kodiak Documentation

Release 0.1.0

Ezequiel Erbaro

Feb 08, 2023

Contents

Contents:

CHAPTER 1

Kodiak

1.1 Overview

Kodiak enhances your feature engineering workflow extracting common patterns so you can create more features faster.

Ex: You have the `writers` dataframe, where `born` is a datetime

name	born
Miguel de Cervantes	09-29-1547
William Shakespeare	04-23-1617

and you want to extract from the `born` column: `day`, `month` and `year` and create 3 new columns

name	born	born_month	born_day	born_year
Miguel de Cervantes	09-29-1547	9	29	1547
William Shakespeare	04-23-1617	4	23	1617

The simplest thing you could do in Pandas is:

```
writers["born_month"] = writers.born.map(lambda x: x.month)
writers["born_day"]   = writers.born.map(lambda x: x.day)
writers["born_year"]  = writers.born.map(lambda x: x.year)
```

With Kodiak you could get the same result in one line:

```
writers.gencol("born_{month,day,year}", "born", lambda x, y: getattr(x, y))
# or more succinctly
writers.gencol("born_{.month,.day,.year}", "born")
```

But, how does it work? Kodiak uses `"born_{month,day,year}"` as a template for the columns: `born_month`, `born_day` and `born_year` and passes `month`,`day` and `year` as arguments to a provided function that also has the current 'born' as an an argument, so you're basically doing:

```
for y in ['month', 'day', 'year']:
    writers["born_{}".format(y)] = writers.born.map(lambda x: getattr(x, y))
```

Kodiak does a lot of other things to boost your workflow, for that, see the *Basic Usage* and *Advanced Usage* sections

1.2 Installation

To install Kodiak, `cd` to the Kodiak folder and run the install command:

```
sudo python setup.py install
```

You can also install Kodiak from PyPI:

```
sudo pip install kodiak
```

1.3 Basic Usage

Kodiak main object is `KodiakDataFrame` an extension of `pandas.Dataframe` that provides the instance method `colgen` to create one or more columns. You create `KodiakDataFrames` the same way you create a `pandas.DataFrame`

`colgen` signature is: `colgen(newcols, col, colbuilder=None, enum=False, config=None)`

newcols has a double function, it works as a specification of the columns you want to create, and it also contains the values passed to `colbuilder`

```
# If you want to create the columns `first_name`, `last_name`
# and pass `first` and `last` as arguments to `colbuilder` you write
>>> newcols = "{first,last}_name"

# More complex patterns allowed
>>> groups = "col_{a,b}_{c,d}"

# Will create the columns: `col_a_c`, `col_b_d`
# The way `a,b` and `c,d` is combined can be configured
```

col is the *KodiakDataframe* column from where you'll extract information to create your new column/s

colbuilder is a function or lambda used to extract info from `col` and create the columns specified in `newcols` with the corresponding `col` instance and the `newcols` values. The signature of *colbuilder* is *colbuilder(x, y)* or *colbuilder(i, x, y)* *x* is an instance of the column passed in *col* and *y* is an argument extracted from *newcols*. The extra argument *i* is an index of the arguments.

config tweak Kodiak inner workings with your own config, see the dedicated section for more info

1.4 Advanced Usage

In this section we're going to describe the main components and concepts that are essential to Kodiak

1.4.1 Templating

The template language is minimal but has some extensions to help you:

Ranges

The range notation is `start:end:step`. Reverse ranges are permitted setting `end` bigger than `start`. `step` default value is `1`, and `start` is `0`, finally if `end` is absent, it'll be setted to `0` and you'll have a reversed range. Ranges are inclusive.

```
simple_range = "col_{1:3}" # -> col_1, col_2, col_3
step_range = "col_{:3:2}" # -> col_0, col_3
inverse_range = "col_{3:1}" # -> col_3,col_2,col_1
no_end = "2:" # -> col_2,col_1,col_0
```

Key-Value

If you want the column name and argument passed to the `colbuilder` to differ you can use key-values.

```
dataframe.gencol("{short=very_long_name}_col", "col", alambda)
# In this case the column name will be ``short_col`` but you'll pass
# ``very_long_name`` to ``alambda``

# key-value notation can be extended to more arguments:
dataframe.gencol("{k1=v1,k2=v2,k3=v3}_col", "col", alambda)
```

Warning: values are always interpreted as *strings* so in: `col_{k=1:5}` the value `1:5` is interpreted as `"1:5"` and not as a range, the same for `col_{k=[1,2,3]}` and any other object, also if you pass a number it will also be interpreted as string so you will need to convert it if you intend to use it as an `int`.

1.4.2 Transforms

Under the hood when you pass `newcols` to `gencol`, Kodiak builds an `OrderedDict` where it's keys are column names and it's values are tuples of `Match` objects -even if it's just one Match it's wrapped inside a tuple-

```
newcol = "{first,last}_name"
# will build
args_dict = {'first_name': (Match('first'),), 'last_name': (Match('last'),)}
```

`Transforms` are a way to pre-process the values and change them enriching the `Match` object with a payload as you will see in the `Default colbuilder` section.

So, if the values are `Match` objects, how is that when you write your `colbuilder` you deal with `strings`? Kodiak understands that if the `Match` object doesn't have a payload it's better to pass strings arguments to `colbuilder`, this behaviour can be controlled.

What's the use of `Match` objects and their `payload`? What're some examples of `Transforms`? The next section will answer this questions

1.4.3 Default colbuilder

As you can see in the `colgen` signature, `colbuilder` default argument is `None`, in special cases Kodiak can infer the `colbuilder` method, let's revisit the opening example.

```
writers.gencol("born_{.month,.day,.year}", "born")
```

The `colbuilder` in this case is inferred from the hint you gave Kodiak in the template: `.month`, prefixing `month` with a `.` indicates that you're referring to an attribute of `born`, so internally Kodiak builds a `colbuider` that extracts the `month` from a `born` instance. Another way of omitting the `colbuilder` is when you have an instance method:

```
# Notice the `!` after weekday
writers.gencol("born_{weekday!}", "born")
```

Warning: This hint only works for methods with no arguments, passing a method with one or more arguments will raise an error

How is that Kodiak infers the `colbuilder`? When the `newcols` are processed they go through a pipeline of `Transforms`, one of them: `PropertyTransform` detects that `.month` refers to an attribute and enriches de `Match` object hinting in the payload the corresponding `colbuilder`, that's why you don't need to pass the `colbuilder` argument. But what happens if you give a `colbuilder`? In this case, as the `Match` object has a `payload` instead of working with plain strings you will work with tuples of `Match` objects

Note: Kodiak will raise an exception when it can't figure out a default colbuilder

1.4.4 Enumerations

Sometimes you care about the position of the arguments not the exact value, in that case you can use the `enum` param or the implicit `enum` with a function or lambda of arity 3, the first argument will be the index starting at 0.

```
writers.gencol("{first=0,last=1}_name", "name", lambda x,y: x.split(" ")[int(y)])

# Another way with enum=True
writers.gencol("{first,last}_name", "name", lambda i,x,y: x.split(" ")[i], enum=True)

# Without enum=True but with a colbuilder with arity 3
writers.gencol("{first,last}_name", "name", lambda i,x,y: x.split(" ")[i])
```

1.5 Configuration

Almost everything is configurable in Kodiak, you could have a per-method configuration or system-wide config.

The `Config` object has the following customizable params:

parser Kodiak by default uses the `ArgsParser` class to parse `newcols`

match_transform data passed to the `colbuilder` could be transformed first, by default we use the `default_transform` pipeline, you could replace it with an array of `Transforms` objects.

new_col_combiner in the newcols template if you have `"col_{a,b}_{c,d}"`, this results in the columns: `"col_a_c"` and `"col_b,d"`, how the different groups `['a','b']` and `['c', 'd']` are combined is controlled with this param, currently we use the `zip` function, and you could replace it with a function with similar signature.

unpack Boolean Default True, when `newcols` is simple, `foo_{a,b}` instead of `foo_{.a,b!}` instead of passing to `colbuilder` tuples of `Match` objects we just pass individual items, `a`, `b`, so it's easier to build a `colbuilder` without having to unwrap the `Match` tuples

col_pair_combiner Once you have the arguments that you're going to pass to the `colbuilder` you can combine them in different ways, currently we use `product` from itertools, ie: the cartesian product between an element, ex: `event`, and the other n-columns, creating the following tuples:

```
[('event', 'day') , ('event', 'month'), ('event', 'year')]
```

You can replace this method with another with the same signature as `product`

Config can be accessed, modified and restored with:

```
>> import config
>> from config import cfg
>> config.options

# Global change on config

>> config.options["unpack"] = False
>> config.options["col_pair_combiner"] = zip

# Restoring one or more fields of the configuration
>> config.restore_default_config("col_pair_combiner")

# Restoring all the options
>> config.restore_default_config()

# With `base_config` or it's alias `cfg` you can create modified versions
# of the default config

>> dataframe.gencol("col_{a!,b!}","col", func, config=cfg(unpack=False))
```

CHAPTER 2

Installation

2.1 Stable release

To install kodiak, run this command in your terminal:

```
$ pip install kodiak
```

This is the preferred method to install kodiak, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for kodiak can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/alejandrodumas/kodiak
```

Or download the tarball:

```
$ curl  -OL https://github.com/alejandrodumas/kodiak/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use kodiak in a project:

```
import kodiak
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/alejandrodumas/kodiak/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

kodiak could always use more documentation, whether as part of the official kodiak docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/alejandrodumas/kodiak/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *kodiak* for local development.

1. Fork the *kodiak* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/kodiak.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv kodiak
$ cd kodiak/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 kodiak tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/alejandrodumas/kodiak/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_kodiak
```

CHAPTER 5

Credits

5.1 Development Lead

- Ezequiel Erbaro <eerbaro@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.0 (2017-08-15)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search