.image.augment.**flip_horizontal**(*img: Image*) → Image

Flip the image horizontally.

faker_file.providers.image.augment.**flip_vertical**(*img: Image*) → Image

Flip the image vertically.

faker_file.providers.image.augment.**gaussian_blur**(*img: Image*, *lower: float = 0.5*, *upper: float = 3*) → Image

Apply Gaussian blur to the image using a random radius.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random radius. Default is 0.5.
- **upper** – Upper bound for the random radius. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**grayscale**(*img: Image*) → Image

Convert the image to grayscale.

faker_file.providers.image.augment.**random_crop**(*img: Image*, *lower: float = 0.6*, *upper: float = 0.9*) → Image

Randomly crop a portion of the image.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random crop. Default is 0.5.
- **upper** – Upper bound for the random crop. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**resize_height**(*img: Image*, *lower: float = 0.5*, *upper: float = 1.5*) → Image

Resize the image in height by a random percentage.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random resize. Default is 0.5.
- **upper** – Upper bound for the random resize. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**resize_width**(*img: Image*, *lower: float = 0.5*, *upper: float = 1.5*) → Image

Resize the image in width by a random percentage.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random resize. Default is 0.5.
- **upper** – Upper bound for the random resize. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**rotate**(*img: Image*, *lower: int = -45*, *upper: int = 45*) → Image

Rotate the image by a random angle.

Parameters

- **img** – Input image to be adjusted.
- **lower** – Lower bound for the random rotation. Default is 0.5.
- **upper** – Upper bound for the random rotation. Default is 1.5.

Returns
Adjusted image.

faker_file.providers.image.augment.**solarize**(*img: Image*, *threshold: int = 128*) → Image

Invert pixel values above a specified threshold.

15.15.1.1.3.28 faker_file.providers.image.imgkit_generator module

class faker_file.providers.image.imgkit_generator.**ImgkitImageGenerator**(***kwargs*)

Bases: *BaseImageGenerator*

Imgkit image generator.

Usage example:

```
from faker import Faker
from faker_file.providers.png_file import PngFileProvider
from faker_file.providers.image.imgkit_generator import (
    ImgkitImageGenerator
)

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file(
    img_generator_cls=ImgkitImageGenerator
)
```

Using *DynamicTemplate*:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.image.imgkit_snippets import (
    add_h1_heading,
    add_h2_heading,
    add_h3_heading,
    add_h4_heading,
    add_h5_heading,
    add_h6_heading,
    add_heading,
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a file with lots of elements
file = FAKER.png_file(
    image_generator_cls=ImgkitImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_paragraph, {}),
            (add_h2_heading, {}),
            (add_h3_heading, {}),
            (add_h4_heading, {}),
            (add_h5_heading, {}),
            (add_h6_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_page_break, {}),
            (add_h6_heading, {}),
            (add_table, {}),
            (add_paragraph, {}),
        ]
    )
)
```

encoding: str = 'utf-8'

generate(*content: str*, *data: Dict[str, Any]*, *provider: Union[Faker, Generator, Provider]*, ***kwargs*) → bytes

Generate image.

handle_kwargs(***kwargs*) → None

Handle kwargs.

15.15.1.1.3.29 faker_file.providers.image.pil_generator module

class faker_file.providers.image.pil_generator.**PilImageGenerator**(***kwargs*)

Bases: *BaseImageGenerator*

PIL image generator.

Usage example:

```
from faker import Faker
from faker_file.providers.png_file import PngFileProvider
from faker_file.providers.image.pil_generator import PilImageGenerator

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
)
```

With options:

```
file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    image_generator_kwargs={
        "spacing": 6,
    },
    wrap_chars_after=119,
)
```

With dynamic content:

```
from faker import Faker
from faker_file.base import DynamicTemplate
from faker_file.contrib.image.pil_snippets import *
from faker_file.providers.image.pil_generator import PilImageGenerator
from faker_file.providers.png_file import PngFileProvider

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
        ]
    )
)
```

(continues on next page)

(continued from previous page)

```
file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_table, {"rows": 5, "cols": 4}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=PilImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {"margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
```

(continues on next page)

(continued from previous page)

```
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
        ]
    )
)
```

combine_images_vertically()

create_image_instance(*height: Optional[int] = None*) → <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/faker-file/envs/dev/lib/python3.10/site-packages/PIL/Image.py'>

encoding: str = 'utf-8'

classmethod find_max_fit_for_multi_line_text(*draw: <module 'PIL.ImageDraw' from '/home/docs/checkouts/readthedocs.org/user_builds/faker-file/envs/dev/lib/python3.10/site-packages/PIL/ImageDraw.py'>, lines: ~typing.List[str], font: <module 'PIL.ImageFont' from '/home/docs/checkouts/readthedocs.org/user_builds/faker-file/envs/dev/lib/python3.10/site-packages/PIL/ImageFont.py'>, max_width: int*)

classmethod find_max_fit_for_single_line_text(*draw: ImageDraw, text: str, font: <module 'PIL.ImageFont' from '/home/docs/checkouts/readthedocs.org/user_builds/faker-file/envs/dev/lib/python3.10/site-packages/PIL/ImageFont.py'>, max_width: int*) → int

font: str = 'Pillow/Tests/fonts/DejaVuSans.ttf'

font_size: int = 12

generate(*content: str, data: Dict[str, Any], provider: Union[Faker, Generator, Provider], **kwargs*) → bytes

Generate image.

handle_kwargs(***kwargs*) → None

Handle kwargs.

line_height: int = 14

page_height: int = 1123

page_width: int = 794

save_and_start_new_page()

spacing: int = 6

```
start_new_page()
```

15.15.1.1.3.30 faker_file.providers.image.weasyprint_generator module

class faker_file.providers.image.weasyprint_generator.**WeasyPrintImageGenerator**(***kwargs*)

Bases: *BaseImageGenerator*

WeasyPrint and Pdf2Image ImageGenerator image generator.

Usage example:

```
from faker import Faker
from faker_file.providers.png_file import PngFileProvider
from faker_file.providers.image.weasyprint_generator import (
    WeasyPrintImageGenerator
)

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file(
    img_generator_cls=WeasyPrintImageGenerator
)
```

With dynamic content:

```
from faker import Faker
from faker_file.base import DynamicTemplate
from faker_file.contrib.image.weasyprint_snippets import *
from faker_file.providers.image.weasyprint_generator import (
    WeasyPrintImageGenerator
)
from faker_file.providers.png_file import PngFileProvider

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_paragraph, {"max_nb_chars": 500}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
```

(continues on next page)

(continued from previous page)

```
        [
            (add_h1_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_paragraph, {}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_picture, {}),
            (add_paragraph, {"max_nb_chars": 500}),
            (add_table, {"rows": 5, "cols": 4}),
        ]
    )
)

file = FAKER.png_file(
    image_generator_cls=WeasyPrintImageGenerator,
    content=DynamicTemplate(
        [
            (add_h1_heading, {"margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
```

(continues on next page)

(continued from previous page)

```
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
            (add_picture, {"margin": (2, 2)}),
            (add_paragraph, {"max_nb_chars": 500, "margin": (2, 2)}),
        ]
    )
)
```

create_image_instance(*width: Optional[int] = None*, *height: Optional[int] = None*) → <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/faker-file/envs/dev/lib/python3.10/site-packages/PIL/Image.py'>

encoding: str = 'utf-8'

generate(*content: str*, *data: Dict[str, Any]*, *provider: Union[Faker, Generator, Provider]*, ***kwargs*) → bytes

Generate image.

handle_kwargs(***kwargs*) → None

Handle kwargs.

page_height: int = 1123

page_width: int = 794

wrap(*content: str*) → str

wrapper_tag: str = 'div'

15.15.1.1.3.31 Module contents

15.15.1.1.3.32 faker_file.providers.mixins package

15.15.1.1.3.33 Submodules

15.15.1.1.3.34 faker_file.providers.mixins.graphic_image_mixin module

class faker_file.providers.mixins.graphic_image_mixin.**GraphicImageMixin**

Bases: *FileMixin*

Graphic image mixin.

15.15.1.1.3.35 faker_file.providers.mixins.image_mixin module

class faker_file.providers.mixins.image_mixin.**ImageMixin**

Bases: *FileMixin*

Image mixin.

15.15.1.1.3.36 faker_file.providers.mixins.tablular_data_mixin module

class faker_file.providers.mixins.tablular_data_mixin.**TabularDataMixin**

Bases: *FileMixin*

Tabular data mixin.

15.15.1.1.3.37 Module contents

15.15.1.1.3.38 faker_file.providers.mp3_file package

15.15.1.1.3.39 Subpackages

15.15.1.1.3.40 faker_file.providers.mp3_file.generators package

15.15.1.1.3.41 Submodules

15.15.1.1.3.42 faker_file.providers.mp3_file.generators.edge_tts_generator module

class faker_file.providers.mp3_file.generators.edge_tts_generator.**EdgeTtsMp3Generator**(*content: str*, *generator: Union[Faker, Generator, Provider]*, ***kwargs*)

Bases: *BaseMp3Generator*

Edge Text-to-Speech generator.

Usage example:

```
from faker import Faker
from faker_file.providers.mp3_file import Mp3FileProvider
from faker_file.providers.mp3_file.generators import edge_tts_generator

FAKER = Faker()
FAKER.add_provider(Mp3FileProvider)

file = FAKER.mp3_file(
    mp3_generator_cls=edge_tts_generator.EdgeTtsMp3Generator
)
```

generate(***kwargs*) → bytes

Generate MP3.

handle_kwargs(***kwargs*) → None

Handle kwargs.

voice: str = 'en-GB-SoniaNeural'

15.15.1.1.3.43 faker_file.providers.mp3_file.generators.gtts_generator module

class faker_file.providers.mp3_file.generators.gtts_generator.**GttsMp3Generator**(*content: str*, *generator: Union[Faker, Generator, Provider]*, ***kwargs*)

Bases: *BaseMp3Generator*

Google Text-to-Speech generator.

Usage example:

```
from faker import Faker
from faker_file.providers.mp3_file import Mp3FileProvider
from faker_file.providers.mp3_file.generators.gtts_generator import (
    GttsMp3Generator,
)

FAKER = Faker()
FAKER.add_provider(Mp3FileProvider)

file = FAKER.mp3_file(
    mp3_generator_cls=GttsMp3Generator
)
```

generate(***kwargs*) → bytes

Generate MP3.

handle_kwargs(***kwargs*) → None

Handle kwargs.

lang: str = 'en'

tld: str = 'com'

15.15.1.1.3.44 Module contents

15.15.1.1.3.45 Module contents

class faker_file.providers.mp3_file.**Mp3FileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

MP3 file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.mp3_file import Mp3FileProvider
```

(continues on next page)

(continued from previous page)

```
FAKER = Faker()
FAKER.add_provider(Mp3FileProvider)

file = FAKER.mp3_file()
```

Usage example with options:

```
file = FAKER.mp3_file(
    prefix="zzz",
    max_nb_chars=500,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.mp3_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=500,
)
```

Default MP3 generator class is *GttsMp3Generator* which uses Google Text-to-Speech services to generate an MP3 file from given or randomly generated text. It does not require additional services to run and the only dependency here is the *gtts* package. You can however implement your own custom MP3 generator class and pass it to te *mp3_file* method in *mp3_generator_cls* argument instead of the default *GttsMp3Generator*.

Usage with custom MP3 generator class.

```
# Imaginary `marytts` Python library
from marytts import MaryTTS

# Import BaseMp3Generator
from faker_file.providers.base.mp3_generator import (
    BaseMp3Generator,
)

# Define custom MP3 generator
class MaryTtsMp3Generator(BaseMp3Generator):

    locale: str = "cmu-rms-hsmm"
    voice: str = "en_US"

    def handle_kwargs(self, **kwargs) -> None:
        # Since it's impossible to unify all TTS systems it's allowed
        # to pass arbitrary arguments to the `BaseMp3Generator`
        # constructor. Each implementation class contains its own
        # additional tuning arguments. Check the source code of the
        # implemented MP3 generators as an example.
```

(continues on next page)

(continued from previous page)

```
        if "locale" in kwargs:
            self.locale = kwargs["locale"]
        if "voice" in kwargs:
            self.voice = kwargs["voice"]

    def generate(self) -> bytes:
        # Your implementation here. Note, that `self.content`
        # in this context is the text to make MP3 from.
        # `self.generator` would be the `Faker` or `Generator`
        # instance from which you could extract information on
        # active locale.
        # What comes below is pseudo implementation.
        mary_tts = MaryTTS(locale=self.locale, voice=self.voice)
        return mary_tts.synth_mp3(self.content)

# Generate MP3 file from random text
file = FAKER.mp3_file(
    mp3_generator_cls=MaryTtsMp3Generator,
)
```

extension: str = 'mp3'

mp3_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_AUDIO_MAX_NB_CHARS*, *content: Optional[str] = None*, *mp3_generator_cls: Optional[Union[str, Type[BaseMp3Generator]]] = DEFAULT_MP3_GENERATOR*, *mp3_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

mp3_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_AUDIO_MAX_NB_CHARS*, *content: Optional[str] = None*, *mp3_generator_cls: Optional[Union[str, Type[BaseMp3Generator]]] = DEFAULT_MP3_GENERATOR*, *mp3_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a MP3 file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **mp3_generator_cls** – Mp3 generator class.
- **mp3_generator_kwargs** – Mp3 generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.46 faker_file.providers.pdf_file package

15.15.1.1.3.47 Subpackages

15.15.1.1.3.48 faker_file.providers.pdf_file.generators package

15.15.1.1.3.49 Submodules

15.15.1.1.3.50 faker_file.providers.pdf_file.generators.pdfkit_generator module

class faker_file.providers.pdf_file.generators.pdfkit_generator.**PdfkitPdfGenerator**(***kwargs*)

Bases: *BasePdfGenerator*

Pdfkit PDF generator.

Usage example:

```
from faker import Faker
from faker_file.providers.pdf_file import PdfFileProvider
from faker_file.providers.pdf_file.generators.pdfkit_generator import (
    PdfkitPdfGenerator
)

FAKER = Faker()
FAKER.add_provider(PdfFileProvider)

file = FAKER.pdf_file(pdf_generator_cls=PdfkitPdfGenerator)
```

Using *DynamicTemplate*:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.pdf_file.pdfkit_snippets import (
    add_h1_heading,
    add_h2_heading,
    add_h3_heading,
    add_h4_heading,
    add_h5_heading,
    add_h6_heading,
    add_heading,
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

# Create a file with lots of elements
file = FAKER.pdf_file(
    pdf_generator_cls=PdfkitPdfGenerator,
    content=DynamicTemplate(
```

(continues on next page)

(continued from previous page)

```
        [
            (add_h1_heading, {}),
            (add_paragraph, {}),
            (add_h2_heading, {}),
            (add_h3_heading, {}),
            (add_h4_heading, {}),
            (add_h5_heading, {}),
            (add_h6_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_page_break, {}),
            (add_h6_heading, {}),
            (add_table, {}),
            (add_paragraph, {}),
        ]
    )
)
```

encoding: str = 'utf-8'

generate(*content: Union[str,* DynamicTemplate*]*, *data: Dict[str, Any]*, *provider: Union[Faker, Generator, Provider]*, ***kwargs*) → bytes

Generate PDF.

handle_kwargs(***kwargs*) → None

Handle kwargs.

15.15.1.1.3.51 faker_file.providers.pdf_file.generators.reportlab_generator module

15.15.1.1.3.52 Module contents

15.15.1.1.3.53 Module contents

class faker_file.providers.pdf_file.**GraphicPdfFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic PDF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.pdf_file import GraphicPdfFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicPdfFileProvider)

file = FAKER.graphic_pdf_file()
```

Usage example with options:

```
file = FAKER.graphic_pdf_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_pdf_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'pdf'

graphic_pdf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_pdf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic PDF file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'pdf'

class faker_file.providers.pdf_file.**PdfFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

PDF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.pdf_file import PdfFileProvider

FAKER = Faker()
FAKER.add_provider(PdfFileProvider)

file = FAKER.pdf_file()
```

Usage example with options:

```
file = FAKER.pdf_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.pdf_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Default PDF generator class is *PdfkitPdfGenerator* which uses *pdfkit* Python package and *wkhtmltopdf* system package for generating PDFs from randomly generated text. The quality of the produced PDFs is very good, but it's less performant than *ReportlabPdfGenerator* (factor 40x), which does not require additional system dependencies to run. To use it, pass *ReportlabPdfGenerator* class in *pdf_generator_cls* argument.

```
from faker_file.providers.pdf_file.generators import (
    reportlab_generator,
)

file = FAKER.pdf_file(
    max_nb_chars=1_000,
    wrap_chars_after=80,
    pdf_generator_cls=reportlab_generator.ReportlabPdfGenerator,
)
```

extension: str = 'pdf'

pdf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *pdf_generator_cls: Optional[Union[str, Type[BasePdfGenerator]]] = DEFAULT_PDF_GENERATOR*, *pdf_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

pdf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *pdf_generator_cls: Optional[Union[str, Type[BasePdfGenerator]]] = DEFAULT_PDF_GENERATOR*, *pdf_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a PDF file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **pdf_generator_cls** – PDF generator class.
- **pdf_generator_kwargs** – PDF generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.54 Submodules

15.15.1.1.3.55 faker_file.providers.augment_image_from_path module

class faker_file.providers.augment_image_from_path.**AugmentImageFromPathProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

Augment image from given path provider.

Usage example:

```
from faker import Faker
from faker_file.providers.augment_image_from_path import (
    AugmentImageFromPathProvider
)

FAKER = Faker()
FAKER.add_provider(AugmentImageFromPathProvider)

file = FAKER.augment_image_from_path(
    path="/path/to/image.png"
)
```

Usage example with options:

```
file = FAKER.augment_image_from_path(
    path="/path/to/image.png",
    prefix="zzz",
)
```

augment_image_from_path(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, *raw: bool = True*, ***kwargs*) → *BytesValue*

augment_image_from_path(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, ***kwargs*) → *StringValue*

Augment an image from given path.

Parameters

- **path** – Path to source file.
- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **augmentations** – List of tuples of callable augmentation functions and their respective keyword arguments. If not provided, the default augmentation functions will be used.
- **num_steps** – Number of augmentation steps (functions) to be applied. If not specified, the length of the *augmentations* list will be used.
- **pop_func** – Callable to pop items from *augmentations* list. By default, the *random_pop* is used, which pops items in random order. If you want the order of augmentations to be constant and as given, replace it with *list.pop* (*pop_func=list.pop*).
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = ''

15.15.1.1.3.56 faker_file.providers.augment_random_image_from_dir module

class faker_file.providers.augment_random_image_from_dir.**AugmentRandomImageFromDirProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

Augment image from given directory provider.

Usage example:

```
from faker import Faker
from faker_file.providers.augment_random_image_from_dir import (
    AugmentRandomImageFromDirProvider
)

FAKER = Faker()
FAKER.add_provider(AugmentRandomImageFromDirProvider)

file = FAKER.augment_random_image_from_dir(
    source_dir_path="/tmp/tmp/"
)
```

Usage example with options:

```
file = FAKER.augment_random_image_from_dir(
    source_dir_path="/tmp/tmp/",
    prefix="zzz",
    extensions={"jpeg", "png"},
)
```

augment_random_image_from_dir(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, *raw: bool = True*, ***kwargs*) → *BytesValue*

augment_random_image_from_dir(*source_dir_path: str*, *extensions: Optional[Iterable[str]] = None*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *augmentations: Optional[List[Tuple[Callable, Dict[str, Any]]]] = None*, *num_steps: Optional[int] = None*, *pop_func: Callable = random_pop*, ***kwargs*) → *StringValue*

Augment a random image from given directory.

Parameters

- **source_dir_path** – Source files directory.
- **extensions** – Allowed extensions.
- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **augmentations** – List of tuples of callable augmentation functions and their respective keyword arguments. If not provided, the default augmentation functions will be used.

- **num_steps** – Number of augmentation steps (functions) to be applied. If not specified, the length of the *augmentations* list will be used.
- **pop_func** – Callable to pop items from *augmentations* list. By default, the *random_pop* is used, which pops items in random order. If you want the order of augmentations to be constant and as given, replace it with *list.pop* (*pop_func=list.pop*).
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

extension: str = ''

15.15.1.1.3.57 faker_file.providers.bin_file module

class faker_file.providers.bin_file.**BinFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

BIN file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.bin_file import BinFileProvider

FAKER = Faker()
FAKER.add_provider(BinFileProvider)

file = FAKER.bin_file()
```

Usage example with options:

```
file = FAKER.bin_file(
    prefix="zzz",
    length=1024**2,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.bin_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    length=1024**2,
)
```

Usage example with AWS S3 storage:

```
from faker_file.storages.aws_s3 import AWSS3Storage

file = FAKER.bin_file(
    storage=AWSS3Storage(bucket_name="My-test-bucket"),
    prefix="zzz",
    length=1024**2,
)
```

bin_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *length: int = 1 * 1024 * 1024*, *content: Optional[bytes] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

bin_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *length: int = 1 * 1024 * 1024*, *content: Optional[bytes] = None*, ***kwargs*) → *StringValue*

Generate a BIN file with random bytes.

Parameters

- **storage** – Storage class. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **length** –
- **content** – File content. If given, used as is.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'bin'

15.15.1.1.3.58 faker_file.providers.bmp_file module

class faker_file.providers.bmp_file.**BmpFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

BMP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.bmp_file import BmpFileProvider

FAKER = Faker()
FAKER.add_provider(BmpFileProvider)

file = FAKER.bmp_file()
```

Usage example with options:

```
file = FAKER.bmp_file(
    prefix="zzz",
```

(continues on next page)

(continued from previous page)

```
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.bmp_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

bmp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

bmp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a GIF file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'bmp'

image_format: str = 'bmp'

class faker_file.providers.bmp_file.**GraphicBmpFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic BMP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.bmp_file import GraphicBmpFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicBmpFileProvider)

file = FAKER.graphic_bmp_file()
```

Usage example with options:

```
file = FAKER.graphic_bmp_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_bmp_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'bmp'

graphic_bmp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_bmp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic BMP file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'bmp'

15.15.1.1.3.59 faker_file.providers.csv_file module

class faker_file.providers.csv_file.**CsvFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

CSV file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.csv_file import CsvFileProvider

FAKER = Faker()
FAKER.add_provider(CsvFileProvider)

file = FAKER.csv_file()
```

Usage example with options:

```
file = FAKER.csv_file(
    prefix="zzz",
    num_rows=100,
    data_columns=('{{name}}', '{{sentence}}', '{{address}}'),
    include_row_ids=True,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.csv_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
```

(continues on next page)

(continued from previous page)

```
    prefix="zzz",
    num_rows=100,
)
```

csv_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *header: Optional[Sequence[str]] = None*, *data_columns: Tuple[str, ...] = ('{{name}}', '{{address}}')*, *num_rows: int = 10*, *include_row_ids: bool = False*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

csv_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *header: Optional[Sequence[str]] = None*, *data_columns: Tuple[str, ...] = ('{{name}}', '{{address}}')*, *num_rows: int = 10*, *include_row_ids: bool = False*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a CSV file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **header** – The `header` argument expects a list or a tuple of strings that will serve as the header row if supplied.
- **data_columns** – The `data_columns` argument expects a list or a tuple of string tokens, and these string tokens will be passed to `parse()` for data generation. Argument Groups are used to pass arguments to the provider methods. Both `header` and `data_columns` must be of the same length.
- **num_rows** – The `num_rows` argument controls how many rows of data to generate, and the `include_row_ids` argument may be set to `True` to include a sequential row ID column.
- **include_row_ids** –
- **content** – File content. If given, used as is.
- **encoding** – Encoding.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'csv'

15.15.1.1.3.60 faker_file.providers.docx_file module

class faker_file.providers.docx_file.**DocxFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

DOCX file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.docx_file import DocxFileProvider

FAKER = Faker()
FAKER.add_provider(DocxFileProvider)

file = FAKER.docx_file()
```

Usage example with options:

```
file = FAKER.docx_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.docx_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with content modifiers:

```
from faker_file.base import DynamicTemplate
from faker_file.contrib.docx_file import (
    add_h1_heading,
    add_h2_heading,
    add_h3_heading,
    add_h4_heading,
    add_h5_heading,
    add_h6_heading,
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
```

(continues on next page)

(continued from previous page)

```
    add_title_heading,
)

file = FAKER.docx_file(
    content=DynamicTemplate(
        [
            (add_title_heading, {}),
            (add_paragraph, {}),
            (add_h1_heading, {}),
            (add_h2_heading, {}),
            (add_h3_heading, {}),
            (add_h4_heading, {}),
            (add_h5_heading, {}),
            (add_h6_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_page_break, {}),
            (add_h6_heading, {}),
            (add_table, {}),
            (add_paragraph, {}),
        ]
    )
)
```

docx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

docx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a DOCX file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements (still being a string), which are then replaced by correspondent fixtures. Can alternatively be a *DynamicTemplate* - list of content modifiers (callables to call after the document instance has been created). Each callable should accept the following arguments: provider, document, data, counter and **kwargs.
- **format_func** – Callable responsible for formatting template strings.

- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'docx'

15.15.1.1.3.61 faker_file.providers.eml_file module

class faker_file.providers.eml_file.**EmlFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

EML file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.eml_file import EmlFileProvider

FAKER = Faker()
FAKER.add_provider(EmlFileProvider)

file = FAKER.eml_file()
```

Usage example with attachments:

```
from faker_file.providers.helpers.inner import create_inner_docx_file

file = FAKER.eml_file(
    prefix="zzz_email_",
    options={
        "count": 5,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "zzz_docx_file_",
            "max_nb_chars": 1_024,
        },
    }
)
```

Usage example of nested EMLs attachments:

```
from faker_file.providers.helpers.inner import create_inner_eml_file

file = FAKER.eml_file(
    options={
        "create_inner_file_func": create_inner_eml_file,
        "create_inner_file_args": {
            "options": {
                "create_inner_file_func": create_inner_docx_file,
            }
        }
    }
)
```

If you want to see, which files were included inside the EML, check the `file.data["files"]`.

eml_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *subject: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

eml_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *subject: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an EML file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **options** – Options (non-structured) for complex types, such as ZIP.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **subject** – Email subject. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'eml'

15.15.1.1.3.62 faker_file.providers.epub_file module

class faker_file.providers.epub_file.**EpubFileProvider**(*generator: Any*)

Bases: `BaseProvider`, *FileMixin*

EPUB file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.epub_file import EpubFileProvider

FAKER = Faker()
```

(continues on next page)

(continued from previous page)

```
FAKER.add_provider(EpubFileProvider)

file = FAKER.epub_file()
```

Usage example with options:

```
file = FAKER.epub_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.epub_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

epub_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *title: Optional[str] = None*, *chapter_title: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

epub_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *title: Optional[str] = None*, *chapter_title: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a EPUB file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **title** – E-book title. Might contain dynamic elements, which are then replaced by correspondent fixtures.

- **chapter_title** – Chapter title. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

extension: str = 'epub'

15.15.1.1.3.63 faker_file.providers.file_from_path module

class faker_file.providers.file_from_path.**FileFromPathProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

File from given path provider.

Usage example:

```
from faker import Faker
from faker_file.providers.file_from_path import (
    FileFromPathProvide
)

FAKER = Faker()
FAKER.add_provider(FileFromPathProvider)

file = FAKER.file_from_path(
    path="/path/to/file.pdf"
)
```

Usage example with options:

```
file = FAKER.file_from_path(
    path="/path/to/file.pdf",
    prefix="zzz",
)
```

extension: str = ''

file_from_path(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

file_from_path(*path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, ***kwargs*) → *StringValue*

File from given path.

Parameters

- **path** – Path to source file.
- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.

- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.64 faker_file.providers.generic_file module

class faker_file.providers.generic_file.**GenericFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

Generic file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.generic_file import GenericFileProvider

FAKER = Faker()
FAKER.add_provider(GenericFileProvider)

file = FAKER.generic_file(
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
)
```

Usage example with options:

```
file = FAKER.generic_file(
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
    prefix="zzz",
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.generic_file(
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
    basename="index",
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
)
```

Usage example with AWS S3 storage:

```
from faker_file.storages.aws_s3 import AWSS3Storage

file = FAKER.generic_file(
```

(continues on next page)

(continued from previous page)

```
    storage=AWSS3Storage(bucket_name="My-test-bucket"),
    content="<html><body><p>{{text}}</p></body></html>",
    extension="html",
)
```

extension: str = None

generic_file(*content: Union[bytes, str]*, *extension: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

generic_file(*content: Union[bytes, str]*, *extension: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a generic file with given content.

Parameters

- **content** – File content. If given, used as is.
- **extension** – File extension.
- **storage** – Storage class. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.65 faker_file.providers.gif_file module

class faker_file.providers.gif_file.**GifFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

GIF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.gif_file import GifFileProvider

FAKER = Faker()
FAKER.add_provider(GifFileProvider)

file = FAKER.gif_file()
```

Usage example with options:

```
file = FAKER.gif_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.gif_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'gif'

gif_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

gif_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a GIF file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.

- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'gif'

class faker_file.providers.gif_file.**GraphicGifFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic GIF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.gif_file import GraphicGifFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicGifFileProvider)

file = FAKER.graphic_gif_file()
```

Usage example with options:

```
file = FAKER.graphic_gif_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_gif_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'gif'

graphic_gif_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_gif_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic GIF file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'gif'

15.15.1.1.3.66 faker_file.providers.ico_file module

class faker_file.providers.ico_file.**GraphicIcoFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic ICO file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.ico_file import GraphicIcoFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicIcoFileProvider)

file = FAKER.graphic_ico_file()
```

Usage example with options:

```
file = FAKER.graphic_ico_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_ico_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'ico'

graphic_ico_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_ico_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic ICO file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'ico'

class faker_file.providers.ico_file.**IcoFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

ICO file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.ico_file import IcoFileProvider

FAKER = Faker()
FAKER.add_provider(IcoFileProvider)

file = FAKER.ico_file()
```

Usage example with options:

```
file = FAKER.ico_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.ico_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'ico'

ico_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

ico_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an ICO file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'ico'

15.15.1.1.3.67 faker_file.providers.jpeg_file module

class faker_file.providers.jpeg_file.**GraphicJpegFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic JPEG file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.jpeg_file import GraphicJpegFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicJpegFileProvider)

file = FAKER.graphic_jpeg_file()
```

Usage example with options:

```
file = FAKER.graphic_jpeg_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_jpeg_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'jpg'

graphic_jpeg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_jpeg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic JPEG file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.

- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'jpeg'

class faker_file.providers.jpeg_file.**JpegFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

JPEG file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.jpeg_file import JpegFileProvider

FAKER = Faker()
FAKER.add_provider(JpegFileProvider)

file = FAKER.jpeg_file()
```

Usage example with options:

```
file = FAKER.jpeg_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.jpeg_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'jpg'

image_format: str = 'jpeg'

jpeg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

jpeg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a JPEG file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.68 faker_file.providers.json_file module

class faker_file.providers.json_file.**JsonFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

JSON file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.json_file import JsonFileProvider

FAKER = Faker()
FAKER.add_provider(JsonFileProvider)

file = FAKER.json_file()
```

Usage example with options:

```
file = FAKER.json_file(
    prefix="zzz",
    num_rows=100,
    data_columns={"name": "{{name}}", "residency": "{{address}}"},
    indent=4,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.json_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    num_rows=100,
)
```

extension: str = 'json'

json_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[List] = None*, *num_rows: int = 10*, *indent: Optional[int] = None*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

json_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[List] = None*, *num_rows: int = 10*, *indent: Optional[int] = None*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a JSON file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **data_columns** – The `data_columns` argument expects a dict of string tokens, and these string tokens will be passed to `parse()` for data generation. Argument Groups are used to pass arguments to the provider methods.
- **num_rows** – The `num_rows` argument controls how many rows of data to generate, and the `include_row_ids` argument may be set to `True` to include a sequential row ID column.
- **indent** – Number of spaces to indent the fields.
- **content** – File content. If given, used as is.
- **encoding** – Encoding.
- **format_func** – Callable responsible for formatting template strings.

- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.69 faker_file.providers.odp_file module

class faker_file.providers.odp_file.**OdpFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

ODP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.odp_file import OdpFileProvider

FAKER = Faker()
FAKER.add_provider(OdpFileProvider)

file = FAKER.odp_file()
```

Usage example with options:

```
file = FAKER.odp_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.odp_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'odp'

odp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

odp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an ODP file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.70 faker_file.providers.ods_file module

class faker_file.providers.ods_file.**OdsFileProvider**(*generator: Any*)

Bases: BaseProvider, *TabularDataMixin*

ODS file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.ods_file import OdsFileProvider

FAKER = Faker()
FAKER.add_provider(OdsFileProvider)

file = FAKER.ods_file()
```

Usage example with options:

```
from faker import Faker
from faker_file.providers.ods_file import OdsFileProvider

file = FAKER.ods_file(
    prefix="zzz",
    num_rows=100,
    data_columns={
        "name": "{{name}}",
        "residency": "{{address}}",
```

(continues on next page)

(continued from previous page)

```
    },
    include_row_ids=True,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.ods_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    num_rows=100,
    data_columns={
        "name": "{{name}}",
        "residency": "{{address}}",
    },
    include_row_ids=True,
)
```

extension: str = 'ods'

ods_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

ods_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an ODS file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **data_columns** – The `data_columns` argument expects a list or a tuple of string tokens, and these string tokens will be passed to `pystr_format()` for data generation. Argument Groups are used to pass arguments to the provider methods. Both `header` and `data_columns` must be of the same length.
- **num_rows** – The `num_rows` argument controls how many rows of data to generate, and the `include_row_ids` argument may be set to `True` to include a sequential row ID column.
- **content** – List of dicts with content (JSON-like format). If given, used as is.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.71 faker_file.providers.odt_file module

class faker_file.providers.odt_file.**OdtFileProvider**(*generator: Any*)

Bases: `BaseProvider`, *FileMixin*

ODT file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.odt_file import OdtFileProvider

FAKER = Faker()
FAKER.add_provider(OdtFileProvider)

file = FAKER.odt_file()
```

Usage example with options:

```
file = FAKER.odt_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.odt_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with content modifiers:

```
from faker_file.base import DynamicTemplate
from faker_file.providers.jpeg_file import JpegFileProvider
from faker_file.contrib.odt_file import (
    add_h1_heading,
    add_h2_heading,
    add_h3_heading,
    add_h4_heading,
    add_h5_heading,
    add_h6_heading,
```

(continues on next page)

(continued from previous page)

```
    add_page_break,
    add_paragraph,
    add_picture,
    add_table,
)

file = FAKER.odt_file(
    content=DynamicTemplate(
        [
            (add_h1_heading, {}),
            (add_paragraph, {}),
            (add_h2_heading, {}),
            (add_h3_heading, {}),
            (add_h4_heading, {}),
            (add_h5_heading, {}),
            (add_h6_heading, {}),
            (add_paragraph, {}),
            (add_picture, {}),
            (add_page_break, {}),
            (add_h6_heading, {}),
            (add_table, {}),
            (add_paragraph, {}),
        ]
    )
)
```

extension: str = 'odt'

odt_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

odt_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[Union[str, DynamicTemplate]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an ODT file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.

- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.72 faker_file.providers.png_file module

class faker_file.providers.png_file.**GraphicPngFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic PNG file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.png_file import GraphicPngFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicPngFileProvider)

file = FAKER.graphic_png_file()
```

Usage example with options:

```
file = FAKER.graphic_png_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_png_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'png'

graphic_png_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_png_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic PNG file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'png'

class faker_file.providers.png_file.**PngFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

PNG file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.png_file import PngFileProvider

FAKER = Faker()
FAKER.add_provider(PngFileProvider)

file = FAKER.png_file()
```

Usage example with options:

```
file = FAKER.png_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.png_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
```

(continues on next page)

(continued from previous page)

```
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'png'

image_format: str = 'png'

png_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

png_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a PNG file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.73 faker_file.providers.pptx_file module

class faker_file.providers.pptx_file.**PptxFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

PPTX file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.pptx_file import PptxFileProvider

FAKER = Faker()
FAKER.add_provider(PptxFileProvider)

file = FAKER.pptx_file()
```

Usage example with options:

```
file = FAKER.pptx_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.pptx_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'pptx'

pptx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

pptx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.

- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.74 faker_file.providers.random_file_from_dir module

class faker_file.providers.random_file_from_dir.**RandomFileFromDirProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

Random file from given directory provider.

Usage example:

```
from faker import Faker
from faker_file.providers.random_file_from_dir import (
    RandomFileFromDirProvider,
)

FAKER = Faker()
FAKER.add_provider(RandomFileFromDirProvider)

file = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
)
```

Usage example with options:

```
file = FAKER.random_file_from_dir(
    source_dir_path="/tmp/tmp/",
    prefix="zzz",
)
```

extension: str = ''

random_file_from_dir(*source_dir_path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

random_file_from_dir(*source_dir_path: str*, *storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, ***kwargs*) → *StringValue*

Pick a random file from given directory.

Parameters

- **source_dir_path** – Source files directory.
- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.75 faker_file.providers.rtf_file module

class faker_file.providers.rtf_file.**RtfFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

RTF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.rtf_file import RtfFileProvider

FAKER = Faker()
FAKER.add_provider(RtfFileProvider)

file = FAKER.rtf_file()
```

Usage example with options:

```
file = FAKER.rtf_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.rtf_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'rtf'

rtf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

rtf_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a RTF file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.76 faker_file.providers.svg_file module

class faker_file.providers.svg_file.**SvgFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

SVG file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.svg_file import SvgFileProvider

FAKER = Faker()
FAKER.add_provider(SvgFileProvider)

file = FAKER.svg_file()
```

Usage example with options:

```
file = FAKER.svg_file(
    prefix="zzz",
    max_nb_chars=100_000,
```

(continues on next page)

(continued from previous page)

```
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.svg_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'svg'

image_format: str = 'svg'

svg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

svg_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an SVG file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.

- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns
Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.77 faker_file.providers.tar_file module

class faker_file.providers.tar_file.**TarFileProvider**(*generator: Any*)

Bases: `BaseProvider`, *FileMixin*

TAR file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.tar_file import TarFileProvider

FAKER = Faker()
FAKER.add_provider(TarFileProvider)

file = FAKER.tar_file()
```

Usage example with options:

```
from faker_file.providers.helpers.inner import create_inner_docx_file
from faker_file.providers.tar_file import TarFileProvider

file = FAKER.tar_file(
    prefix="ttt_archive_",
    options={
        "count": 5,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "ttt_docx_file_",
            "max_nb_chars": 1_024,
        },
        "directory": "ttt",
    },
)
```

Usage example of nested TARs:

```
from faker_file.providers.helpers.inner import create_inner_tar_file

file = FAKER.tar_file(
    options={
        "create_inner_file_func": create_inner_tar_file,
        "create_inner_file_args": {
            "options": {
                "create_inner_file_func": create_inner_docx_file,
            }
        }
```

(continues on next page)

(continued from previous page)

```
    },
)
```

If you want to see, which files were included inside the TAR, check the `file.data["files"]`.

extension: str = 'tar'

tar_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, *compression: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

tar_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, *compression: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a TAR file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **options** – Options (non-structured) for complex types, such as ZIP.
- **compression** – Desired compression. Can be None or *gz*, *bz2* or *xz*.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.78 faker_file.providers.tiff_file module

class faker_file.providers.tiff_file.**GraphicTiffFileProvider**(*generator: Any*)

Bases: `BaseProvider`, *GraphicImageMixin*

Graphic TIFF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.tiff_file import GraphicTiffFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicTiffFileProvider)

file = FAKER.graphic_tiff_file()
```

Usage example with options:

```
file = FAKER.graphic_tiff_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_tiff_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'tif'

graphic_tiff_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_tiff_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic TIFF file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'tiff'

class faker_file.providers.tiff_file.**TiffFileProvider**(*generator: Any*)

Bases: `BaseProvider`, *ImageMixin*

TIFF file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.tiff_file import TiffFileProvider
```

(continues on next page)

(continued from previous page)

```
FAKER = Faker()
FAKER.add_provider(TiffFileProvider)

file = FAKER.tiff_file()
```

Usage example with options:

```
file = FAKER.tiff_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.tiff_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'tif'

image_format: str = 'tiff'

tiff_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

tiff_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = WEASYPRINT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a TIFF file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.

- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.79 faker_file.providers.txt_file module

class faker_file.providers.txt_file.**TxtFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

TXT file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

file = FAKER.txt_file()
```

Usage example with options:

```
file = FAKER.txt_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.txt_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
```

(continues on next page)

(continued from previous page)

```
    wrap_chars_after=80,
)
```

extension: str = 'txt'

txt_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

txt_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_TEXT_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a TXT file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.80 faker_file.providers.webp_file module

class faker_file.providers.webp_file.**GraphicWebpFileProvider**(*generator: Any*)

Bases: BaseProvider, *GraphicImageMixin*

Graphic WEBP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.webp_file import GraphicWebpFileProvider

FAKER = Faker()
FAKER.add_provider(GraphicWebpFileProvider)

file = FAKER.graphic_webp_file()
```

Usage example with options:

```
file = FAKER.graphic_webp_file(
    prefix="zzz",
    size=(800, 800),
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.graphic_webp_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    basename="yyy",
    size=(1024, 1024),
)
```

extension: str = 'webp'

graphic_webp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

graphic_webp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *size: Tuple[int, int] = (256, 256)*, *hue: Optional[Union[int, Sequence[int], str]] = None*, *luminosity: Optional[str] = None*, ***kwargs*) → *StringValue*

Generate a graphic WEBP file with random lines.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **size** – Image size in pixels.
- **hue** – Read more about ://faker.readthedocs.io/en/dev/providers/faker.providers.color.html
- **luminosity** – If given, the output string would be separated by line breaks after the given position.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

image_format: str = 'webp'

class faker_file.providers.webp_file.**WebpFileProvider**(*generator: Any*)

Bases: BaseProvider, *ImageMixin*

WEBP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.webp_file import WebpFileProvider

FAKER = Faker()
FAKER.add_provider(WebpFileProvider)

file = FAKER.webp_file()
```

Usage example with options:

```
file = FAKER.webp_file(
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.webp_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    max_nb_chars=100_000,
    wrap_chars_after=80,
)
```

extension: str = 'webp'

image_format: str = 'webp'

webp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

webp_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *max_nb_chars: int = DEFAULT_IMAGE_MAX_NB_CHARS*, *wrap_chars_after: Optional[int] = None*, *content: Optional[str] = None*, *image_generator_cls: Optional[Union[str, Type[BaseImageGenerator]]] = DEFAULT_IMAGE_GENERATOR*, *image_generator_kwargs: Optional[Dict[str, Any]] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a WEBP file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).

- **prefix** – File name prefix.
- **max_nb_chars** – Max number of chars for the content.
- **wrap_chars_after** – If given, the output string would be separated by line breaks after the given position.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **image_generator_cls** – Image generator class.
- **image_generator_kwargs** – Image generator kwargs.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.81 faker_file.providers.xlsx_file module

class faker_file.providers.xlsx_file.**XlsxFileProvider**(*generator: Any*)

Bases: BaseProvider, *TabularDataMixin*

XLSX file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.xlsx_file import XlsxFileProvider

FAKER = Faker()
FAKER.add_provider(XlsxFileProvider)

file = FAKER.xlsx_file()
```

Usage example with options:

```
file = FAKER.xlsx_file(
    prefix="zzz",
    num_rows=100,
    data_columns={
        "name": "{{name}}",
        "residency": "{{address}}",
    },
    include_row_ids=True,
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.xlsx_file(
```

(continues on next page)

(continued from previous page)

```
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    num_rows=100,
    data_columns={
        "name": "{{name}}",
        "residency": "{{address}}",
    },
    include_row_ids=True,
)
```

extension: str = 'xlsx'

xlsx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

xlsx_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate a XLSX file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **data_columns** – The `data_columns` argument expects a list or a tuple of string tokens, and these string tokens will be passed to `pystr_format()` for data generation. Argument Groups are used to pass arguments to the provider methods. Both `header` and `data_columns` must be of the same length.
- **num_rows** – The `num_rows` argument controls how many rows of data to generate, and the `include_row_ids` argument may be set to `True` to include a sequential row ID column.
- **content** – List of dicts with content (JSON-like format). If given, used as is.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.82 faker_file.providers.xml_file module

class faker_file.providers.xml_file.**XmlFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

XML file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.xml_file import XmlFileProvider

FAKER = Faker()
FAKER.add_provider(XmlFileProvider)

file = FAKER.xml_file()
```

Usage example with options:

```
file = FAKER.xml_file(
    prefix="zzz",
    num_rows=100,
    data_columns={
        "name": "{{name}}",
        "sentence": "{{sentence}}",
        "address": "{{address}}",
    },
)
```

Usage example with *FileSystemStorage* storage (for *Django*):

```
from django.conf import settings
from faker_file.storages.filesystem import FileSystemStorage

file = FAKER.xml_file(
    storage=FileSystemStorage(
        root_path=settings.MEDIA_ROOT,
        rel_path="tmp",
    ),
    prefix="zzz",
    num_rows=100,
)
```

Usage example with template:

```
XML_TEMPLATE = '''
<books>
    <book>
        <name>{{sentence}}</name>
        <description>{{paragraph}}</description>
        <isbn>{{isbn13}}</isbn>
    </book>
    <book>
        <name>{{sentence}}</name>
```

(continues on next page)

(continued from previous page)

```
        <description>{{paragraph}}</description>
        <isbn>{{isbn13}}</isbn>
    </book>
    <book>
        <name>{{sentence}}</name>
        <description>{{paragraph}}</description>
        <isbn>{{isbn13}}</isbn>
    </book>
</books>
'''

file = FAKER.xml_file(content=XML_TEMPLATE)
```

extension: str = 'xml'

xml_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *root_element: str = 'root'*, *row_element: str = 'row'*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, *raw: bool = True*, ***kwargs*) → *BytesValue*

xml_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *root_element: str = 'root'*, *row_element: str = 'row'*, *data_columns: Optional[Dict[str, str]] = None*, *num_rows: int = 10*, *content: Optional[str] = None*, *encoding: Optional[str] = None*, *format_func: Callable[[Union[Faker, Generator, Provider], str], str] = DEFAULT_FORMAT_FUNC*, ***kwargs*) → *StringValue*

Generate an XML file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **root_element** – Root XML element.
- **row_element** – Row XML element.
- **data_columns** – The `data_columns` argument expects a list or a tuple of string tokens, and these string tokens will be passed to `pystr_format()` for data generation. Argument Groups are used to pass arguments to the provider methods. Both `header` and `data_columns` must be of the same length.
- **num_rows** – The `num_rows` argument controls how many rows of data to generate, and the `include_row_ids` argument may be set to `True` to include a sequential row ID column.
- **content** – File content. Might contain dynamic elements, which are then replaced by correspondent fixtures.
- **encoding** – Encoding.
- **format_func** – Callable responsible for formatting template strings.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.83 faker_file.providers.zip_file module

class faker_file.providers.zip_file.**ZipFileProvider**(*generator: Any*)

Bases: BaseProvider, *FileMixin*

ZIP file provider.

Usage example:

```
from faker import Faker
from faker_file.providers.zip_file import ZipFileProvider

FAKER = Faker()
FAKER.add_provider(ZipFileProvider)

file = FAKER.zip_file()
```

Usage example with options:

```
from faker_file.providers.helpers.inner import create_inner_docx_file

file = FAKER.zip_file(
    prefix="zzz_archive_",
    options={
        "count": 5,
        "create_inner_file_func": create_inner_docx_file,
        "create_inner_file_args": {
            "prefix": "zzz_docx_file_",
            "max_nb_chars": 1_024,
        },
        "directory": "zzz",
    },
)
```

Usage example of nested ZIPs:

```
from faker_file.providers.helpers.inner import create_inner_zip_file

file = FAKER.zip_file(
    options={
        "create_inner_file_func": create_inner_zip_file,
        "create_inner_file_args": {
            "options": {
                "create_inner_file_func": create_inner_docx_file,
            },
        },
    },
)
```

If you want to see, which files were included inside the ZIP, check the `file.data["files"]`.

extension: str = 'zip'

zip_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, *raw: bool = True*, ***kwargs*) → *BytesValue*

zip_file(*storage: Optional[BaseStorage] = None*, *basename: Optional[str] = None*, *prefix: Optional[str] = None*, *options: Optional[Dict[str, Any]] = None*, ***kwargs*) → Union[*BytesValue*, *StringValue*]

Generate a ZIP file with random text.

Parameters

- **storage** – Storage. Defaults to *FileSystemStorage*.
- **basename** – File basename (without extension).
- **prefix** – File name prefix.
- **options** – Options (non-structured) for complex types, such as ZIP.
- **raw** – If set to True, return *BytesValue* (binary content of the file). Otherwise, return *StringValue* (path to the saved file).

Returns

Relative path (from root directory) of the generated file or raw content of the file.

15.15.1.1.3.84 Module contents

15.15.1.1.4 faker_file.storages package

15.15.1.1.4.1 Submodules

15.15.1.1.4.2 faker_file.storages.aws_s3 module

class faker_file.storages.aws_s3.**AWSS3Storage**(*bucket_name: str*, *root_path: Optional[str] = 'tmp'*, *rel_path: Optional[str] = 'tmp'*, *credentials: Optional[Dict[str, Any]] = None*, **args*, ***kwargs*)

Bases: *CloudStorage*

AWS S3 Storage.

Usage example:

```
from faker_file.storages.aws_s3 import AWSS3Storage

s3_storage = AWSS3Storage(
    bucket_name="artur-testing-1",
    rel_path="tmp",
)
file = s3_storage.generate_filename(prefix="zzz_", extension="docx")
s3_storage.write_text(file, "Lorem ipsum")
s3_storage.write_bytes(file, b"Lorem ipsum")
```

authenticate(*key_id: str*, *key_secret: str*, ***kwargs*) → None

Authenticate to AWS S3.

schema: str = 's3'

15.15.1.1.4.3 faker_file.storages.azure_cloud_storage module

class faker_file.storages.azure_cloud_storage.**AzureCloudStorage**(*bucket_name: str*, *root_path: Optional[str] = 'tmp'*, *rel_path: Optional[str] = 'tmp'*, *credentials: Optional[Dict[str, Any]] = None*, **args*, ***kwargs*)

Bases: *CloudStorage*

Azure Cloud Storage.

Usage example:

```
from faker_file.storages.azure_cloud_storage import AzureCloudStorage

azure_storage = AzureCloudStorage(
    bucket_name="artur-testing-1",
    rel_path="tmp",
)
file = azure_storage.generate_filename(prefix="zzz_", extension="docx")
azure_storage.write_text(file, "Lorem ipsum")
azure_storage.write_bytes(file, b"Lorem ipsum")
```

authenticate(*connection_string: str*, ***kwargs*) → None

Authenticate to Azure Cloud Storage.

schema: Optional[str] = 'azure'

15.15.1.1.4.4 faker_file.storages.base module

class faker_file.storages.base.**BaseStorage**(**args*, ***kwargs*)

Bases: `object`

Base storage.

abspath(*filename: Any*) → str

Return absolute path.

exists(*filename: Any*) → bool

Check if file exists.

generate_filename(*extension: str*, *prefix: Optional[str] = None*, *basename: Optional[str] = None*) → Any

Generate filename.

relpath(*filename: Any*) → str

Return relative path.

unlink(*filename: Any*) → None

Delete the file.

write_bytes(*filename: Any*, *data: bytes*) → int

Write bytes.

write_text(*filename: Any*, *data: str*, *encoding: Optional[str] = None*) → int

Write text.

15.15.1.1.4.5 faker_file.storages.cloud module

class faker_file.storages.cloud.**CloudStorage**(*bucket_name: str*, *root_path: Optional[str] = 'tmp'*, *rel_path: Optional[str] = 'tmp'*, *credentials: Optional[Dict[str, Any]] = None*, **args*, ***kwargs*)

Bases: *BaseStorage*

Base cloud storage.

abspath(*filename: Pathy*) → str

Return relative path.

abstract authenticate(**args*, ***kwargs*)

bucket: Pathy

bucket_name: str

credentials: Dict[str, str]

exists(*filename: Union[Pathy, str]*) → bool

Check if file exists.

generate_filename(*extension: str*, *prefix: Optional[str] = None*, *basename: Optional[str] = None*) → Pathy

Generate filename.

relpath(*filename: Pathy*) → str

Return relative path.

schema: Optional[str] = None

unlink(*filename: Union[Pathy, str]*) → None

Delete the file.

write_bytes(*filename: Pathy*, *data: bytes*) → int

Write bytes.

write_text(*filename: Pathy*, *data: str*, *encoding: Optional[str] = None*) → int

Write text.

class faker_file.storages.cloud.**PathyFileSystemStorage**(*bucket_name: str*, *root_path: Optional[str] = 'tmp'*, *rel_path: Optional[str] = 'tmp'*, *credentials: Optional[Dict[str, Any]] = None*, **args*, ***kwargs*)

Bases: *CloudStorage*

Pathy FileSystem Storage.

Usage example:

```
from faker_file.storages.cloud import PathyFileSystemStorage

fs_storage = PathyFileSystemStorage(bucket_name="artur-testing-1")
file = fs_storage.generate_filename(prefix="zzz_", extension="docx")
fs_storage.write_text(file, "Lorem ipsum")
fs_storage.write_bytes(file, b"Lorem ipsum")
```

authenticate(***kwargs*) → None

Authenticate. Does nothing.

schema: str = 'file'

15.15.1.1.4.6 faker_file.storages.filesystem module

class faker_file.storages.filesystem.**FileSystemStorage**(*root_path: Optional[str] = '/tmp'*, *rel_path: Optional[str] = 'tmp'*, **args*, ***kwargs*)

Bases: *BaseStorage*

File storage.

Usage example:

```
from faker_file.storages.filesystem import FileSystemStorage

storage = FileSystemStorage()
file = storage.generate_filename(prefix="zzz_", extension="docx")
storage.write_text(file, "Lorem ipsum")
storage.write_bytes(file, b"Lorem ipsum")
```

Initialization with params:

```
storage = FileSystemStorage()
```

abspath(*filename: str*) → str

Return absolute path.

exists(*filename: str*) → bool

Check if file exists.

generate_filename(*extension: str*, *prefix: Optional[str] = None*, *basename: Optional[str] = None*) → str

Generate filename.

relpath(*filename: str*) → str

Return relative path.

unlink(*filename: str*) → None

Delete the file.

write_bytes(*filename: str*, *data: bytes*) → int

Write bytes.

write_text(*filename: str*, *data: str*, *encoding: Optional[str] = None*) → int

Write text.

15.15.1.1.4.7 faker_file.storages.google_cloud_storage module

class faker_file.storages.google_cloud_storage.**GoogleCloudStorage**(*bucket_name: str*, *root_path: Optional[str] = 'tmp'*, *rel_path: Optional[str] = 'tmp'*, *credentials: Optional[Dict[str, Any]] = None*, **args*, ***kwargs*)

Bases: *CloudStorage*

Google Cloud Storage.

Usage example:

```
from faker_file.storages.google_cloud_storage import GoogleCloudStorage

gs_storage = GoogleCloudStorage(
    bucket_name="artur-testing-1",
    rel_path="tmp",
)
file = gs_storage.generate_filename(prefix="zzz_", extension="docx")
gs_storage.write_text(file, "Lorem ipsum")
gs_storage.write_bytes(file, b"Lorem ipsum")
```

authenticate(*json_file_path: str*, ***kwargs*) → None

Authenticate to Google Cloud Storage.

schema: Optional[str] = 'gs'

15.15.1.1.4.8 faker_file.storages.sftp_storage module

class faker_file.storages.sftp_storage.**SFTPStorage**(*host: str*, *port: int = 22*, *username: str = ''*, *password: Optional[str] = None*, *key: Optional[PKey] = None*, *root_path: str = ''*, *rel_path: str = ''*, **args*, ***kwargs*)

Bases: *BaseStorage*

SFTP storage.

Usage example:

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider
from faker_file.storages.sftp_storage import SFTPStorage

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

# SFTP storage class
STORAGE = SFTPStorage(
    host="0.0.0.0",
    username="foo",
    password="pass",
)
```

(continues on next page)

(continued from previous page)

```
# Generate TXT file in the default directory
txt_file = FAKER.txt_file(storage=STORAGE)

# Another SFTP storage class, but inside a `/upload/another` directory
STORAGE_SUB_DIR = SFTPStorage(
    host="0.0.0.0",
    username="foo",
    password="pass",
    root_path="/upload/another",
)

# Generate TXT file inside `/upload/another` directory
txt_file = FAKER.txt_file(storage=STORAGE_SUB_DIR)
```

abspath(*filename: str*) → str

Return absolute path.

close()

Explicitly close the connection.

exists(*filename: str*) → bool

Check if file exists.

generate_filename(*extension: str*, *prefix: Optional[str] = None*, *basename: Optional[str] = None*) → str

Generate filename.

relpath(*filename: str*) → str

Return relative path.

sftp: Optional[SFTPClient] = None

transport: Optional[Transport] = None

unlink(*filename: str*) → None

Remove a file.

write_bytes(*filename: str*, *data: bytes*) → int

Write bytes.

write_text(*filename: str*, *data: str*, *encoding: Optional[str] = 'utf-8'*) → int

Write text.

15.15.1.1.4.9 Module contents

15.15.1.1.5 faker_file.tests package

15.15.1.1.5.1 Submodules

15.15.1.1.5.2 faker_file.tests.data module

15.15.1.1.5.3 faker_file.tests.sftp_server module

class faker_file.tests.sftp_server.**SFTPServer**(*conn: SSHServerChannel*)

Bases: SFTPServer

class faker_file.tests.sftp_server.**SFTPServerManager**(*host: str = '0.0.0.0'*, *port: int = 2222*)

Bases: object

start() → None

async start_server() → None

stop() → None

class faker_file.tests.sftp_server.**SSHServer**(*connection_semaphore: Semaphore*)

Bases: SSHServer

auth_completed() → None

Authentication was completed successfully

This method is called when authentication has completed succesfully. Applications may use this method to perform processing based on the authenticated username or options in the authorized keys list or certificate associated with the user before any sessions are opened or forwarding requests are handled.

async begin_auth(*username: str*) → bool

Authentication has been requested by the client

This method will be called when authentication is attempted for the specified user. Applications should use this method to prepare whatever state they need to complete the authentication, such as loading in the set of authorized keys for that user. If no authentication is required for this user, this method should return *False* to cause the authentication to immediately succeed. Otherwise, it should return *True* to indicate that authentication should proceed.

If blocking operations need to be performed to prepare the state needed to complete the authentication, this method may be defined as a coroutine.

Parameters
username (*str*) – The name of the user being authenticated

Returns
A *bool* indicating whether authentication is required

password_auth_supported() → bool

Return whether or not password authentication is supported

This method should return *True* if password authentication is supported. Applications wishing to support it must have this method return *True* and implement *validate_password()* to return whether or not the password provided by the client is valid for the user being authenticated.

By default, this method returns *False* indicating that password authentication is not supported.

Returns
A *bool* indicating if password authentication is supported or not

session_requested() → bool

Handle an incoming session request

This method is called when a session open request is received from the client, indicating it wishes to open a channel to be used for running a shell, executing a command, or connecting to a subsystem. If the application wishes to accept the session, it must override this method to return either an SSHServerSession object to use to process the data received on the channel or a tuple consisting of an SSHServerChannel object created with create_server_channel and an SSHServerSession, if the application wishes to pass non-default arguments when creating the channel.

If blocking operations need to be performed before the session can be created, a coroutine which returns an `SSHServerSession` object can be returned instead of the session iself. This can be either returned directly or as a part of a tuple with an `SSHServerChannel` object.

To reject this request, this method should return *False* to send back a "Session refused" response or raise a `ChannelOpenError` exception with the reason for the failure.

The details of what type of session the client wants to start will be delivered to methods on the `SSHServerSession` object which is returned, along with other information such as environment variables, terminal type, size, and modes.

By default, all session requests are rejected.

Returns

One of the following:

- An `SSHServerSession` object or a coroutine which returns an `SSHServerSession`
- A tuple consisting of an `SSHServerChannel` and the above
- A *callable* or coroutine handler function which takes AsyncSSH stream objects for stdin, stdout, and stderr as arguments
- A tuple consisting of an `SSHServerChannel` and the above
- *False* to refuse the request

Raises

`ChannelOpenError` if the session shouldn't be accepted

sftp_requested() → Type[*SFTPServer*]

validate_password(*username: str*, *password: str*) → bool

Return whether password is valid for this user

This method should return *True* if the specified password is a valid password for the user being authenticated. It must be overridden by applications wishing to support password authentication.

If the password provided is valid but expired, this method may raise `PasswordChangeRequired` to request that the client provide a new password before authentication is allowed to complete. In this case, the application must override `change_password()` to handle the password change request.

This method may be called multiple times with different passwords provided by the client. Applications may wish to limit the number of attempts which are allowed. This can be done by having *password_auth_supported()* begin returning *False* after the maximum number of attempts is exceeded.

If blocking operations need to be performed to determine the validity of the password, this method may be defined as a coroutine.

By default, this method returns *False* for all passwords.

Parameters

- **username** (*str*) – The user being authenticated
- **password** (*str*) – The password sent by the client

Returns

A *bool* indicating if the specified password is valid for the user being authenticated

Raises

`PasswordChangeRequired` if the password provided is expired and needs to be changed

faker_file.tests.sftp_server.**start_server**(*host: str = '0.0.0.0'*, *port: int = 2222*) → None

async faker_file.tests.sftp_server.**start_server_async**(*host: str = '0.0.0.0'*, *port: int = 2222*) → None

15.15.1.1.5.4 faker_file.tests.test_augment module

class faker_file.tests.test_augment.**TestSolarizeFunction**(*methodName='runTest'*)

Bases: TestCase

Test *solarize* function.

create_rgb_image()

Create a sample RGB image.

create_rgba_image()

Create a sample RGBA image.

test_solarize_rgb()

test_solarize_rgba()

15.15.1.1.5.5 faker_file.tests.test_augment_file_from_dir_provider module

15.15.1.1.5.6 faker_file.tests.test_base module

class faker_file.tests.test_base.**StringListTestCase**(*methodName='runTest'*)

Bases: TestCase

StringList test case.

test_string_list() → None

15.15.1.1.5.7 faker_file.tests.test_cli module

class faker_file.tests.test_cli.**TestCLI**(*methodName='runTest'*)

Bases: TestCase

CLI tests.

tearDown() → None

Hook method for deconstructing the test fixture after testing it.

test_broken_imports() → None

Test broken imports.

test_cli

test_cli_error_no_provider() → None

Test CLI, no provider given.

test_cli_generate_completion() → None

Test CLI, generate-completion.

test_cli_version() → None
Test CLI, version.

15.15.1.1.5.8 faker_file.tests.test_data_integrity module

15.15.1.1.5.9 faker_file.tests.test_django_integration module

class faker_file.tests.test_django_integration.**DjangoIntegrationTestCase**(*methodName='runTest'*)
Bases: TestCase

Django integration test case.

FAKER: Faker

tearDown(**args*, ***kwargs*) → None
Hook method for deconstructing the test fixture after testing it.

test_file

15.15.1.1.5.10 faker_file.tests.test_helpers module

class faker_file.tests.test_helpers.**HelpersTestCase**(*methodName='runTest'*)
Bases: TestCase

Test StringList test case.

test_random_pop() → None
Test *random_pop*.

test_random_pop_empty_list() → None
Test *random_pop*.

15.15.1.1.5.11 faker_file.tests.test_providers module

15.15.1.1.5.12 faker_file.tests.test_registry module

class faker_file.tests.test_registry.**RegistryTestCase**(*methodName='runTest'*)
Bases: TestCase

Test *registry* module.

test_clean_up_exceptions()

test_integration() → None
Test *add*.

test_remove_by_string_not_found()

test_remove_exceptions()

15.15.1.1.5.13 faker_file.tests.test_sftp_server module

class faker_file.tests.test_sftp_server.**TestSFTPServerWithManager**(*methodName='runTest'*)

Bases: IsolatedAsyncioTestCase, __TestSFTPServerMixin

manager: *SFTPServerManager*

manager_thread: Thread

classmethod **setUpClass**()

Hook method for setting up class fixture before running tests in the class.

sftp_host: str = '127.0.0.1'

sftp_pass: str = 'pass'

sftp_port: int = 2645

sftp_user: str = 'foo'

classmethod **tearDownClass**()

Hook method for deconstructing the class fixture after running all tests in the class.

class faker_file.tests.test_sftp_server.**TestSFTPServerWithStartServer**(*methodName='runTest'*)

Bases: IsolatedAsyncioTestCase, __TestSFTPServerMixin

classmethod **setUpClass**()

Hook method for setting up class fixture before running tests in the class.

sftp_host: str = '127.0.0.1'

sftp_pass: str = 'pass'

sftp_port: int = 4367

sftp_user: str = 'foo'

classmethod **tearDownClass**()

Hook method for deconstructing the class fixture after running all tests in the class.

class faker_file.tests.test_sftp_server.**TestSFTPServerWithStartServerAsync**(*methodName='runTest'*)

Bases: IsolatedAsyncioTestCase, __TestSFTPServerMixin

classmethod **setUpClass**()

Hook method for setting up class fixture before running tests in the class.

sftp_host: str = '127.0.0.1'

sftp_pass: str = 'pass'

sftp_port: int = 3477

sftp_user: str = 'foo'

classmethod **tearDownClass**()

Hook method for deconstructing the class fixture after running all tests in the class.

15.15.1.1.5.14 faker_file.tests.test_sftp_storage module

class faker_file.tests.test_sftp_storage.**TestSFTPStorageTestCase**(*methodName='runTest'*)

Bases: TestCase

Test SFTP storage.

classmethod free_port() → None

Check if the port is in use and wait until it is free.

static is_port_in_use(*host: str*, *port: int*) → bool

max_port_retry_limit: int = 10

server_manager: *SFTPServerManager*

server_thread: Thread

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

sftp_host: str = '127.0.0.1'

sftp_pass: str = 'pass'

sftp_port: int = 4877

sftp_root_path: str = '/upload'

sftp_user: str = 'foo'

tearDown() → None

Hook method for deconstructing the test fixture after testing it.

test_file_system_storage_abspath() → None

Test *FileSystemStorage abspath*.

test_integration() → None

test_integration_sub_dir() → None

test_storage

test_storage_exists_exceptions() → None

test_storage_generate_filename_exceptions

test_storage_initialization_exceptions

test_storage_write_bytes_exceptions() → None

test_storage_write_text_exceptions() → None

15.15.1.1.5.15 faker_file.tests.test_sqlalchemy_integration module

15.15.1.1.5.16 faker_file.tests.test_storages module

class faker_file.tests.test_storages.**TestStoragesTestCase**(*methodName='runTest'*)

Bases: TestCase

Test storages.

setUp() → None

Hook method for setting up the test fixture before exercising it.

tearDown() → None

Hook method for deconstructing the test fixture after testing it.

test_base_storage_exceptions

test_cloud_storage_exceptions

test_file_system_storage_abspath() → None

Test *FileSystemStorage abspath*.

test_pathy_file_system_storage_abspath() → None

Test *PathyFileSystemStorage abspath*.

test_pathy_file_system_storage_unlink() → None

Test *PathyFileSystemStorage unlink*.

test_storage

test_storage_generate_filename_exceptions

test_storage_initialization_exceptions

15.15.1.1.5.17 faker_file.tests.texts module

15.15.1.1.5.18 faker_file.tests.utils module

class faker_file.tests.utils.**AutoFreePortInt**(*min_port: int = 2223*, *max_port: int = 5000*, *host: str = 'localhost'*, **args*, ***kwargs*)

Bases: int

Automatically and randomly picks a free port within a specified range.

For instance:

Random free port between default range 2223 and 5000 port = AutoFreePortInt()

Random free port between 3000 and 4000 port = AutoFreePortInt(min_port=3000, max_port=4000)

For the rest, it behaves like a normal integer.

For better integration, it's recommended to cast the value to *int*, like this:

port = int(AutoFreePortInt())

DEFAULT_MAX_PORT: int = 5000

DEFAULT_MIN_PORT: int = 2223

class faker_file.tests.utils.**AutoIncPortInt**(**args*, ***kwargs*)

Bases: int

Automatically incremented integer value.

Contains state of issued values. Starts from 2223. Every time initialized, value increases.

Usage example:

port = AutoInt() # 2223 port = AutoInt() # 2224 port = AutoInt() # 2225

For the rest, it behaves like a normal integer.

For better integration, it's recommended to cast the value to *int*, like this:

port = int(AutoInt())

15.15.1.1.5.19 Module contents

15.15.1.2 Submodules

15.15.1.3 faker_file.base module

class faker_file.base.**BytesValue**(*value*, **args*, ***kwargs*)

Bases: bytes

data: Dict[str, Any]

faker_file.base.**DEFAULT_FORMAT_FUNC**(*generator: Union[Faker, Generator, Provider]*, *content: str*) → str

class faker_file.base.**DynamicTemplate**(*content_modifiers: List[Tuple[Callable, Dict[str, Any]]]*)

Bases: object

Dynamic template.

class faker_file.base.**FileMixin**

Bases: object

File mixin.

extension: str

formats: List[str]

generator: Union[Faker, Generator, Provider]

numerify: Callable

random_element: Callable

class faker_file.base.**StringList**(*strings: Optional[List[str]] = None*, *separator: str = ' '*)

Bases: object

String list.

Usage example:

```
my_string = StringList(separator="
```

“)

my_string += “grape” my_string += “peaches” print(my_string)

add_string(*value: str*) → None

remove_string(*value: str*) → None

class faker_file.base.**StringValue**(*value*, **args*, ***kwargs*)

Bases: `str`

data: Dict[str, Any]

faker_file.base.**parse_format_func**(*generator: Union[Faker, Generator, Provider]*, *content: str*) → str

faker_file.base.**pystr_format_func**(*generator: Union[Faker, Generator, Provider]*, *content: str*) → str

faker_file.base.**returns_list**(*func: Callable*) → bool

Checks if callable returns a list of Union[BytesValue, StringValue].

Returns True if it’s a List. Returns False otherwise.

15.15.1.4 faker_file.constants module

15.15.1.5 faker_file.helpers module

faker_file.helpers.**load_class_from_path**(*full_path: str*) → Type

Load a class from a given full path string identifier.

Parameters

full_path – The full path to the class, e.g. ‘module.submodule.MyClass’.

Returns

The loaded class.

Raise

If the module cannot be found or the class does not exist in the module, it raises ImportError.

Usage example:

```
my_class = load_class_from_path("module.submodule.MyClass")
instance = my_class()
```

faker_file.helpers.**random_pop**(*lst: list*) → Any

Randomly pops element from the given list. Alters the list.

Parameters

lst – List to pop element from.

Returns

A single element from the list.

Usage example:

```
from faker_file.helpers import random_pop
my_list = [1, 2, 3, 4, 5]
element = random_pop(my_list)
```

faker_file.helpers.**wrap_text**(*text: str*, *wrap_chars_after: int*) → str

15.15.1.6 faker_file.registry module

class faker_file.registry.**FileRegistry**

Bases: object

Stores list *StringValue* instances.

```
from faker import Faker
from faker_file.providers.txt_file import TxtFileProvider

FAKER = Faker()
FAKER.add_provider(TxtFileProvider)

txt_file_1 = FAKER.txt_file()
txt_file_2 = FAKER.txt_file()
...
txt_file_n = FAKER.txt_file()

# The FileRegistry._registry would then contain this:
{
    txt_file_1,
    txt_file_2,
    ...,
    txt_file_n,
}
```

add(*string_value:* StringValue) → None

clean_up() → None

remove(*string_value: Union[*StringValue*, str]*) → bool

search(*value: str*) → Optional[*StringValue*]

15.15.1.7 Module contents

15.16 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

A

B

C

D

E

F

G

H

N

O

P

R

S

T

U

V

W

X

Z