pathtrees

Release 0.0.3

Bea Steers

Oct 19, 2022

CONTENTS

Have you ever managed a project with a complex file structure where your paths encode a lot of information? Aren't you tired of sprinkling your code with tons of splits and joins and index counting? I am. It gets so confusing and hard to reason about sometimes. Instead, I want to define my file structure in one place and just fill it with variables.

Here's what we can do with a single path object.

```
import pathtrees

path = pathtrees.Path('data/{sensor_id}/raw/{date}/temperature_{file_id:04d}.csv')
path.update(sensor_id='asdf-123')  # assign the sensor ID to the path

# format the path using some data
path_13 = path.format(date='02022022', file_id=13)
assert path_13 == 'data/asdf-123/raw/02022022/temperature_0013.csv'
# now use the formatted string, along with the path format, to parse the data from the␣
↪path
assert path.parse(path_13) == {'sensor_id': 'asdf-123', 'date': '02022022', 'file_id':␣
↪13}
```

And it's pretty handy with glob too - you can glob over any variables that haven't been specified.

```
# list all files for asdf-123 on 02/02/2022
for f in path.specify(date='02022022').glob():  # specify() ~~ copy.update()
    # parse the file ID out of the path
    print(path.parse(f)['file_id'])

# list all dates
date_path = path.parent.specify(sensor_id='asdf-123')
for date_dir in date_path.glob():
    # parse the date out of each path
    print("Date": date_path.parse(date_dir)['date'])
```

Now let's see a whole directory tree.

Here is an example where we're processing an audio dataset and want to save some outputs to disk.

By using `pathtrees` we can define the path structure at the top of a file and then the rest of your code can operate independent of that structure. Want to add a prefix or suffix to your filename? Want to move a couple directories around? Feeling evil and want to nest the SPL files under 15 extra directories? `pathtrees` D. G. A. F.. And neither will your code! As long as the core pieces of information are still there (here: `{sensor_id}` and `{file_id}`) the rest of your code doesn't have to know about it!

First define the path structure.

```
import pathtrees
import librosa

paths = pathtrees.tree('{project}', {
    'data': {
        '{sensor_id}': {
            '': 'sensor',
            'audio': { '{file_id:04d}.flac': 'audio' },
            'spl': { 'spl_{file_id:04d}.csv': 'spl' },
            'embeddings': { 'emb_{file_id:04d}.csv': 'embeddings' },
        },
    },
```

(continues on next page)

(continued from previous page)

```
})
# set some data to start with
paths.update(project='some-project')
```

Let's try formatting some path objects.

```
# partial format

# treating the path as a string will partially format it
# meaning that only the keys that are defned will be replaced.
assert paths.audio == 'some-project/data/{sensor_id}/audio/{file_id:04d}.flac'
assert (
    paths.audio.partial_format(sensor_id='aaa') ==
    'some-project/data/aaa/audio/{file_id:04d}.flac')

# format

try:
    paths.audio.format(sensor_id='aaa')  # forgot file ID
except KeyError:
    print("oops")

# when you have all data specified, you can format it and get a complete path
# and then you can take a formatted path and reverse it to get the data back out.
p = paths.audio.format(sensor_id='aaa', file_id=0)
assert p == 'some-project/data/aaa/audio/0000.flac'
assert (
    paths.audio.parse(p) ==
    {'project': 'some-project', 'sensor_id': 'aaa', 'file_id': 0})
```

But don't worry, if the path is missing data and you try to use it as a path, it will throw an error.

```
try:
    with open(paths.spl, 'r') as f:  # some-project/data/{sensor_id}/audio/{file_id:04d}.
↪flac
        ...
except KeyError:
    print("I didn't provide all of the data, so this was bound to happen.")

spl_path = paths.spl.specify(sensor_id='bbb', file_id=15)
with open(spl_path, 'r') as f:  # some-project/data/bbb/audio/0015.flac
    print("Ah much better..")
    print(f.read())
```

Now let's use the paths to deal with some data.

```
# loop over sensors - {sensor_id} automatically turned to '*'
for sensor_dir in path.sensor.glob():  # some-project/data/*
    sensor_id = path.sensor.parse(sensor_dir)['sensor_id']

    # loop over a sensors flac files - {file_id} automatically turned to '*'
    for audio_fname in path.audio.glob():  # some-project/data/{sensor_id}/audio/*.flac
        y, sr = librosa.load(audio_fname)
```

(continues on next page)

(continued from previous page)

```
        # convert audio path to an spl path - some-project/data/{sensor_id}/spl/{file_id}
↪.csv
        spl_fname = path.translate(audio_fname, 'audio', 'spl')
        # convert audio path to an embedding path - some-project/data/{sensor_id}/
↪embedding/{file_id}.csv
        embedding_fname = path.translate(audio_fname, 'audio', 'embedding')

        # just make sure that everything is in order
        file_id = path.audio.parse(audio_fname)['file_id']
        assert sensor_id in spl_fname and file_id in spl_fname
        assert sensor_id in embedding_fname and file_id in embedding_fname

        # calculate some stuff and write to file
        write_csv(spl_fname, get_spl(y, sr))
        write_csv(embedding_fname, get_embedding(y, sr))
```

See how working with the paths is all independent of the actual folder structure? No path joins or weird splits and split counting to parse out the bits and pieces of a path.

As long as you preserve the basic data relationships, (here it's a many-to-one between data and sensors), then you can change the file structure at the top and not have to worry about it elsewhere.

CHAPTER

ONE

INSTALLATION

```
pip install pathtrees
```

1.1 API

pathtrees.**tree**(*root: Union[str, _TREE_DEF_TYPE, None] = None*, *paths: Union[str, _TREE_DEF_TYPE, None] = None*, *data: dict | None = None*) → *Paths*

Build paths from a directory spec.

Parameters

- **root** (*str*) – the root directory.
- **paths** (*dict*) – the directory structure.

Returns

The initialized Paths object

```
import pathtrees

# define the file structure

path = pathtrees.tree('{project}', {
    'data': {
        '{sensor_id}': {
            '': 'sensor',
            'audio': { '{file_id:04d}.flac': 'audio' },
            'spl': { 'spl_{file_id:04d}.csv': 'spl' },
            'embeddings': { 'emb_{file_id:04d}.csv': 'embeddings' },
        },
    },
})
```

Note: use empty strings to reference the directory. This works because `os.path.join(path, '') == path`

class pathtrees.**Path**(**args*, *data: dict | None = None*, *tree:* Paths | *None = None*)

Represents a `pathlib.Path` with placeholders for bits of data. It uses python string formatting to let you fill in the missing bits at a later date.

```
path = pathtrees.Path('projects/{name}/images/frame_{frame_id:04d}.jpg')
path.update(name='my_project')

# loop over all frames
for f in path.glob():
    # print out some info about each frame
    data = path.parse(f)
    print("frame ID:", data['frame_id'])
    print("path:", f)
    ...  # do something - load an image idk
```

There are quite a few methods that had to be wrapped from the original path object so that if we manipulate the path in any way that it can copy the extra attributes needed to manage the data.

rjoinpath(*root: PosixPath*) → *Path*

Return an absolute form of the path. TODO: is there a better way?

property copy: P

Creates a copy of the path object so that data can be altered without affecting the original object.

update(**kw*) → P

Update specified data in place

specify(**kw*) → P

Update specified data and return a new object.

unspecify(**keys*, *inplace: bool = True*, *parent: bool = True*) → P

Remove keys from path dictionary

property fully_specified: bool

Check if the path is fully specified (if True, it can be formatted without raising an Underspecified error.).

format(**kw*) → str

Insert data into the path string. (Works like string format.)

Raises

KeyError if the format string is underspecified. –

partial_format(**kw*) → str

Format a field, leaving all unspecified fields to be filled later.

glob_format(**kw*) → str

Format a field, setting all unspecified fields as a wildcard (asterisk).

format_path(**kw*) → PosixPath

Insert data into the path string. (Works like string format.)

Raises

KeyError if the format string is underspecified. –

partial_format_path(**kw*) → P

Format a field, setting all unspecified fields as a wildcard (asterisk).

glob_format_path(**kw*) → PosixPath

Format a field, setting all unspecified fields as a wildcard (asterisk).

maybe_format(**kw*) → Union[str, P]

Try to format a field. If it fails, return as a Path object.

glob(**fs*) → List[str]

Glob over all unspecified variables.

Parameters
***path** (*str*) – additional paths to join. e.g. for a directory you can use "*.txt" to get all .txt files.

Returns
The paths matching the glob pattern.

Return type
list

iglob(**fs*) → Iterable[str]

Iterable glob over all unspecified variables. See *glob()* for signature.

rglob(**fs*) → List[str]

Recursive glob over all unspecified variables. See *glob()* for signature.

irglob(**fs*) → Iterable[str]

Iterable, recursive glob over all unspecified variables. See *glob()* for signature.

parse(*path: str*, *use_data: bool = True*) → dict

Extract variables from a compiled path.

See parse to understand the amazing witchery that makes this possible!

https://pypi.org/project/parse/

Parameters

- **path** (*str*) – The path containing data to parse.
- **use_data** (*bool*) – Should we fill in the data we already have before parsing? This means fewer variables that need to be parsed. Set False if you do not wish to use the data.

translate(*path: str*, *to: str*, ***kw*) → P

Translate the paths to another pattern

property parents: _PathParents

A sequence of this path's logical parents.

absolute()

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

expanduser()

Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)

property parent

The logical parent of the path.

relative_to(**other*)

Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

resolve(*strict=False*)

Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

with_name(*name*)

Return a new path with the file name changed.

with_suffix(*suffix*)

Return a new path with the file suffix changed. If the path has no suffix, add given suffix. If the given suffix is an empty string, remove the suffix from the path.

class pathtrees.**Paths**(*paths: Dict[str, 'Path']*, *data: dict | None = None*)

A hierarchy of paths in your project.

You can arbitrarily nest them and it will join all of the keys leading down to that path. The value is the name that you can refer to it by.

```
# define your file structure.

# a common ML experiment structure (for me anyways)
paths = Paths.define('./logs', {
    '{log_id}': {
        'model.h5': 'model',
        'model_spec.pkl': 'model_spec',
        'plots': {
            'epoch_{step_name}': {
                '{plot_name}.png': 'plot',
                '': 'plot_dir'
            }
        },
        # a path join hack that gives you: log_dir > ./logs/{log_id}
        '', 'log_dir',
    }
})
paths.update(log_id='test1', step_name='epoch_100')

# get paths by name
paths.model  # logs/test1/model.h5
paths.model_spec  # logs/test1/model_spec.pkl
paths.plot  # logs/test1/plots/{step_name}/{plot_name}.png

# for example, a keras callback that saves a matplotlib plot every epoch
class MyCallback(Callback):
    def on_epoch_end(self, epoch, logs):
        # creates a copy of the path tree that has step_name=epoch
        epoch_paths = paths.specify(step_name=epoch)

        ...
        # save one plot
        plt.imsave(epoch_paths.plot.specify(plot_name='confusion_matrix'))
        ...
        # save another plot
        plt.imsave(epoch_paths.plot.specify(plot_name='auc'))

# you can glob over any missing data (e.g. step_name => '*')
# equivalent to: glob("logs/test1/plots/{step_name}/auc.png")
for path in paths.plot.specify(plot_name='auc').glob():
    print(path)
```

keys() → Iterable[str]

Iterate over path names in the tree.

add(*root=None*, *paths=None*) → Ps

Build paths from a directory spec.

Parameters

- **root** (*str*) – the root directory.
- **paths** (*dict*) – the directory structure.

Returns

The initialized Paths object

rjoinpath(*path*) → *Paths*

Give these paths a new root! Basically doing root / path for all paths in this tree. This is useful if you want to nest a folder inside another.py

relative_to(*path*) → *Paths*

Make these paths relative to another path! Basically doing path.relative_to(root) for all paths in this tree. Use this with `with_root` to change the root directory of the paths.

parse(*path*, *name: str*) → dict

Parse data from a formatted string (reverse of string format)

Parameters

- **path** (*str*) – the string to parse
- **name** (*str*) – the name of the path pattern to use.

property copy: *Paths*

Create a copy of a path tree and its paths.

update(**kw*) → Ps

Update specified data in place.

```
paths = pathtrees.tree({'{a}': aaa})
assert not paths.fully_specified
paths.update(a=5)
assert paths.fully_specified
assert paths.data['a'] == 5
```

specify(**kw*) → Ps

Creates a copy of the path tree then updates the copy's data.

```
paths = pathtrees.tree({'{a}': aaa})
paths2 = paths.specify(a=5)

assert not paths.fully_specified
assert paths2.fully_specified

assert 'a' not in paths.data
assert paths2.data['a'] == 5
```

Equivalent to:

```
paths.copy.update(**kw)
```

unspecify(**keys*, *inplace=False*, *children=True*) → *Paths*

Remove keys from paths dictionary.

```
paths = pathtrees.tree({'{a}': aaa})
paths.update(a=5)
assert paths.fully_specified
assert paths.data['a'] == 5

paths.unspecify('a')
assert not paths.fully_specified
assert 'a' not in paths.data
```

property fully_specified: bool

Are all paths fully specified?

```
paths = pathtrees.tree({'{a}': aaa})
assert not paths.fully_specified
paths.update(a=5)
assert paths.fully_specified
```

format(***kw*) → Dict[str, str]

Try to format all paths as strings. Raises Underspecified if data is missing.

Parameters
****kw** – additional data specified for formatting.

Returns
key is the name of the path, and the value is the formatted `pathlib.Path`.

Return type
dict

maybe_format(***kw*) → Dict[str, Union[str, *Path*]]

Return a dictionary where all fully specified paths are converted to strings and underspecified strings are left as Path objects.

Parameters
****kw** – additional data specified for formatting.

partial_format(***kw*) → Dict[str, str]

Return a dictionary where all paths are converted to strings and underspecified fields are left in for later formatting.

Parameters
****kw** – additional data specified for formatting.

Note: This is a code redesign from `path-tree`. I re-wrote it because that was one of my very first public Pypi projects and I'm not exactly proud of some of the design decisions I made.

On top of that! I found out that it breaks with Python 3.10. Which was the real reason. Crossing my fingers that I can get everything pushed out before I get a GitHub issue saying everything is broken!

This is an effort to turn an old, fragile, and about-to-break project into something that I might actually import into a new project!

The rename from `path-tree` to `pathtrees` is because I've always hated that the pip install name is not the same as the import name. That is a big pet peeve so I'll be glad to be rid of that in at least my own projects.

I threw this together (including docs) in a couple nights after work so it's still a WIP.

The code is mostly together, but the docs and examples need work. And there's a couple small quirks around things like: Should `path.format()` return a `pathlib.Path` or a `str`?

PYTHON MODULE INDEX

INDEX