Parameterize Jobs Documentation

Release 0.1.1

ClimateImpactLab

Aug 05, 2019

Contents

Contents:

CHAPTER 1

Parameterize Jobs

`parameterize_jobs` is a lightweight, pure-python toolkit for concisely and clearly creating large, parameterized, mapped job specifications.

- Free software: MIT license
- Documentation: https://parameterize-jobs.readthedocs.io

1.1 Features

- Expand a job's dimensionality by multiplying `ComponentSet`, `Constant`, or `ParallelComponentSet` objects
- Extend the number of jobs by adding `ComponentSet`, `Constant`, or `ParallelComponentSet` objects
- Jobs are provided to functions as dictionaries of parameters
- The helper decorator `@expand_kwargs` turns these kwarg dictionaries into named argument calls
- Works seamlessly with many task running frameworks, including dask's *client.map* and profiling tools

1.2 TODOs

View and submit issues on the issues page.

1.3 Quickstart

`ComponentSet` objects are the base objects, and can be defined with any number of named iterables:

```
>>> import parameterize_jobs as pjs

>>> a = pjs.ComponentSet(a=range(5))
>>> a
<ComponentSet {'a': 5}>
```

These objects have defined lengths (if the provided iterable has a defined length), and can be indexed and iterated over:

```
>>> len(a)
5

>>> a[0]
{'a': 0}

>>> list(a)
[{'a': 0},
 {'a': 1},
 {'a': 2},
 {'a': 3},
 {'a': 4}]
```

Adding two `ComponentSet` objects extends the total job length

```
>>> a2 = pjs.ComponentSet(a=range(3))

>>> a+a2
<MultiComponentSet [{'a': 5}, {'a': 3}]>

>>> len(a+a2)
8

>>> list(a+a2)

[{'a': 0},
 {'a': 1},
 {'a': 2},
 {'a': 3},
 {'a': 4},
 {'a': 0},
 {'a': 1},
 {'a': 2}]
```

Multiplying two `ComponentSet` objects expands their dimensionality:

```
>>> b = pjs.ComponentSet(b=range(3))

>>> a*b
<ComponentSet {'a': 5, 'b': 3}>

>>> len(a*b)
15

>>> (a*b)[-1]
{'a': 4, 'b': 2}

>>> list(a*b)
[{'a': 0, 'b': 0},
```

(continues on next page)

(continued from previous page)

```
 {'a': 0, 'b': 1},
 {'a': 0, 'b': 2},
 {'a': 1, 'b': 0},
 {'a': 1, 'b': 1},
 {'a': 1, 'b': 2},
 {'a': 2, 'b': 0},
 {'a': 2, 'b': 1},
 {'a': 2, 'b': 2},
 {'a': 3, 'b': 0},
 {'a': 3, 'b': 1},
 {'a': 3, 'b': 2},
 {'a': 4, 'b': 0},
 {'a': 4, 'b': 1},
 {'a': 4, 'b': 2}]
```

These parameterized job specifications can be used in mappable jobs. The helper decorator `expand_kwargs` modifies a function to accept a dictionary and expands them into keyword arguments:

```
>>> @pjs.expand_kwargs
... def my_simple_func(a, b, c=1):
...     return a * b * c

>>> list(map(my_simple_func, a*b))
[0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 2, 4, 6, 8, 0, 3, 6, 9, 12]
```

Jobs do not have to be the combinatorial product of all components:

```
>>> ab1 = pjs.ComponentSet(a=[0, 1], b=[0, 1])
>>> ab2 = pjs.ComponentSet(a=[10, 11], b=[-1, 1])

>>> list(map(my_simple_func, ab1 + ab2))
[0, 0, 0, 1, -10, -11, 10, 11]
```

A `Constant` object is simply a `ComponentSet` object defined with single values passed as keyword arguments rather than iterables passed as keyword arguments:

```
>>> c = pjs.Constant(c=5)

>>> list(map(my_simple_func, (ab1 + ab2) * c))
[0, 0, 0, 5, -50, -55, 50, 55]
```

A `ParallelComponentSet` object is simply a `MultiComponentSet` object where each `Component` is a `Constant` object.

```
>>> pcs = pjs.ParallelComponentSet(a = [1, 2],
                                   b = [10, 20])

>>> list(map(my_simple_func, pcs))
[10, 40]
```

Arbitrarily complex combinations of ComponentSets can be created:

```
>>> c1 = pjs.Constant(c=1)
>>> c2 = pjs.Constant(c=2)

>>> list(map(my_simple_func, (ab1 + ab2) * c1 + (ab1 + ab2) * c2))
[0, 0, 0, 1, -10, -11, 10, 11, 0, 0, 0, 2, -20, -22, 20, 22]
```

Anything can be inside a `ComponentSet` iterable, including data, functions, or other objects:

```
>>> transforms = (
...     pjs.Constant(transform=lambda x: x, transform_name='linear')
...     + pjs.Constant(transform=lambda x: x**2, transform_name='quadratic'))
...

>>> fps = pjs.Constant(
...     read_pattern='source/my-fun-data_{year}.csv',
...     write_pattern='transformed/my-fun-data_{transform_name}_{year}.csv')

>>> years = pjs.ComponentSet(year=range(1980, 2018))

>>> @pjs.expand_kwargs
... def process_data(read_pattern, write_pattern, transform, transform_name, year):
...
...     df = pd.read_csv(read_pattern.format(year=year))
...
...     transformed = transform(df)
...
...     transformed.to_csv(
...         write_pattern.format(
...             transform_name=transform_name,
...             year=year))
...

>>> _ = list(map(process_data, transforms * fps * years))
```

This works seamlessly with dask's client.map to provide intuitive job parameterization:

```
>>> import dask.distributed as dd
>>> client = dd.LocalClient()
>>> futures = client.map(my_simple_func, (ab1 + ab2) * c1 + (ab1 + ab2) * c2)
>>> dd.progress(futures)
```

CHAPTER 2

Installation

2.1 Stable release

To install Parameterize Jobs, run this command in your terminal:

```
$ pip install parameterize_jobs
```

This is the preferred method to install Parameterize Jobs, 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 Parameterize Jobs can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ClimateImpactLab/parameterize_jobs
```

Or download the tarball:

```
$ curl  -OL https://github.com/ClimateImpactLab/parameterize_jobs/tarball/master
```

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

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

CHAPTER 3

Usage

To use Parameterize Jobs in a project:

```
In [1]: import parameterize_jobs as pj
```

3.1 The Component class

```
In [2]: component = Component([0, 50, 100])
In [3]: component
Out[3]:
<Component [0, 50, 100]>
```

components are essentially just a wrapper around whatever data you provided, which should be an iterable.

```
In [4]: component[0]
Out[4]:
0
```

```
In [5]: len(component)
Out[5]:
3
```

```
In [6]: list(component)
Out[6]:
[0, 50, 100]
```

Q: sweet. but why would we want that?
A: you don't. components are just a helper class. you want to use a `ComponentSet`!

3.2 The ComponentSet class

```
In [7]: cs = ComponentSet(a=range(5), b=['a', 'b', 'c', 'd'])
In [8]: cs
Out[8]:
<ComponentSet {a: 5, b: 4}>
```

A `ComponentSet` is sort of like itertools.product with some additional features:

- `ComponentSet` objects have a length if the constituent `Component` objects have lengths:

```
In [ 9]: cs = ComponentSet(a=range(5), b=['a', 'b', 'c', 'd'])
In [10]: len(cs)
Out[10]:
20
```

- `ComponentSet` objects can be positionally indexed:

```
In [11]: cs[0]
Out[11]:
{'a': 0, 'b': 'a'}

In [12]: cs[1]
Out[12]:
{'a': 0, 'b': 'b'}

In [13]: cs[-1]
Out[13]:
{'a': 4, 'b': 'd'}
```

This is all done without computing the full set of combinations. `ComponentSet` objects can be iterated over to retrieve all combinations:

```
In [14]: for c in cs:
    ...:     print(c)
    ...:
Out[14]:
{'a': 0, 'b': 'a'}
{'a': 0, 'b': 'b'}
{'a': 0, 'b': 'c'}
{'a': 0, 'b': 'd'}
{'a': 1, 'b': 'a'}
...
```

You can see the performance implications of not producing the full product by comparing `len(cs)` with `len(list(cs))`:

```
In [15]: %%timeit
    ...:
    ...: len(ComponentSet(a=range(100), b=range(100), c=range(100)))
Out[15]:
6.89 µs ± 265 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [16]: %%timeit
    ...:
    ...: len(list(ComponentSet(a=range(100), b=range(100), c=range(100))))
```

(continues on next page)

(continued from previous page)

```
Out[16]:
1.35 s ± 41.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Q: that's cool. can we do anything else with these?
A: Yeah! You can do math!

```
In [17]: a = ComponentSet(a=range(5), b=list('abcd'))
In [18]: b = ComponentSet(c=range(0, 101, 50))
In [19]: c = a * b
```

whoa. what is this?

```
In [20]: c
Out[20]:
<ComponentSet {a: 5, b: 4, c: 3}>
```

3.3 Multiplication: adding a new dimension

When you multiply two `ComponentSet` objects, the constituent `Component` objects are combined into a new `ComponentSet` with the outer product of the constituent components.

```
In [21]: a = ComponentSet(a=range(5), b=list('abcd'))
In [22]: b = ComponentSet(c=range(0, 101, 50))
In [23]: c = a * b
```

```
In [24]: len(c)
Out[24]:
60
```

```
In [25]: c[0]
Out[25]:
{'a': 0, 'b': 'a', 'c': 0}
```

```
In [26]: c[-1]
Out[26]:
{'a': 4, 'b': 'd', 'c': 100}
```

```
In [27]: list(c)
Out[27]:
[{'a': 0, 'b': 'a', 'c': 0},
 {'a': 0, 'b': 'a', 'c': 50},
 {'a': 0, 'b': 'a', 'c': 100},
 {'a': 0, 'b': 'b', 'c': 0},
 {'a': 0, 'b': 'b', 'c': 50},
 {'a': 0, 'b': 'b', 'c': 100},
 {'a': 0, 'b': 'c', 'c': 0},
 {'a': 0, 'b': 'c', 'c': 50},
 {'a': 0, 'b': 'c', 'c': 100},
 {'a': 0, 'b': 'd', 'c': 0},
```

(continues on next page)

(continued from previous page)

```
{'a': 0, 'b': 'd', 'c': 50},
{'a': 0, 'b': 'd', 'c': 100},
{'a': 1, 'b': 'a', 'c': 0},
{'a': 1, 'b': 'a', 'c': 50},
{'a': 1, 'b': 'a', 'c': 100},
{'a': 1, 'b': 'b', 'c': 0},
{'a': 1, 'b': 'b', 'c': 50},
{'a': 1, 'b': 'b', 'c': 100},
{'a': 1, 'b': 'c', 'c': 0},
{'a': 1, 'b': 'c', 'c': 50},
{'a': 1, 'b': 'c', 'c': 100},
{'a': 1, 'b': 'd', 'c': 0},
{'a': 1, 'b': 'd', 'c': 50},
{'a': 1, 'b': 'd', 'c': 100},
{'a': 2, 'b': 'a', 'c': 0},
{'a': 2, 'b': 'a', 'c': 50},
{'a': 2, 'b': 'a', 'c': 100},
{'a': 2, 'b': 'b', 'c': 0},
{'a': 2, 'b': 'b', 'c': 50},
{'a': 2, 'b': 'b', 'c': 100},
{'a': 2, 'b': 'c', 'c': 0},
{'a': 2, 'b': 'c', 'c': 50},
{'a': 2, 'b': 'c', 'c': 100},
{'a': 2, 'b': 'd', 'c': 0},
{'a': 2, 'b': 'd', 'c': 50},
{'a': 2, 'b': 'd', 'c': 100},
{'a': 3, 'b': 'a', 'c': 0},
{'a': 3, 'b': 'a', 'c': 50},
{'a': 3, 'b': 'a', 'c': 100},
{'a': 3, 'b': 'b', 'c': 0},
{'a': 3, 'b': 'b', 'c': 50},
{'a': 3, 'b': 'b', 'c': 100},
{'a': 3, 'b': 'c', 'c': 0},
{'a': 3, 'b': 'c', 'c': 50},
{'a': 3, 'b': 'c', 'c': 100},
{'a': 3, 'b': 'd', 'c': 0},
{'a': 3, 'b': 'd', 'c': 50},
{'a': 3, 'b': 'd', 'c': 100},
{'a': 4, 'b': 'a', 'c': 0},
{'a': 4, 'b': 'a', 'c': 50},
{'a': 4, 'b': 'a', 'c': 100},
{'a': 4, 'b': 'b', 'c': 0},
{'a': 4, 'b': 'b', 'c': 50},
{'a': 4, 'b': 'b', 'c': 100},
{'a': 4, 'b': 'c', 'c': 0},
{'a': 4, 'b': 'c', 'c': 50},
{'a': 4, 'b': 'c', 'c': 100},
{'a': 4, 'b': 'd', 'c': 0},
{'a': 4, 'b': 'd', 'c': 50},
{'a': 4, 'b': 'd', 'c': 100}]
```

3.4 Addition: creating a new MultiComponentSet

Adding two `ComponentSet` objects can be used when combining two objects with similar dimensions but different labels within those dimensions.

For example, the following ComponentSets are both indexed by `a` and `b`, but there is no overlap *along* these dimensions:

```
In [28]:  = ComponentSet(a=range(5), b=list('abcd'))
In [29]: b = ComponentSet(a=range(10, 15), b=list('wxyz'))

In [30]: ab = a + b
```

```
In [31]: ab
Out[31]:
<MultiComponentSet [{a: 5, b: 4}, {a: 5, b: 4}]>
```

Instead of adding a new dimension or extending each dimension, addition creates a new type of object, which is essentially a concatenated list of `ComponentSet` objects

The `MultiComponentSet` has a length equal to the sum of the lengths of the constituent `Componentset` objects, and on iteration, the result simply proceeds thorugh each of the constituent ComponentSets.

```
In [32]: len(a), len(b)
Out[32]:
(20, 20)
```

```
In [33]: len(ab)
Out[33]:
40
```

```
In [34]: list(ab)
Out[34]:
[{'a': 0, 'b': 'a'},
 {'a': 0, 'b': 'b'},
 {'a': 0, 'b': 'c'},
 {'a': 0, 'b': 'd'},
 {'a': 1, 'b': 'a'},
 {'a': 1, 'b': 'b'},
 {'a': 1, 'b': 'c'},
 {'a': 1, 'b': 'd'},
 {'a': 2, 'b': 'a'},
 {'a': 2, 'b': 'b'},
 {'a': 2, 'b': 'c'},
 {'a': 2, 'b': 'd'},
 {'a': 3, 'b': 'a'},
 {'a': 3, 'b': 'b'},
 {'a': 3, 'b': 'c'},
 {'a': 3, 'b': 'd'},
 {'a': 4, 'b': 'a'},
 {'a': 4, 'b': 'b'},
 {'a': 4, 'b': 'c'},
 {'a': 4, 'b': 'd'},
 {'a': 10, 'b': 'w'},
 {'a': 10, 'b': 'x'},
 {'a': 10, 'b': 'y'},
 {'a': 10, 'b': 'z'},
 {'a': 11, 'b': 'w'},
 {'a': 11, 'b': 'x'},
 {'a': 11, 'b': 'y'},
 {'a': 11, 'b': 'z'},
 {'a': 12, 'b': 'w'},
```

(continues on next page)

(continued from previous page)

```
 {'a': 12, 'b': 'x'},
 {'a': 12, 'b': 'y'},
 {'a': 12, 'b': 'z'},
 {'a': 13, 'b': 'w'},
 {'a': 13, 'b': 'x'},
 {'a': 13, 'b': 'y'},
 {'a': 13, 'b': 'z'},
 {'a': 14, 'b': 'w'},
 {'a': 14, 'b': 'x'},
 {'a': 14, 'b': 'y'},
 {'a': 14, 'b': 'z'}]
```

3.5 Math with MultiComponentSets

Works just like you'd expect! Multiplication applies to each consitutent ComponentSet, Addition nests MultiComponentSets.

```
In [35]: d1 = ComponentSet(d=['first', 'second'])
```

```
In [36]: ab
Out[36]:
<MultiComponentSet [{a: 5, b: 4}, {a: 5, b: 4}]>
```

```
In [37]: ab*d1
Out[37]:
<MultiComponentSet [{a: 5, b: 4, d: 2}, {a: 5, b: 4, d: 2}]>
```

```
In [38]: d2 = ComponentSet(d=['third', 'fourth'])
```

```
In [39]: e = ComponentSet(e=['another'])
```

```
In [40]: abdde = ((ab * d1) + (ab * d2)) * e
In [41]: abdde
Out[41]:
<MultiComponentSet [[{a: 5, b: 4, d: 2, e: 1}, {a: 5, b: 4, d: 2, e: 1}], [{a: 5, b:␣
↪4, d: 2, e: 1}, {a: 5, b: 4, d: 2, e: 1}]]>
```

```
In [42]: len(abdde)
Out[42]:
160
```

```
In [43]: abdde[0]
Out[43]:
{'a': 0, 'b': 'a', 'd': 'first', 'e': 'another'}
```

```
In [44]: abdde[-1]
Out[44]:
{'a': 14, 'b': 'z', 'd': 'fourth', 'e': 'another'}
```

3.6 ComponentSets with exhaustible generators

ComponentSet objects can be used with generators, but the length and indexing features will not work:

```
In [45]: with_generator = ComponentSet(gen=(i for i in [1, 2, 3, 4]))
```

```
In [46]: with_generator
Out[46]:
<ComponentSet {gen: (...)}>
```

The following would return an error:

```
In [47]: len(with_generator)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-028f83238a52> in <module>
----> 1 len(with_generator)

<ipython-input-1-2d6ab0f3cd2e> in __len__(self)
     69
     70     def __len__(self):
---> 71         return product(map(len, self._sets.values()))
     72
     73     def __iter__(self):

<ipython-input-1-2d6ab0f3cd2e> in product(arr)
      4
      5 def product(arr):
----> 6     return reduce(lambda x, y: x * y, arr, 1)
      7
      8 def cumprod(arr):

<ipython-input-1-2d6ab0f3cd2e> in __len__(self)
     20
     21     def __len__(self):
---> 22         return len(self._values)
     23
     24     def __iter__(self):

TypeError: object of type 'generator' has no len()
```

but this can still be iterated over:

```
In [48]: list(with_generator)
Out[48]:
[{'gen': 1}, {'gen': 2}, {'gen': 3}, {'gen': 4}]
```

as it's a generator, the list is exhausted on use:

```
In [49]: list(with_generator)
Out[49]:
[]
```

3.7 Use with dask

ComponentSet and MultiComponentSet objects can be used with many queueing libraries, including dask

```
In [50]: import dask.distributed as dd
```

```
In [51]: client = dd.Client()
In [52]: client
```

```
In [53]: def do_something(kwargs):
    ...:     import time
    ...:     import random
    ...:     time.sleep(random.random())
    ...:     return str(kwargs)
```

```
In [54]: futures = client.map(do_something, abdde)
In [55]: dd.progress(futures)
Out[55]:
VBox()
```

3.8 A real-world example

parameterizing operations over multiple incompatible climate model, year, and scenario combinations

Global climate model outputs from CMIP5 simulations typically have an incompatible set of historical and projection years, ensemble members, and even models, as some models are run with some scenario and ensemble combinations, and others do not. At the same time, you may wish to do the same operation across all the existing model years, and would like to manage the runs with a single job generator.

This can be easily handled by building a MultiComponentset:

```
In [56]: hist = Constant(rcp='historical', model='obs')
In [57]: hist_years = ComponentSet(year=list(range(1950, 2006)))
```

```
In [58]: rcp45 = ComponentSet(
    ...:     rcp=['rcp45'],
    ...:     model=(
    ...:         ['ACCESS1-0', 'CCSM4']
    ...:         + ['pattern{}'.format(i) for i in [1, 2, 3, 5, 6, 27, 28, 29, 30, 31,
↪ 32]]))

In [59]: rcp85 = ComponentSet(
    ...:     rcp=['rcp85'],
    ...:     model=(
    ...:         ['ACCESS1-0', 'CCSM4']
    ...:         + ['pattern{}'.format(i) for i in [1, 2, 3, 4, 5, 6, 28, 29, 30, 31,␣
↪32, 33]]))

In [60]: proj_years = ComponentSet(year=list(range(2006, 2100)))
```

Jobs can also be added into the parameterization

```
In [61]: days_under = Constant(func = lambda x, thresh: x <= thresh, threshold=32)
In [62]: days_over = ComponentSet(func = [lambda x, thresh: x >= thresh],␣
↪threshold=[90, 95])
```

The entire job set is the sum of valid (model * model years), the entire set of which is run for each job specification:

```
In [63]: runs = ((hist * hist_years) + ((rcp45 + rcp85) * proj_years)) * (days_under␣
↪+ days_over)
```

```
In [64]: runs
Out[64]:
<MultiComponentSet [[{rcp: 1, model: 1, year: 56, func: 1, threshold: 1}, {rcp: 1,␣
↪model: 1, year: 56, func: 1, threshold: 2}], [[{rcp: 1, model: 13, year: 94, func:␣
↪1, threshold: 1}, {rcp: 1, model: 13, year: 94, func: 1, threshold: 2}], [{rcp: 1,␣
↪model: 14, year: 94, func: 1, threshold: 1}, {rcp: 1, model: 14, year: 94, func: 1,␣
↪threshold: 2}]]]>
```

```
In [65]: len(runs)
Out[65]:
7782
```

The different job specifications can be examined to make sure the job was built the way you expect:

```
In [66]: runs[0]
Out[66]:
{'rcp': 'historical',
 'model': 'obs',
 'year': 1950,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [67]: runs[55]
Out[67]:
{'rcp': 'historical',
 'model': 'obs',
 'year': 2005,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [68]: runs[56]
Out[68]:
{'rcp': 'historical',
 'model': 'obs',
 'year': 1950,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 90}
```

```
In [69]: runs[167]
Out[69]:
{'rcp': 'historical',
 'model': 'obs',
 'year': 2005,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 95}
```

```
In [70]: runs[168]
Out[70]:
{'rcp': 'rcp45',
 'model': 'ACCESS1-0',
 'year': 2006,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [71]: runs[261]
Out[71]:
{'rcp': 'rcp45',
 'model': 'ACCESS1-0',
 'year': 2099,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [72]: runs[262]
Out[72]:
{'rcp': 'rcp45',
 'model': 'CCSM4',
 'year': 2006,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [73]: runs[3833]
Out[73]:
{'rcp': 'rcp45',
 'model': 'pattern32',
 'year': 2099,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 95}
```

```
In [74]: runs[3834]
Out[74]:
{'rcp': 'rcp85',
 'model': 'ACCESS1-0',
 'year': 2006,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 32}
```

```
In [75]: runs[-1]
Out[75]:
{'rcp': 'rcp85',
 'model': 'pattern33',
 'year': 2099,
 'func': <function __main__.<lambda>(x, thresh)>,
 'threshold': 95}
```

This entire set can be run using a single call

```
In [76]: def do_something_fast(kwargs):
    ...:     return str(kwargs)
```

```
In [77]: futures = client.map(do_something_fast, runs)
In [78]: dd.progress(futures)
```

(continues on next page)

(continued from previous page)

```
Out[78]:
VBox()
```

```
In [79]: client.gather(futures[-1])
Out[79]:
"{'rcp': 'rcp85', 'model': 'pattern33', 'year': 2099, 'func': <function <lambda> at␣
↪0x10f4c9400>, 'threshold': 95}"
```

CHAPTER 4

parameterize_jobs

4.1 parameterize_jobs package

4.1.1 Submodules

4.1.2 parameterize_jobs.parameterize_jobs module

class parameterize_jobs.parameterize_jobs.**Component**(*values*)
Bases: object

class parameterize_jobs.parameterize_jobs.**ComponentSet**(***kwargs*)
Bases: object

Indexable combinatorial product job specification

class parameterize_jobs.parameterize_jobs.**Constant**(***kwargs*)
Bases: *parameterize_jobs.parameterize_jobs.ComponentSet*

A ComponentSet where each iterable has only one element

class parameterize_jobs.parameterize_jobs.**MultiComponentSet**(*components*)
Bases: object

A list of multiple ComponentSet objects

class parameterize_jobs.parameterize_jobs.**ParallelComponentSet**(***kwargs*)
Bases: *parameterize_jobs.parameterize_jobs.MultiComponentSet*

A MultiComponentSet object created by multiple lists of the same length, where each job will take the nth element of each list

parameterize_jobs.parameterize_jobs.**expand_kwargs**(*func*)
Decorator to expand an kwargs in function calls

Parameters **func** (*function*) – Function to have arguments expanded. Func can have any number of keyword arguments.

Returns **wrapped** – Wrapped version of `func` which accepts a single `kwargs` dict.

Return type function

Examples

```
>>> @expand_kwargs
... def my_func(a, b, exp=1):
...     return (a * b)**exp
...
>>> my_func({'a': 2, 'b': 3})
6
>>> my_func({'a': 2, 'b': 3, 'exp': 2})
36
```

4.1.3 Module contents

Top-level package for Parameterize Jobs.

class parameterize_jobs.**Component**(*values*)
Bases: object

class parameterize_jobs.**ComponentSet**(***kwargs*)
Bases: object

Indexable combinatorial product job specification

class parameterize_jobs.**MultiComponentSet**(*components*)
Bases: object

A list of multiple ComponentSet objects

class parameterize_jobs.**Constant**(***kwargs*)
Bases: *parameterize_jobs.parameterize_jobs.ComponentSet*

A ComponentSet where each iterable has only one element

class parameterize_jobs.**ParallelComponentSet**(***kwargs*)
Bases: *parameterize_jobs.parameterize_jobs.MultiComponentSet*

A MultiComponentSet object created by multiple lists of the same length, where each job will take the nth element of each list

parameterize_jobs.**expand_kwargs**(*func*)
Decorator to expand an kwargs in function calls

Parameters **func** (*function*) – Function to have arguments expanded. Func can have any number of keyword arguments.

Returns **wrapped** – Wrapped version of `func` which accepts a single `kwargs` dict.

Return type function

Examples

```
>>> @expand_kwargs
... def my_func(a, b, exp=1):
...     return (a * b)**exp
...
>>> my_func({'a': 2, 'b': 3})
6
>>> my_func({'a': 2, 'b': 3, 'exp': 2})
36
```

CHAPTER 5

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:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/ClimateImpactLab/parameterize_jobs/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.

5.1.2 Fix Bugs

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

5.1.3 Implement Features

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

5.1.4 Write Documentation

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

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ClimateImpactLab/parameterize_jobs/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 :)

5.2 Get Started!

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

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

```
$ git clone git@github.com:your_name_here/parameterize_jobs.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 parameterize_jobs
$ cd parameterize_jobs/
$ 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 parameterize_jobs tests
$ python setup.py test or pytest
$ 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.

5.3 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/ClimateImpactLab/parameterize_jobs/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ pytest tests.test_parameterize_jobs
```

CHAPTER 6

History

6.1 0.1.0 (2018-11-30)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index