Filefinder

Release 1.0.1

Clément Haëck

Apr 24, 2024

CONTENTS

FileFinder allows to specify the structure of filenames using a simple syntax. Parts of the file structure varying from file to file are indicated within named groups, either with format strings or regular expressions (with some pre-defined values for some names). Once setup, it can:

- Find corresponding files in a directory (and sub-directories)
- Parse values from the filenames
- Select only filenames with specific values
- Generate filenames

The following example will find all files with the structure `Data/param_[parameter]/Temperature_[date].nc`:

```
finder = Finder('/.../Data', 'param_%(parameter:fmt=.1f)/%(Y)/Temperature_%(Y)%(m)%(d).nc
↪')
files = finder.get_files()
```

We can also select only some files, for instance only in january:

```
finder.fix_group('m', 1)
files = finder.get_files()
```

We can retrieve values from found files:

```
filename, matches = finder.files[0]
parameter = matches["parameter"]
# the date as a datetime object
date = filefinder.library.get_date(matches)
```

And we can generate a filename with a set of parameters:

```
finder.make_filename(parameter=0.5, Y=2000, m=1, d=1)
# Specifying the month is optional since we already fixed it to 1.
```

CHAPTER

ONE

INSTALLATION

Requirements
Python >= 3.10

FileFinder can be installed directly from pip:

```
pip install filefinder
```

or from source with:

```
pip install -e https://github.com/Descanonge/filefinder.git#egg=filefinder
```

CHAPTER

TWO

CONTENTS

2.1 Usage

Let's demonstrate the main features of FileFinder using a simple example. Detailed information about some steps will be provided in separate pages.

We are going to deal with a dataset with multiple files all located in the directory `/data/`. They are organized by sub-directories corresponding to different parameter values, then in yearly sub-directories:

```
/data/param_[parameter]/[year]/variable_[date].nc
/data/param_0.0/2012/variable_2012-01-01.nc
/data/param_0.0/2012/variable_2012-01-02.nc
...
/data/param_1.5/2012/variable_2012-01-01.nc
...
```

2.1.1 Create the Finder object

To manage this, we are going to use the main entry point of this package: the *Finder* class. Its main arguments are the root directory containing the files, and a pattern specifying the filename structure. That pattern allows to get filenames corresponding to given values, but also to scan for files matching the pattern on disk.

```
finder = Finder(
    "/data/",
    "param_%(param:fmt=.1f)/%(Y)/variable_%(Y)-%(m)-%(d).nc"
)
```

The parts that vary from file to file are indicated in the pattern by parentheses, preceded by a percent sign. Within the parentheses are specifications for a *Group*, that will handle creating the regular expression to find files and formatting values appropriately.

Important: Details on the different ways to specify a varying group are available there: *Pattern*.

Here quickly, for date related parts, we only need to indicate the name: filefinder has them as *default*. For the parameter, indicating a *string format* will suffice.

2.1.2 Fix groups

Each group can be fixed to one possible value or a set of possible values. This will restrict the filenames that match the pattern when scanning files.

Note: Also, when *creating filenames*, if a group already has a fixed value it will be used by default.

Fixing groups can be done with either the *Finder.fix_group()* or *Finder.fix_groups()* methods. Groups can be selected either by their index in the filename pattern (starting from 0), or by their name. If using a group name, multiple groups can be fixed to the same value at once.

The given value can be:

- a **number**: will be formatted to a string according to the group specification. For scanning files, the string will be properly escaped for use in a regular expression.
- a **boolean**: if the group has two options (specified with the *bool* keyword), one of the options is selected and used as a string.
- a **string**: the value is directly interpreted as a regular expression and used as-is when scanning files or creating filenames, without further escaping or formatting.
- a **list** of any of the above: each element will be formatted to a string if not already. When scanning files, all elements are considered by joining them with *OR* (`(value1|value2|...)`), and when creating files only the **first** element of the list is used.

So for example:

```
>>> finder.fix_group("param", "[a-z]+")
will be kept as is
>>> finder.fix_group("param", 3.)
will be formatted as "3\.0"
```

More practically, we could keep only the files corresponding to january:

```
finder.fix_groups("m", 1)
```

We could also select specific days using a list:

```
finder.fix_groups(d=[1, 3, 5, 7])
```

Note: Fixed values can be changed/overwritten at any time, or unfixed using the *Finder.unfix_groups()* method.

Warning: A group flagged as *:discard* will not be fixed by default, unless using the keyword argument `fix_discard` in *fix_group()* and *fix_groups()*.

2.1.3 Find files

Retrieve files

Files can be retrieved with the *Finder.get_files()* method, or from the *Finder.files* attribute. Both will automatically scan the directory for matching files and cache the results for future accesses. The files are stored in alphabetical order.

Note: The cache is appropriately voided when using some methods, like for fixing groups. For that reason, avoid setting attributes directly and use set methods.

The method *get_files()* simply returns a sorted list of the filenames found when scanning. By default the full path is returned, ie the concatenation of the root directory and the pattern part. It can also return the filename relative to the root directory (ie only the pattern part).

Instead of a flat list of filenames, *get_files()* can also arrange them in nested lists. To that end, one must provide the `nested` argument with a list that specify the order in which groups must be nested. Each element of the list gives:

- a group, by index or name, so that files be grouped together based on the value of that group
- multiple groups, by a tuple of indices or names, so files are grouped based on the combination of values from those groups.

An example might help to grasp this. Again with the same pattern, we can ask to group by values of 'param':

```
>>> finder.get_files(nested=["param"])
[
  [
    "/data/param_0.0/2012-01-01.nc",
    "/data/param_0.0/2012-01-02.nc",
    ...
  ],
  [
    "/data/param_1.5/2012-01-01.nc",
    "/data/param_1.5/2012-01-02.nc",
    ...
  ],
  ...
]
```

We obtain as many lists as different values found for 'param'. Because we did not specify any other group, the nesting stop there. But we could chose to *also* group by the year:

```
>>> finder.get_files(nested=["param", "Y"])
[
  [  # param = 0
    [  # Y = 2012
      "/data/param_0.0/2012-01-01.nc",
      ...
    ],
    [  # Y = 2013
      "/data/param_0.0/2013-01-01.nc",
      ...
    ],
```

(continues on next page)

(continued from previous page)

```
    ...
  ],
  [  # param = 1.5
    ...
  ],
  ...
]
```

Or if we wanted to group by date as well we can specify multiple groups for one nesting level:

```
>>> finder.get_files(nested=["param", ("Y", "m", "d")])
[
  [  # param = 0
    ["/data/param_0.0/2012-01-01.nc"],
    ["/data/param_0.0/2012-01-02.nc"],
    ...
  ],
  [  # param = 1.5
    ["/data/param_1.5/2012-01-01.nc"],
    ["/data/param_1.5/2012-01-02.nc"],
    ...
  ],
  ...
]
```

Note: This is aimed to work with xarray.open_mfdataset, which will merge files in a specific order when supplied a nested list of files.

Retrieve information

As some metadata might only be found in the filenames, FileFinder offer the possibility to retrieve it easily. The Finder caches a list of files matching the pattern, along with information about parts that matched the groups.

The *Finder.files* attribute stores a list of tuples each containing a filename and a *Matches* object storing that information.

Note: One can also scan any filename for matches with the *Finder.find_matches()* function.

For most cases, the simplest is to access the Matches object with a group index or name:

```
>>> file, matches = finder.files[0]
>>> matches["param"]
0.0  # a float, parsed from the filename
```

This method has several caveats:

- When using a group name, the first group in the pattern with that name is taken, even if there could be more groups with different values (a warning is issued if that is the case).
- Only groups not flagged as ':discard' will be selected. If no group can be found, an error will be raised.

- The parsing of a value from the filename can fail for a variety of reasons, if that is the case, an error will be raised.

To counter those, one can use *Matches.get_values()* which will return a list of values corresponding to the selected group(s). It has arguments `keep_discard` and `parse` to choose whether keep discarded groups and whether to use the parsed value or solely the string that matched.

Matches.get_value() will return the first element of that list, raise if the list is empty, and warn if the values are not all equal.

Note: `matches[key]` is a thin wrapper around `matches.get_value(key, parse=True, keep_discard=False)`.

As date/time values are scattered among multiple groups, the package supply the function *library.get_date()* to easily retrieve a `datetime` object from matches:

```
from filefinder.library import get_date
matches = finder.get_matches(filename)
date = get_date(matches)
```

Directories in pattern

The pattern can contain directory separators. The *Finder* can explore sub-directories to find the files.

Important: In the pattern, a directory separator should always be indicated with the forward slash `/`, even on Windows where we normally use the backslash. It will be replaced by the correct character when necessary.

We do this because the backslash has special meanings in regular expressions, and it is difficult to disambiguate the two.

The scanning process is as follows. It first generates a regular expression based on the pattern and the fixed values. This expression is meant to match paths relative to the root directory and have a capturing group for each pattern group.

The Finder then explore all sub-directories to find matching files using one of two methods.

1. By default, the regular expression is split at each path separator occurrence, so that we can eliminate folders that do not match the pattern. However, it cannot deal with some patterns in which a group contains a path separator.
2. For those more complicated patterns, by setting the attribute/parameter *Finder.scan_everything* to true, we will explore all sub-directories up to a depth of *Finder.max_scan_depth*.

The second method can be more costly for some directory structures —with many siblings folders for instance— but can deal with more exotic patterns. A likely example could be that of an optional directory:

```
>>> "basedir/%(subdir:bool=subdir_name/:)rest_of_pattern"
basedir/rest_of_pattern
basedir/subdir_name/rest_of_pattern
```

2.1.4 Create filenames

Using the information contained in the filename pattern we can also generate arbitrary filenames. This is done with *Finder.make_filename()*. Any group that does not already have its value *fixed* must have a value supplied as argument. As for fixing, a value will be appropriately formatted but a string will be left untouched.

So for instance:

```
>>> finder.make_filename(param=1.5, Y=2012, m=1, d=5)
"/data/param_1.5/2012-01-05.nc"
```

we can also fix some groups:

```
>>> finder.fix_groups(param=2., Y=2014)
>>> finder.make_filename(m=5, d=1)
"/data/param_2.0/2014-05-01.nc"
>>> finder.make_filename(m=6, d=1)
"/data/param_2.0/2014-06-01.nc"
```

and also supply a string to forgo formatting:

```
>>> finder.make_filename(param="this-feels-wrong", m=6, d=1)
"/data/param_this-feels-wrong/2014-06-01.nc"
```

2.2 Pattern

The pattern specifies the structure of the filenames relative to the root directory. Parts that vary from file to file are indicated by **groups**, enclosed by parenthesis and preceded by '%'. They are represented by the *Group* class.

Each group definition starts with a *name*, and is then followed by multiple optional properties, separated by colons (in no particular order):

Property	Format	Description
Format string	`:fmt=<format string>`	Use a python format string to match this group in filenames.
Boolean format	`:bool=<true>[:<false>`	Choose between two alternatives. The second option (false) can be omitted.
Custom regex	`:rgx=<custom regex>`	Specify a custom regular expression directly.
Optional flag	`:opt`	Mark the group as optional.
Discard flag	`:discard`	Discard the value parsed from this group when retrieving information.

So for instance, we can specify a filename pattern that will match an integer padded with zeros, followed by two possible options:

```
>>> "parameter_%(param:fmt=04d)_type_%(type:bool=foo:bar).txt"
parameter_0012_type_foo.txt
parameter_2020_type_bar.txt
```

Note: Groups are uniquely identified by their index in the pattern (starting at 0) and can share the same name. When using a name rather than an index, some functions may return more than one result if they are multiple groups with

that name.

Warning: Groups are first found in the pattern by looking at matching parentheses. The pattern should thus have balanced parentheses or unexpected behavior can occur.

2.2.1 Name

The name of the group will dictate the regex and format string used for that group (unless overridden the 'fmt' and 'rgx' properties). The *Group.DEFAULT_GROUPS* class attribute will make the correspondence between name and regex:

Name		Regex	Format
F	Date (YYYY-MM-DD)	%Y-%m-%d	s
x	Date (YYYYMMDD)	%Y%m%d	08d
X	Time (HHMMSS)	%H%M%S	06d
Y	Year (YYYY)	\d{4}	04d
m	Month (MM)	\d\d	02d
d	Day of month (DD)	\d\d	02d
j	Day of year (DDD)	\d{3}	03d
B	Month name	[a-zA-Z]*	s
H	Hour 24 (HH)	\d\d	02d
M	Minute (MM)	\d\d	02d
S	Seconds (SS)	\d\d	02d
I	Index	\d+	d
text	Letters	\w	s
char	Character	\S*	s

Most of them are related to dates and follow the specification of strftime() and strptime() Behavior and strftime.

A letter preceded by a percent sign '%' in the regex will be recursively replaced by the corresponding name in the table. This can be used in the custom regex. This still counts as a single group and its name will not be changed, only the regex. So `%x` will be replaced by `%Y%m%d`, in turn replaced by `\d{4}\d\d\d\d`. A percentage character in the regex is escaped by another percentage ('%%').

2.2.2 Format string

All the possible use cases are not covered in the table above. A simple way to specify a group is by using a format string following the Format Mini Language Specification. This will automatically be transformed into a regular expression.

Having a format specified has other benefits: it can be used to convert values into strings to generate a filename from parameters values (using *Finder.make_filename*), or vice-versa to parse filenames matches into parameters values.

It's easy as `scale_%(scale:fmt=.1f)` which will find files such as `scale_15.0` or `scale_-5.6`. Because we know how to transform a value into a string we can fix the group directly with a value:

```
finder.fix_group('scale', 15.)
```

or we can generate a filename:

```
>>> finder.make_filename(scale=2.5)
'scale_2.5'
```

In the opposite direction, we can retrieve a value from a filename:

```
>>> matches = finder.find_matches('scale_2.5')
>>> print(matches['scale'].get_match())
2.5  # a float
```

If the format is never specified, it defaults to a `s` format.

Warning: Only s, d, f, e, and E format types are supported.

Parsing of numbers will fail in some ambiguous (and quite unrealistic) cases that involves alignment padding with numbers or the minus signs. Creating a format object where we can't unambiguously remove the padding character is not allowed and will raise a *DangerousFormatError*.

Similarly, for a string format (s) it can be impossible to separate correctly the alignment padding character (the "fill") from the actual value. Here the user is entrusted with making sure the format fill character is adapted to the expected values to parse.

2.2.3 Boolean format

The boolean format allows to easily select between two *strings*. It is specified as `:bool=<true>[:<false>]`. The second option (false), can be omitted.

Here are a couple of examples. `my_file%(special:bool=_special).txt` would match both `my_file.txt` and `my_file_special.txt`. We could select only 'special' files using `finder.fix_groups(special=True)`.

We can also specify both options with `my_file_%(kind:bool=good:bad).txt`, and select either like so

```
>>> finder.make_filename(kind=True)
my_file_good.txt
>>> finder.make_filename(kind=False)
my_file_bad.txt
```

2.2.4 Optional flag

The optional flag `:opt` marks the group as an optional part of the pattern. In effect, it appends a `?` to the group regular expression. It does not affect the group in other ways.

2.2.5 Custom regex

Finally, one can directly use a regular expression. This will supersede the default regex, or the one generated from the format string if specified.

It can be done like so:

```
idx_%(idx:rgx=\d+?)
```

2.2.6 Discard keyword

Information can be retrieved from the matches in the filename, but one might discard a group so that it is not used. For example for a file of weekly averages with a filename indicating the start and end dates of the average, we might want to only recover the starting date:

```
sst_%(x)-%(x:discard)
```

Note: By default, when *fixing a group to a value*, discarded groups will not be fixed. This can be overridden with the `fix_discard` keyword argument.

2.2.7 Regex outside groups

By default, special characters (`()[]{}?*+-|^$\\.&~# \t\n\r\v\f`) outside of groups are escaped, and thus not interpreted as a regular expression. To use regular expressions outside of groups, it is necessary to pass `use_regex=True` when creating the Finder object.

Note: When using regex outside groups, *Finder.make_filename* won't work.

2.3 API References

Content

finder.Finder	Find files using a filename pattern.

Submodules

finder	Main class.
format	Generate regex from string format, and parse strings.
group	Group management.
library	Functions to retrieve values from filename.
matches	Matches management.

2.3.1 filefinder.finder

Main class.

Classes

Finder(root, pattern[, use_regex, ...])	Find files using a filename pattern.

class Finder(*root*, *pattern*, *use_regex=False*, *scan_everything=False*)

Find files using a filename pattern.

The Finder object is the main entrance point to this library. Given a root directory and a filename pattern, it can search for all corresponding files.

Parameters

- **root** (*str*) – The root directory of the filetree where all files can be found.
- **pattern** (*str*) – A regular expression with the addition of 'groups'. See *Find files* for details.
- **use_regex** (*bool*) – If True, characters outside of groups are considered as valid regex (and not escaped). Default is False.
- **scan_everything** (*bool*) – If true, look into all sub-directories up to a depth of *max_scan_depth* . This is appropriate if the pattern contains optional sub-directories. If false (default), check that every sub-directory matches its part of the regular expression, thus avoiding some work.

find_files()

Find files to scan and store them.

Is automatically called when accessing *files* or *get_files()*. Sort files alphabetically.

find_matches(*filename*, *relative=True*)

Find matches for a given filename.

Apply regex to *filename* and return the results as a *Matches* object. Fixed values are applied as normal.

Parameters

- **filename** (*str*) – Filename to retrieve matches from.
- **relative** (*bool*) – True if the filename is relative to the finder root directory (default). If False, the filename is made relative before being matched.

Returns

matches – A *Matches* object, or None if the filename did not match.

Return type

Matches | None

fix_group(*key*, *value*, *fix_discard=False*)

Fix a group to a string.

Parameters

- **key** (*int | str*) – Can be the index of a group in the pattern (starts at 0), or the name of a group. If multiple groups share the same name, they are all fixed to the same value.
- **value** (*str | Any*) – Will replace the match for all files. Can be a string, or a value that will be formatted using the group format string. A list of values will be joined by the regex '|' OR. A string will be interpreted as a regular expression, so all special characters should be properly escaped.
- **fix_discard** (*bool*) – If True, groups with the 'discard' option will still be fixed. Default is False.

fix_groups(*fixes=None*, *fix_discard=False*, ***fixes_kw*)

Fix multiple groups at once.

Parameters

- **fixes** (*dict[Any, str | Any] | None*) – Dictionnary of *{group key: value}*. See *fix_group()* for details.
- **fix_discard** (*bool*) – If True, groups with the 'discard' option will still be fixed. Default is False.
- **fixes_kw** (*str | Any*) – Same as *fixes*. Takes precedence.

get_absolute(*filename*)

Get absolute path to filename.

Parameters
filename (*str*) –

Return type
str

get_files(*relative=False*, *nested=None*)

Return files that matches the regex.

Lazily scan files: if files were already scanned, just return the stored list of files. Scanned files are flushed if the regex is changed (by fixing group for instance).

Parameters

- **relative** (*bool*) – If True, filenames are returned relative to the finder root directory. If not, paths are absolute (default).
- **nested** (*Sequence[str | Sequence[str]] | None*) – If not None, return nested list of filenames with each level corresponding to a group, or set of group. Last set in the list is at the innermost level.

Raises
KeyError – A group name in *nested* is not found in the pattern.:

Return type
list

get_groups(*key*)

Return list of groups corresponding to key.

Parameters
key (*int, str, or list of int*) – Can be group index, name, or group+name combination with the syntax: 'group:name'.

Returns
List of groups corresponding to key.

Raises

- **KeyError** – No group found.:
- **TypeError** – Key type is not valid.:

Return type
list[*filefinder.group.Group*]

get_pattern()

Get filename pattern.

Return type
str

get_regex()

Return regex.

Return type
str

get_regex_subdirs()

Return regexes for each sub-directory.

Return type
list[str]

get_relative(*filename*)

Get filename path relative to root.

Parameters
filename (*str*) –

Return type
str

make_filename(*fixes=None*, *relative=False*, ***kw_fixes*)

Return a filename.

Replace groups with provided values. All groups must be fixed prior, or with *fixes* argument.

Parameters

- **fixes** (*dict | None*) – Dictionnary of fixes (group name or index: value). For details, see *fix_group()*. Will (temporarily) supplant group fixed prior. If prior fix is a list, first item will be used.
- **relative** (*bool*) – If the filename should be relative to the finder root directory. Default is False.
- **kw_fixes** (*Any*) – Same as *fixes*. Takes precedence.

Raises
ValueError – *use_regex* is activated.:

Return type
str

set_pattern(*pattern*)

Set pattern and parse for group objects.

Parameters
pattern (*str*) –

set_scan_everything(*scan_everything*, */*)

Set value for attribute *scan_everything*.

Void cache if necessary.

Parameters
scan_everything (*bool*) –

Return type
None

set_use_regex(*use_regex*, /)

Set value for attribute *use_regex*.

Parameters
use_regex (*bool*) –

Return type
None

unfix_groups(**keys*)

Unfix groups.

Parameters
keys (*str*) – Keys to find groups to unfix. See *get_groups()*. If no key is provided, all groups will be unfixed.

property files: list[tuple[str, *filefinder.matches.Matches*]]

List of filenames and their matches.

Will scan files when accessed and cache the result, if it has not already been done.

max_scan_depth: int = 32

Maximum sub-directory depth to scan when *scan_everything* is True.

property n_groups: int

Number of groups in pre-regex.

root: str

The root directory of the finder.

scan_everything: bool

Whether to scan all subdirectories.

scanned: bool

True if files have been scanned with current parameters.

Is reset to False if the cache (of scanned files) is voided, for instance by operation like changing fixed values of groups.

use_regex: bool

If True, characters outside of groups are considered as valid regex (and not escaped). Default is False.

2.3.2 filefinder.format

Generate regex from string format, and parse strings.

Parameters of the format-string are retrieved. See Format Mini Language Specification.

Thoses parameters are then used to generate a regular expression, or to parse a string formed from the format.

Only ‘s’, ‘d’, ‘f’, ‘e’ and ‘E’ formats types are supported.

The width of the format string is not respected when matching with a regular expression.

Module Attributes

FORMAT_REGEX	The regular expression used to parse a format string.

Functions

Format(format)	Parse format parameters and return appropriate Format object.
get_format(format)	Parse format parameters and return appropriate Format object.

Classes

FormatAbstract(fmt, params)	Represent a format string.
FormatFloat(*args, **kwargs)	Represent a format string for floats (type f, e, E).
FormatInteger(*args, **kwargs)	Represent a format string for integers (type d).
FormatNumberAbstract(*args, **kwargs)	Represent a format string for numbers (type d, f, e, E).
FormatString(*args, **kwargs)	Represent a format string for strings (type s).

Exceptions

DangerousFormatError	Dangerous format-string leading to ambiguities.
FormatError	Error related to Format object.
FormatParsingError	Could not parse a format-string.
FormatValueParsingError	Could not parse value.
InvalidFormatTypeError	Unsupported type of format-string.

exception DangerousFormatError

Dangerous format-string leading to ambiguities.

exception FormatError

Error related to Format object.

exception FormatParsingError

Could not parse a format-string.

exception FormatValueParsingError

Could not parse value.

exception InvalidFormatTypeError

Unsupported type of format-string.

class FormatAbstract(*fmt*, *params*)

Represent a format string.

Can generate an appropriate regular expression corresponding to that format string (to some limitations), generate a string from a value, or parse such a string into a value.

Users are not meant to instanciate those objects directly, use *get_format()* instead (or its alias for retro-compatibility *Format()*).

Parameters

- **fmt** (*str*) – Format string.
- **params** (*Mapping[str, Any]*) – Mapping of options/parameters of the format mini-language to their values. (type, fill, align, sign, alternate, zero, width, grouping, precision). They should not contain None values.

add_outer_alignement(*rgx*)

Add necessary regex for alignement characters.

If width is not specified, does nothing.

Parameters
rgx (*str*) –

Return type
str

format(*value*)

Return formatted string of a value.

Parameters
value (*Any*) –

Return type
str

generate_expression(*capture=False*)

Generate a regular expression matching strings created with this format.

Parameters
capture – If true, add capturing groups that will be used to parse the value by selecting only relevant information. Default is false.

Return type
str

get_fill_regex()

Return regex for matching fill characters.

parse(*s*)

Parse string generated with this format into an appropriate value.

Parameters
s (*str*) –

Return type
Any

class FormatFloat(**args*, ***kwargs*)

Represent a format string for floats (type f, e, E).

generate_expression(*capture=False*)

Generate a regular expression matching strings created with this format.

Return type
str

`get_left_of_decimal()`

Get regex for the numbers left of decimal point.

Will deal with grouping if present. Some simplifications for eE formats. Only use groupings for '0=' alignment and enforce single digits grouping.

Return type
str

`get_right_of_decimal()`

Return regex for numbers after decimal points.

Including the decimal point itself. It will respect 'alternate' option and the specified precision.

Return type
str

`parse(s)`

Parse string generated with format.

Parameters
s (*str*) –

Return type
float

`class FormatInteger(*args, **kwargs)`

Represent a format string for integers (type d).

`generate_expression(capture=False)`

Generate regex from format string.

Return type
str

`parse(s)`

Parse string generated with format.

Parameters
s (*str*) –

Return type
int

`class FormatNumberAbstract(*args, **kwargs)`

Represent a format string for numbers (type d, f, e, E).

`get_left_of_decimal()`

Get regex for the numbers left of decimal point.

Will deal with grouping if present.

Return type
str

`get_sign_regex(capture=False)`

Get sign regex with approprite zero padding.

Return type
str

`prepare_parse`(*s*)

Remove special characters.

Remove characters that throw off int() and float() parsing: fill/alignment characters and grouping symbols.

Returns
s – a string ready to be casted to the appropriate type.

Parameters
s (*str*) –

Return type
str

`class FormatString`(**args*, ***kwargs*)

Represent a format string for strings (type s).

`generate_expression`(*capture=False*)

Generate a regular expression matching strings created with this format.

Return type
str

`parse`(*s*)

Parse string generated with this format into an appropriate value.

Parameters
s (*str*) –

Return type
str

`Format`(*format*)

Parse format parameters and return appropriate Format object.

Parameters
format (*str*) –

Return type
FormatAbstract

`get_format`(*format*)

Parse format parameters and return appropriate Format object.

Parameters
format (*str*) –

Return type
FormatAbstract

```
FORMAT_REGEX = '((?P<fill>.)?(?P<align>[<>=^]))?(?P<sign>[-+
])?(?P<z>z)?(?P<alternate>#)?(?P<zero>0)?(?P<width>\\d+?)?(?P<grouping>[,_])?(?
P<precision>\\.\\d+?)?(?P<type>[a-zA-Z])'
```

The regular expression used to parse a format string.

Follows the Format Specification Mini-Language. `[[fill]align][sign]["z"]["#"]["0"][width][grouping_option]["."
" precision][type]`

2.3.3 filefinder.group

Group management.

Module Attributes

GroupKey	Can be used to select one or more groups in a pattern.

Classes

Group(definition, idx)	Manage a group inside the filename pattern.

Exceptions

GroupParseError(message[, group])	Custom errors when parsing group definition.

exception GroupParseError(*message*, *group=None*)

Custom errors when parsing group definition.

Parameters

- **message** (*str*) –
- **group** (*Group | None*) –

class Group(*definition*, *idx*)

Manage a group inside the filename pattern.

Parameters

- **definition** (*str*) – Group definition.
- **idx** (*int*) – Index of the group in the filename pattern.

Raises

GroupParseError – Invalid group definition.:

fix_value(*fix*)

Fix the group regex to a specific value.

Parameters

fix (*Any | bool | str*) – A string is directly used as a regular expression, otherwise the value is formatted according to the group 'format' specification.

format(*value*)

Return formatted string from value.

Parameters

value (*Any*) –

Return type

str

get_regex()

Get group regex.

Returns the fixed value if previously specified. Insert the regex into a capturing group, and make it optional if the `:opt` was indicated

Return type
str

parse(*string*)

Return parsed value from string.

Parameters
string (*str*) –

Return type
Any

unfix()

Unfix value.

DEFAULT_GROUPS = {'B': ['[a-zA-Z]*', 's'], 'F': ['%Y-%m-%d', 's'], 'H': ['\\d\\d', '02d'], 'I': ['\\d+', 'd'], 'M': ['\\d\\d', '02d'], 'S': ['\\d\\d', '02d'], 'X': ['%H%M%S', '06d'], 'Y': ['\\d{4}', '04d'], 'char': ['\\S*', 's'], 'd': ['\\d\\d', '02d'], 'j': ['\\d{3}', '03d'], 'm': ['\\d\\d', '02d'], 'text': ['\\w', 's'], 'x': ['%Y%m%d', '08d']}

Regex and format strings for various default groups.

See the *Name* section of documentation for details.

PATTERN = re.compile('(?P<name>[^:]+?)(?:(?P<fmt>:fmt=.+?)|(?P<rgx>:rgx=.*?)|(?P<bool>:bool=.*?(?::.*?)??)|(?P<opt>:opt)|(?P<discard>:discard)){,5}')

Pattern used to find properties in group definition.

definition

The string that created the group `%(definition)`.

discard: bool

If the group should not be used when retrieving values from matches.

fmt: *FormatAbstract*

Format string object.

idx: int

Index inside the pre-regex.

name: str

Group name.

optional: bool

If True, the whole group is marked as optional `()?`. Is set to False unless specification ':opt' is indicated.

options: tuple[str, str] | None

Tuple of the two possibilities indicated by the full `:bool` specification, in order (False, True), so that a simple getitem works.

rgx: str

Regex.

GroupKey = int | str

Can be used to select one or more groups in a pattern.

2.3.4 filefinder.library

Functions to retrieve values from filename.

Functions

get_date(matches[, default_date])	Retrieve date from matched elements.

get_date(*matches*, *default_date=None*)

Retrieve date from matched elements.

If a matcher is *not* found in the filename, it will be replaced by the element of the default date argument. Matchers that can be used are (in order of increasing priority): YBmdjHMSFxX. If two matchers have the same name, the last one in the pre-regex will get priority.

Parameters

- **matches** (*Matches*) – Matches obtained from a filename.
- **default_date** (*dict | None*) – Default date. Dictionnary with keys: year, month, day, hour, minute, and second. Defaults to 1970-01-01 00:00:00

Return type

datetime

2.3.5 filefinder.matches

Matches management.

Functions

get_groups_indices(groups, key)	Get sorted list of groups indices corresponding to key.

Classes

Match(group, match, idx)	Match extract from a filename.
Matches(match, groups)	Scan an input file and store the results.

class Match(*group*, *match*, *idx*)

Match extract from a filename.

Parameters

- **group** (*Group*) – Group used to get this match.
- **match** (*Match*) – Match object for the complete filename.

- **idx** (*int*) – Index of the group in the match object.

group

Group used to get this match.

match_str

String matched in the filename.

start

Start index of match in the filename.

end

End index of match in the filename.

match_parsed

Parsed value. None if parsing was not successful.

get_match(*parse=True*)

Get match string or value.

Parameters

parse (*bool*) – If True (default), and the parsing was successful, return the parsed value instead of the matched string.

Raises

ValueError – Could not parse the match.:

Return type

str | *Any*

class Matches(*match*, *groups*)

Scan an input file and store the results.

Parameters

- **match** (*Match*) – Match object obtained from a filename. It should have as much capturing groups as the pattern.
- **groups** (*Sequence[Group]*) – Sequence of Groups objects present in the pattern.

classmethod from_filename(*filename*, *pattern*, *groups*)

Find matches for a given filename.

Parameters

- **filename** (*str*) – Filename to retrieve matches from.
- **pattern** (*Pattern | str*) – Compiled match pattern to use. If left to None, we generate the current regex.
- **groups** (*Sequence[Group]*) –

Returns

matches – A *Matches* object, or None if the filename did not match.

Raises

IndexError – Not as many matches as groups. Maybe one of the group regex contains an additional (unwanted) capturing group ?

Return type

Matches | None

get_matches(*key*, *keep_discard=False*)

Get Match objects corresponding to key.

Parameters

- **key** (*int | str*) – Group(s) to select, either by index or name.
- **keep_discard** (*bool*) – If true groups with the 'discard' option are kept. Defauult is false.

Returns

List of Match corresponding to the key.

Return type

list[*filefinder.matches.Match*]

get_value(*key*, *parse=True*, *keep_discard=False*)

Get matched value corresponding to key.

Return a single value. If multiple groups correspond to **key**, the value of the first one to appear in the pattern is returned.

Parameters

- **key** (*int | str*) – Group(s) to select, either by index or name.
- **parse** (*bool*) – If True (default), return the parsed value. If False return the matched string.
- **keep_discard** (*bool*) – If true groups with the 'discard' option are kept. Defauult is false.

Raises

KeyError – No group with no 'discard' option was found.:

Return type

str | *Any*

get_values(*key*, *parse=True*, *keep_discard=False*)

Get matched values corresponding to key.

Return a list of values, even if only one group is selected.

Parameters

- **key** (*int | str*) – Group(s) to select, either by index or name.
- **parse** (*bool*) – If True (default), return the parsed value. If False return the matched string.
- **keep_discard** (*bool*) – If true groups with the 'discard' option are kept. Defauult is false.

Return type

list[str | *Any*]

groups

Groups used.

matches: list[*filefinder.matches.Match*]

Matches for a single filename.

get_groups_indices(*groups*, *key*)

Get sorted list of groups indices corresponding to key.

Key can be an integer index, or a string of a group name. Since multiple groups can share the same name, multiple indices can be returned (sorted).

Raises

- **IndexError** – No group found corresponding to the key:
- **TypeError** – Key is not int or str:

Parameters

- **groups** (*list[filefinder.group.Group]*) –
- **key** (*int | str*) –

Return type
list[int]

Source code: https://github.com/Descanonge/filefinder

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

M

N

O

P

R

S

U