namefiles

Release 0.5.1

David Scheliga

Jul 27, 2021

CONTENTS

Name-files is an approach for a standardized file naming for multiple files of different sources with equal formatting, which all are related to the same entity.

CHAPTER

ONE

INSTALLATION

Install the latest release from pip.

```
$ pip install namefiles
```

1.1 Basic Usage

At the current implementation *namefiles* revolves around getting filenames within python scripts, which comply to a file naming convention.

A first use case is using a source filename for a new filename. You might have a source file which is used for a process resulting into a new file, for which a related name is required.

By quickly setting the fresh filename parts a new path can be obtained.

```
>>> from namefiles import FilenameParts
    >>> source_filename = NameGiver.disassemble("/root/path/A#file.txt")
    >>> target_filename = source_filename.with_parts(
    ...     sub_id="NEW", source_id="filename"
    ... )
    >>> target_filename.to_path()
    PosixPath('/root/path/A.txt')


Another use case is using metadata already carrying the filename parts.
>>> source_filename = NameGiver.disassemble("/root/path/A#file.txt")
>>> target_filename = source_filename.with_parts(
...     sub_id="NEW", source_id="filename"
... )
>>> target_filename.to_path()
PosixPath('/root/path/A.txt')
```

Another use case is using metadata already carrying the filename parts.

```
>>> sample_metadata = {
    ...     "identifier": "A",
    ...     "sub_id": "FILE",
    ...     "context": "name",
    ...     "non-filename-field": "Is not for the filename."
    ... }
    >>> from namefiles import FilenameParts
```

(continues on next page)

(continued from previous page)

```
    >>> new_filepath_giver = NameGiver(
    ...     root_path="/root/path", extension=".txt", **sample_metadata
    ... )
    >>> str(new_filepath)
    '/root/path/A#FILE.name.txt'
...     "identifier": "A",
...     "sub_id": "FILE",
...     "context": "name",
...     "non-filename-field": "Is not for the filename."
... }
>>> from namefiles import NameGiver
>>> new_filepath_giver = NameGiver(
...     root_path="/root/path", extension=".txt", **sample_metadata
... )
>>> str(new_filepath)
'/root/path/A#FILE.name.txt'
```

1.2 API reference

namefiles	
namefiles.disassemble_filename(target_path)	Disassembles a file's name into the parts defined by a file naming convention.
namefiles.construct_filename([...])	Constructs a filename using a filename convention.
namefiles.construct_filepath([...])	Constructs a filepath using a file naming convention.
namefiles.extract_filename_parts(...[, ...])	Extracts filename parts from a dictionary based by a file naming convention.
namefiles.get_filename_convention([...])	Gets the currently defined file naming convention.
namefiles.get_filename_validator([...])	Returns a filename validator for applying the file naming convention.
namefiles.is_a_filename_part(part_name[, ...])	Returns if the part name is within the file naming convention.
namefiles.register_filename_validator(...)	Registers file naming convention.

1.2.1 namefiles

Module Attributes

JsonschemaValidator	A Validator implementing the jsonschema.IValidator interface.

Functions

construct_filename([filename_template, ...])	Constructs a filename using a filename convention.
construct_filepath([filename_template, ...])	Constructs a filepath using a file naming convention.
disassemble_filename(target_path[, ...])	Disassembles a file's name into the parts defined by a file naming convention.
extract_filename_parts(potential_filename_parts)	Extracts filename parts from a dictionary based by a file naming convention.
get_filename_convention([convention_name])	Gets the currently defined file naming convention.
get_filename_validator([convention_name])	Returns a filename validator for applying the file naming convention.
is_a_filename_part(part_name[, ...])	Returns if the part name is within the file naming convention.
register_filename_validator(filename_validator)	Registers file naming convention.

Classes

ANameGiver(**filename_parts)	*A Name Giver* is the abstract base class, which can be used to define a custom file naming convention.
ExtractsFilenamePart()	
FilenameParts(identifier, sub_id, source_id, ...)	The *filename parts* implements the current standard file naming convention.
NameGiver(identifier, sub_id, source_id, ...)	The *Name Giver* implements the current standard file naming convention.

1.2.2 disassemble_filename

namefiles.**disassemble_filename**(*target_path: Union[pathlib.Path, str]*, *filename_validator: Optional[namefiles.JsonschemaValidator] = None*) → dict

Disassembles a file's name into the parts defined by a file naming convention.

Parameters

- **target_path** – The file which name should be disassembled.
- **filename_validator** – The validator to validate the filename parts with its file naming convention.

Returns dict

Examples

```
>>> from namefiles import disassemble_filename
>>> from doctestprinter import doctest_print
>>> disassemled_filename = disassemble_filename(
...     "zoo/cage/Zebra#A#Afrika#_ffffff_000000.animal.stock"
... )
>>> doctest_print(disassemled_filename, max_line_width=70)
```

(continues on next page)

(continued from previous page)

```
{'identifier': 'zoo', 'extension': '.stock', 'source_id': 'Afrika', 'sub_id':
'A', 'context': 'animal', 'vargroup': ['ffffff', '000000']}
```

1.2.3 construct_filename

namefiles.**construct_filename**(*filename_template: Optional[str] = None*, *filename_validator: Optional[namefiles.JsonschemaValidator] = None*, ***filename_parts*) → str

Constructs a filename using a filename convention.

Parameters

- **filename_template** – A template providing a `format` method to be called with the fileparts.
- **filename_validator** – The validator to validate the filename parts with its file naming convention.
- ****filename_parts** – File name parts as keywords to be used for the filename construction.

Returns str

Examples

```
>>> import namefiles
>>> namefiles.construct_filename(identifier="basename")
'basename'
>>> namefiles.construct_filename(identifier="basename", extension=".txt")
'basename.txt'
>>> namefiles.construct_filename(
...     identifier="basename", sub_id="SUB1", extension=".txt"
... )
'basename#SUB1.txt'
>>> namefiles.construct_filename(
...     identifier="basename",
...     sub_id="SUB1",
...     vargroup = ["Alibaba", 40, "thieves"],
...     source_id = "arabia",
...     context = "tale",
...     extension = ".txt"
... )
'basename#SUB1#arabia#_Alibaba_40_thieves.tale.txt'
```

```
>>> namefiles.construct_filename(identifier="basename", context="tale")
Traceback (most recent call last):
...
ValueError: Filename cannot be constructed, because , 'extension' is a dependency␣
↪of 'context'
```

1.2.4 construct_filepath

namefiles.**construct_filepath**(*filename_template: Optional[str] = None*, *filename_validator: Optional[namefiles.JsonschemaValidator] = None*, *root_path: Optional[str] = None*, ***filename_parts*) → pathlib.Path

Constructs a filepath using a file naming convention.

Parameters

- **filename_template** – A template providing a `format` method to be called with the fileparts.
- **filename_validator** – The validator to validate the filename parts with its file naming convention.
- **root_path** – The file's root path.
- ****filename_parts** – File name parts as keywords to be used for the filename construction.

Returns Path

Examples

```
>>> from namefiles import construct_filepath, FILENAME_VALIDATOR
>>> sample_pathlib_path = construct_filepath(
...     root_path="/a/path", identifier="file", sub_id="NAME"
... )
>>> str(sample_pathlib_path)
'/a/path/file#NAME'
```

1.2.5 extract_filename_parts

namefiles.**extract_filename_parts**(*potential_filename_parts: dict*, *file_naming_convention: Optional[dict] = None*) → dict

Extracts filename parts from a dictionary based by a file naming convention.

Parameters

- **potential_filename_parts** – A dictionary which potentially contains filename parts to be applied by the defined file naming convention.
- **file_naming_convention** – The file naming convention jsonschema, which should be applied.

Returns dict

Examples

```
>>> from namefiles import extract_filename_parts
>>> sample_parts = {"identifier": "A", "sub_id": "BRA", "not-a": "name-part"}
>>> extract_filename_parts(sample_parts)
{'identifier': 'A', 'sub_id': 'BRA'}
```

1.2.6 get_filename_convention

namefiles.**get_filename_convention**(*convention_name: None = None*) → dict

Gets the currently defined file naming convention.

Parameters **convention_name** – The targeted file naming convention's name. Returns the standard file naming convention by default, if no name is defined.

Returns dict

Examples

```
>>> from namefiles import get_filename_convention
>>> import json
>>> standard_convention = get_filename_convention()
>>> standard_representation = json.dumps(standard_convention, indent="  ")
>>> print(standard_representation)
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Naming Convention for Specimen Files",
  "description": "JSON Schema for filenames related to specimen.",
  "name": "Standard March 2021",
  "type": "object",
  "template": "{identifier}{sub_id}{source_id}{vargroup}{context}{extension}",
  "properties": {
    "identifier": {
      "description": "The specimen's identifier by its projects name or its UUID4",
      "type": "string",
      "minLength": 1,
      "maxLength": 36,
      "pattern": "^[0-9a-zA-Z-_]+$",
      "search_pattern": "^(?P<name_part>[0-9a-zA-Z-_]+)"
    },
    "extension": {
      "type": "string",
      "search_pattern": "(?P<name_part>\\.[0-9a-zA-Z]+$)"
    },
    "source_id": {
      "type": "string",
      "minLength": 5,
      "maxLength": 12,
      "pattern": "^[0-9a-zA-Z-]+$",
      "search_pattern": "#(?P<name_part>[0-9a-zA-Z-]{5,12})",
      "prefix": "#"
    },
    "sub_id": {
      "type": "string",
      "minLength": 1,
      "maxLength": 4,
      "pattern": "^[0-9A-Z]+$",
      "search_pattern": "#(?P<name_part>[0-9A-Z]{1,4})",
      "prefix": "#"
    },
```

(continues on next page)

(continued from previous page)

```
    "context": {
      "type": "string",
      "minLength": 3,
      "maxLength": 16,
      "pattern": "^[a-zA-Z]+[0-9a-zA-Z-]+$",
      "search_pattern": "\\.(?P<name_part>[a-zA-Z]{1}[0-9a-zA-Z-]{2,15})",
      "prefix": "."
    },
    "vargroup": {
      "type": "array",
      "pattern": "^#(_[a-zA-Z0-9+-,. ]+)+$",
      "search_pattern": "#_(?P<name_part>[0-9a-zA-Z+-,. ]+(_[0-9a-zA-Z+-,. ]+)+)",
      "delimiter": "_",
      "prefix": "#_"
    }
  },
  "required": [
    "identifier"
  ],
  "dependencies": {
    "context": [
      "extension"
    ]
  }
}
```

1.2.7 get_filename_validator

namefiles.**get_filename_validator**(*convention_name: Optional[str] = None*) → namefiles.JsonschemaValidator

Returns a filename validator for applying the file naming convention.

Parameters **convention_name** – The registered conventions name.

Returns JsonschemaValidator

Raises **KeyError** – On missing convention for requested name.

Examples

```
>>> from namefiles import get_filename_validator, STANDARD_CONVENTION_NAME
>>> type(get_filename_validator())
<class 'jsonschema.validators.create.<locals>.Validator'>
```

```
>>> get_filename_convention("not existing")
Traceback (most recent call last):
...
KeyError: "No file naming convention named by 'not existing' could be found."
```

1.2.8 is_a_filename_part

namefiles.**is_a_filename_part**(*part_name: str*, *file_naming_convention: Optional[dict] = None*) → dict

Returns if the part name is within the file naming convention.

Parameters

- **potential_filename_parts** – A dictionary which potentially contains filename parts to be applied by the defined file naming convention.
- **file_naming_convention** – The file naming convention jsonschema, which should be applied.

Returns dict

Examples

```
>>> from namefiles import is_a_filename_part
>>> is_a_filename_part("identifier")
True
>>> is_a_filename_part("not-a-name-part")
False
```

1.2.9 register_filename_validator

namefiles.**register_filename_validator**(*filename_validator: namefiles.JsonschemaValidator*) → bool

Registers file naming convention. Does **not override existing** file naming conventions.

Parameters **filename_validator** –

Raises

- **TypeError** – If validator doesn't provide a schema attribute, meaning it doesn't implement JsonschemaValidator.
- **AttributeError** – If 'name' is missing within the file naming convention.

Warning: This function doesn't override already registered filename conventions. If the convention name is already occupied this function returns *False*. Check the functions return value and decide how to react on it.

Warning: This function doesn't check the full validity of the custom file naming convention. Whether your custom file naming convention runs or not should be tested using standard python testing environments. Use this function, if you know what you are doing.

Returns *True* if the naming convention is registered using the name of this convention, or *False* if the name was already occupied.

Return type bool

1.2.10 ANameGiver

namefiles.ANameGiver.set_parts(**filename_parts)	Sets filename parts with new values.
namefiles.ANameGiver.to_path([root_path])	Returns a `pathlib.Path` of the declared filename parts.
namefiles.ANameGiver.get_filename_validator()	Returns this name givers validator providing the file naming convention.
namefiles.ANameGiver.set_name_part(...)	Sets the value of a convention's filename part.
namefiles.ANameGiver.disassemble(...)	Disassembles the filename returning ANameGiver.

set_parts

namefiles.ANameGiver.**set_parts**(*self*, ***filename_parts*)

Sets filename parts with new values. Only parts complying to the file name convention are taken.

Parameters ****filename_parts** – Parts which should be set with new values.

Examples

```
>>> from doctestprinter import doctest_print
>>> from jsonschema import Draft7Validator
>>> from namefiles import ANameGiver, get_filename_convention
>>> class MyNameGiver(ANameGiver):
...     CUSTOM_VALIDATOR = Draft7Validator(get_filename_convention())
...     @classmethod
...     def get_filename_validator(cls) -> FilenameConvention:
...         # Put your custom file naming convention (jsonschema) here
...         return cls.CUSTOM_VALIDATOR
>>> sample_giver = MyNameGiver()
>>> a_lot_of_stuff = {
...     "identifier": "Z",
...     "sub_id": "BRA",
...     "location": "zoo",
...     "age": 27,
...     "name": "Hank"
... }
>>> sample_giver.set_parts(**a_lot_of_stuff)
>>> sample_giver
MyNameGiver(identifier: Z, sub_id: BRA)
```

to_path

namefiles.ANameGiver.**to_path**(*self*, *root_path: Optional[Union[str, pathlib.Path]] = None*) → pathlib.Path

Returns a `pathlib.Path` of the declared filename parts.

Parameters **root_path** – Optional root path of the resulting filename. Default will be the working directory.

Returns pathlib.Path

get_filename_validator

namefiles.ANameGiver.**get_filename_validator**() → namefiles.JsonschemaValidator
Returns this name givers validator providing the file naming convention.

Returns JsonschemaValidator

set_name_part

namefiles.ANameGiver.**set_name_part**(*self*, *part_name: str*, *part_value: str*)
Sets the value of a convention's filename part.

Parameters

- **part_name** – Name of the convention's filename part.
- **part_value** – New value of the filename part.

disassemble

namefiles.ANameGiver.**disassemble**(*filename_or_path*) → *namefiles.ANameGiver*
Disassembles the filename returning ANameGiver.

Parameters **filename_or_path** – Either a path or filename. A path's parent is used if a filepath is provided.

Returns ANameGiver

class namefiles.**ANameGiver**(***filename_parts*)
A Name Giver is the abstract base class, which can be used to define a custom file naming convention. This can achived subclassing ANameGiver and overriding its classmethod *get_filename_validator*, which needs to return a jsonschema.IValidator.

Notes

jsonschema has no declaration of IValidator. The called methods within namefiles are declared within JsonschemaValidator as a substitution.

Parameters ****filename_parts** – Filename parts for the implemented file name convention.

Examples

To enable a custom filename convention you subclass *namefiles.ANameGiver* and override the *namefiles.ANameGiver.get_filename_validator()* providing your file naming convention. In this example the naming convention of *namefiles* is used, which uses the jsonschema draft 7 specification.

```
>>> from doctestprinter import doctest_print
>>> from jsonschema import Draft7Validator
>>> from namefiles import ANameGiver, get_filename_convention
>>> class MyFilenameParts(ANameGiver):
...     CUSTOM_VALIDATOR = Draft7Validator(get_filename_convention())
...     @classmethod
...     def get_filename_validator(cls) -> FilenameConvention:
...         # Put your custom file naming convention (jsonschema) here
```

(continues on next page)

(continued from previous page)

```
...         return cls.CUSTOM_VALIDATOR
>>> sample_parts = MyFilenameParts.disassemble("A#NAME.txt")
>>> sample_parts
MyFilenameParts(root_path: ., identifier: A, extension: .txt, sub_id: NAME)
>>> str(sample_parts)
'A#NAME.txt'
>>> sample_parts.set_parts(
...     identifier="Zebra", vargroup=["in", "the"], extension=".zoo"
... )
>>> str(sample_parts)
'Zebra#NAME#_in_the.zoo'
>>> sample_parts.set_parts(
...     identifier="Z", sub_id="BRA", vargroup="", extension=""
... )
>>> str(sample_parts)
'Z#BRA'
```

Implements collections.abc.Mapping

```
>>> converted_into_dict = dict(sample_parts)
>>> doctest_print(converted_into_dict, max_line_width=70)
{'root_path': '.', 'identifier': 'Z', 'extension': '', 'source_id': '',
'sub_id': 'BRA', 'context': '', 'vargroup': ''}
```

```
>>> len(sample_parts)
7
>>> sample_parts["sub_id"]
'BRA'
```

Disassembling of path and filename

```
>>> sample_parts = MyFilenameParts.disassemble("/a/path/Z#BRA.txt")
>>> sample_parts
MyFilenameParts(root_path: /a/path, identifier: Z, extension: .txt, sub_id: BRA)
>>> str(sample_parts.to_path())
'/a/path/Z#BRA.txt'
>>> str(sample_parts.to_path(root_path="/another/path"))
'/another/path/Z#BRA.txt'
```

1.2.11 FilenameParts

namefiles.FilenameParts.set_parts(...)	Sets filename parts with new values.
namefiles.FilenameParts.to_path([root_path])	Returns a `pathlib.Path` of the declared filename parts.
namefiles.FilenameParts.get_filename_validator()	Returns this name givers validator providing the file naming convention.
namefiles.FilenameParts.set_name_part(...)	Sets the value of a convention's filename part.
namefiles.FilenameParts.disassemble(...)	Disassembles the filename returning ANameGiver.
namefiles.FilenameParts.identifier	The mandatory entity's name which relates to multiple files.
namefiles.FilenameParts.sub_id	The *sub id* is the first branch of the identifier.

continues on next page

Table 6 – continued from previous page

namefiles.FilenameParts.source_id	The *source id* states, where this file came from.
namefiles.FilenameParts.vargroup	The *group of variables* (vargroup) contains meta attributes.
namefiles.FilenameParts.context	*Context* of the file's content.
namefiles.FilenameParts.extension	The common file extension with a leading dot.

set_parts

namefiles.FilenameParts.**set_parts**(*self*, ***filename_parts*)

Sets filename parts with new values. Only parts complying to the file name convention are taken.

Parameters ****filename_parts** – Parts which should be set with new values.

Examples

```
>>> from doctestprinter import doctest_print
>>> from jsonschema import Draft7Validator
>>> from namefiles import ANameGiver, get_filename_convention
>>> class MyNameGiver(ANameGiver):
...     CUSTOM_VALIDATOR = Draft7Validator(get_filename_convention())
...     @classmethod
...     def get_filename_validator(cls) -> FilenameConvention:
...         # Put your custom file naming convention (jsonschema) here
...         return cls.CUSTOM_VALIDATOR
>>> sample_giver = MyNameGiver()
>>> a_lot_of_stuff = {
...     "identifier": "Z",
...     "sub_id": "BRA",
...     "location": "zoo",
...     "age": 27,
...     "name": "Hank"
... }
>>> sample_giver.set_parts(**a_lot_of_stuff)
>>> sample_giver
MyNameGiver(identifier: Z, sub_id: BRA)
```

to_path

namefiles.FilenameParts.**to_path**(*self*, *root_path: Optional[Union[str, pathlib.Path]] = None*) → pathlib.Path

Returns a `pathlib.Path` of the declared filename parts.

Parameters **root_path** – Optional root path of the resulting filename. Default will be the working directory.

Returns pathlib.Path

get_filename_validator

namefiles.FilenameParts.**get_filename_validator**() → Dict
Returns this name givers validator providing the file naming convention.

Returns JsonschemaValidator

set_name_part

namefiles.FilenameParts.**set_name_part**(*self*, *part_name: str*, *part_value: str*)
Sets the value of a convention's filename part.

Parameters

- **part_name** – Name of the convention's filename part.
- **part_value** – New value of the filename part.

disassemble

namefiles.FilenameParts.**disassemble**(*filename_or_path*) → *namefiles.ANameGiver*
Disassembles the filename returning ANameGiver.

Parameters **filename_or_path** – Either a path or filename. A path's parent is used if a filepath is provided.

Returns ANameGiver

identifier

FilenameParts.**identifier**
The mandatory entity's name which relates to multiple files. The *identifier* is the leading filename part.

Notes

The identifier has a maximum length of 36 characters and can consist of words [*a-zA-Z0-9_*] with the addition of the hyphen-minus '-' (U+002D), which should be the default on keyboards.

Its regular expression `^[0-9a-zA-Z-_]+$`

Examples

Minimal to maximal identifier examples.

```
a                                    # At leas 1 character is needed.
1044e098-7bfb-11eb-9439-0242ac130002 # 36 chars allows a UUID
```

Returns str

sub_id

FilenameParts.**sub_id**
The *sub id* is the first branch of the identifier.

Notes

The sub identifier allows uppercase words without the underscore [A-Z0-9] with a maximum length of 4.

Its regular expression is `^[0-9A-Z-]{1,4}+$`

The sub identifier's task is to distinguish different states of the same context. A context in this term could be different video captures of the same object with multiple cameras or just different file versions.

The sub identifier should be seen as a branch of the identifier. Not a version within a sequence.

Examples

Multiple different video captures of the same object.

```
ant#CAM0.avi
ant#CAM1.avi
ant#CAM2.avi
```

Different children (versions).

```
a#1
a#1ST
a#2ND
a#RAW
```

Returns str

source_id

FilenameParts.**source_id**
The *source id* states, where this file came from.

Notes

The source identifier allows words without underscores [*a-zA-Z0-9*] with the addition of the hyphen-minus '-' (U+002D), which should be the default on keyboards.

Its regular expression is `^[0-9A-Z-]{5-12}+$`

The source identifier states different sources, whenever the context would lead to equal filenames. it might be the name of the program or device which made this file.

Examples

A comparison of sources onto 2 different sub versions of *Zeb-a*.

```
Zeb-a#1#canon.jpg
Zeb-a#2#canon.jpg
Zeb-a#1#nikon.jpg
Zeb-a#2#nikon.jpg
```

Returns str

vargroup

FilenameParts.**vargroup**

The *group of variables* (vargroup) contains meta attributes.

Notes

Each variable of the group is a string. It allows words [*a-zA-Z0-9_*] with the addition of:

- '-' *hyphen-minus* (U+002D)
- '+' plus
- ',' comma
- '.' dot

Its regular expression is `^#(_[a-zA-Z0-9+-,. ]+)+$`

Examples in which meta attributes are stored in the filename:

- number of a subsequent sequence e.g. image sequences
- a date neither being the creation nor the change date

Examples

```
>>> from namefiles import FilenameParts
>>> FilenameParts.disassemble("Zeb-a#_000000_ffffff_1.9m_no color").vargroup
['000000', 'ffffff', '1.9m', 'no color']
```

Returns List[str]

context

FilenameParts.**context**

Context of the file's content. What is this file about?

Notes

The context allows words without underscores [a-zA-Z0-9] starting with alphabetic character.

Its regular expression is `^[a-zA-Z]+[0-9a-zA-Z-]+$`

While the file extension just states the formatting of the file like '.txt' being a text file or '.csv' being a specifically formatted text file, they do not state any information about their context.

Returns str

extension

`FilenameParts.`**`extension`**

The common file extension with a leading dot.

Notes

The extension states how the content is encoded and which structure it has.

Examples

A file ending with '.txt' is a plain text file, which is encoded with 'utf-8' in best case.

A file ending with '.csv' is a plain text file, which contains a table having *'comma seperated values'*. Other examples are common formats like *.json* or *.yml*.

Instead of creating non-common file endings for custom text based file formats. The text files should end with '.txt'. To state the custom content the *context* file part can be used.

Returns str

`class namefiles.`**`FilenameParts`**(*identifier: Optional[str] = None*, *sub_id: Optional[str] = None*, *source_id: Optional[str] = None*, *vargroup: Optional[List[str]] = None*, *context: Optional[str] = None*, *extension: Optional[str] = None*, *root_path: Optional[str] = None*, ***kwargs*)

The *filename parts* implements the current standard file naming convention. The FilenameParts is the convinient tool to make a new filename based on the latest standard file naming convention.

Parameters

- **`identifier`** – The mandatory entity's name which relates to multiple files. The *identifier* is the leading filename part.
- **`sub_id`** – The *sub id* is the first branch of the identifier.
- **`source_id`** – The *source id* states, where this file came from.
- **`vargroup`** – The *group of variables* (vargroup) contains meta attributes.
- **`context`** – *Context* of the file's content. What is this about?
- **`extension`** – The extension of this file. The extension states the files format or structure.
- **`root_path`** – The files location.

Examples

The major entry point is the disassemble method, which returns the FilenameParts instance containing all filename parts based on the latest standard file naming convention.

```
>>> from namefiles import FilenameParts
>>> sample_giver = FilenameParts.disassemble("A#NAME.txt")
>>> sample_giver
FilenameParts(root_path: ., identifier: A, extension: .txt, sub_id: NAME)
```

The FilenameParts mimics a Mapping and additionally providing the major filename parts as properties.

```
>>> sample_giver["identifier"]
'A'
>>> sample_giver["sub_id"]
'NAME'
>>> sample_giver.identifier
'A'
>>> sample_giver.identifier = "Zebra"
>>> sample_giver.identifier
'Zebra'
```

Either convert the instance to a string to get a filename (filepath)

```
>>> str(sample_giver)
'Zebra#NAME.txt'
```

or use the *FilenameParts.to_path()* method to receive a `pathlib.PurePath`.

CHAPTER

TWO

CONCEPT

The filename is defined by 6 parts, which take on different contexts, all being related to one entity.

- identifier: The mandatory name (identification) of an entity.
- sub_id: A branch of this entity.
- source_id: The source from which the file (data) origins.
- vargroup: The possibility to state variables.
- context: Context of the files content. What is in there, not how it is stored in there. The context must be always accompanied with an *extension*.
- extension: The file extension, which should state the format of the file. How is it stored in there.

All filename parts except the identifier are optional.

Within *namefiles* file naming conventions are defined by a JsonSchema using the *python jsonschema* module. *namefiles* proposes a standard naming convention, which is used if no custom naming convention is defined.

The ENBF of the namefiles's naming convention is

```
filename     ::= identifier ["#" sub_id] ["#" source_id] ["#" vargroup] ["." context] [".
↪" extension]
identifier   ::= [0-9a-zA-Z-_]{1,36}
sub_id       ::= [0-9A-Z]{1,4}
source_id    ::= [0-9A-Z]{5,12}
vargroup     ::= ("_" var_value])+
var_value    ::= [a-zA-Z0-9,.+-\ ]+
context      ::= [a-zA-Z]+[0-9a-zA-Z-]+
extention    ::= common file extension (.csv, .txt, ...)
```

CHAPTER

THREE

IMPLEMENTATION

The recommended *namefiles.FilenameParts* implements the default *namefiles* file naming convention, providing access to each part via properties.

FilenameParts.**identifier**

The mandatory entity's name which relates to multiple files. The *identifier* is the leading filename part.

Notes

The identifier has a maximum length of 36 characters and can consist of words [*a-zA-Z0-9_*] with the addition of the hyphen-minus '-' (U+002D), which should be the default on keyboards.

Its regular expression `^[0-9a-zA-Z-_]+$`

Examples

Minimal to maximal identifier examples.

```
a                                    # At leas 1 character is needed.
1044e098-7bfb-11eb-9439-0242ac130002    # 36 chars allows a UUID
```

Returns str

FilenameParts.**sub_id**

The *sub id* is the first branch of the identifier.

Notes

The sub identifier allows uppercase words without the underscore [A-Z0-9] with a maximum length of 4.

Its regular expression is `^[0-9A-Z-]{1,4}+$`

The sub identifier's task is to distinguish different states of the same context. A context in this term could be different video captures of the same object with multiple cameras or just different file versions.

The sub identifier should be seen as a branch of the identifier. Not a version within a sequence.

Examples

Multiple different video captures of the same object.

```
ant#CAM0.avi
ant#CAM1.avi
ant#CAM2.avi
```

Different children (versions).

```
a#1
a#1ST
a#2ND
a#RAW
```

Returns str

FilenameParts.**source_id**

The *source id* states, where this file came from.

Notes

The source identifier allows words without underscores [*a-zA-Z0-9*] with the addition of the hyphen-minus '-' (U+002D), which should be the default on keyboards.

Its regular expression is `^[0-9A-Z-]{5-12}+$`

The source identifier states different sources, whenever the context would lead to equal filenames. it might be the name of the program or device which made this file.

Examples

A comparison of sources onto 2 different sub versions of *Zeb-a*.

```
Zeb-a#1#canon.jpg
Zeb-a#2#canon.jpg
Zeb-a#1#nikon.jpg
Zeb-a#2#nikon.jpg
```

Returns str

FilenameParts.**vargroup**

The *group of variables* (vargroup) contains meta attributes.

Notes

Each variable of the group is a string. It allows words [*a-zA-Z0-9_*] with the addition of:

- '-' *hyphen-minus* (U+002D)
- '+' plus
- ',' comma
- '.' dot

Its regular expression is `^#(_[a-zA-Z0-9+-,. ]+)+$`

Examples in which meta attributes are stored in the filename:

- number of a subsequent sequence e.g. image sequences
- a date neither being the creation nor the change date

Examples

```
>>> from namefiles import FilenameParts
>>> FilenameParts.disassemble("Zeb-a#_000000_ffffff_1.9m_no color").vargroup
['000000', 'ffffff', '1.9m', 'no color']
```

Returns List[str]

`FilenameParts.`**`context`**

Context of the file's content. What is this file about?

Notes

The context allows words without underscores [a-zA-Z0-9] starting with alphabetic character.

Its regular expression is `^[a-zA-Z]+[0-9a-zA-Z-]+$`

While the file extension just states the formatting of the file like '.txt' being a text file or '.csv' being a specifically formatted text file, they do not state any information about their context.

Returns str

`FilenameParts.`**`extension`**

The common file extension with a leading dot.

Notes

The extension states how the content is encoded and which structure it has.

Examples

A file ending with '.txt' is a plain text file, which is encoded with 'utf-8' in best case.

A file ending with '.csv' is a plain text file, which contains a table having *'comma seperated values'*. Other examples are common formats like *.json* or *.yml*.

Instead of creating non-common file endings for custom text based file formats. The text files should end with '.txt'. To state the custom content the *context* file part can be used.

Returns str

CHAPTER

FOUR

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

INDEX