patsy Documentation

Release 0.2.0

Nathaniel J. Smith

August 10, 2013

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW

> *"It's only a model."*

patsy is a Python package for describing statistical models (especially linear models, or models that have a linear component) and building design matrices. It is closely inspired by and compatible with the formula mini-language used in R and S.

For instance, if we have some variable *y*, and we want to regress it against some other variables *x*, *a*, *b*, and the interaction of *a* and *b*, then we simply write:

```
patsy.dmatrices("y ~ x + a + b + a:b", data)
```

and Patsy takes care of building appropriate matrices. Furthermore, it:

- Allows data transformations to be specified using arbitrary Python code: instead of `x`, we could have written `log(x)`, `(x > 0)`, or even `log(x) if x > 1e-5 else log(1e-5)`,
- Provides a range of convenient options for coding categorical variables, including automatic detection and removal of redundancies,
- Knows how to apply 'the same' transformation used on original data to new data, even for tricky transformations like centering or standardization (critical if you want to use your model to make predictions),
- Has an incremental mode to handle data sets which are too large to fit into memory at one time,
- Provides a language for symbolic, human-readable specification of linear constraint matrices,
- Has a thorough test suite (>97% statement coverage) and solid underlying theory, allowing it to correctly handle corner cases that even R gets wrong, and
- Features a simple API for integration into statistical packages.

What Patsy *won't* do is, well, statistics — it just lets you describe models in general terms. It doesn't know or care whether you ultimately want to do linear regression, time-series analysis, or fit a forest of decision trees, and it certainly won't do any of those things for you — it just gives a high-level language for describing which factors you want your underlying model to take into account. It's not suitable for implementing arbitrary non-linear models from scratch; for that, you'll be better off with something like Theano, SymPy, or just plain Python. But if you're using a statistical package that requires you to provide a raw model matrix, then you can use Patsy to painlessly construct that model matrix; and if you're the author of a statistics package, then I hope you'll consider integrating Patsy as part of your front-end.

Patsy's goal is to become the standard high-level interface to describing statistical models in Python, regardless of what particular model or library is being used underneath.

1.1 Download

The current release may be downloaded from the Python Package index at

> http://pypi.python.org/pypi/patsy/

Or the latest *development version* may be found in our Git repository:

```
git clone git://github.com/pydata/patsy.git
```

1.2 Requirements

Installing `patsy` requires:

- Python (version 2.4 or later; Python 3 is fully supported)
- NumPy

1.3 Installation

If you have `pip` installed, then a simple

```
pip install --upgrade patsy
```

should get you the latest version. Otherwise, download and unpack the source distribution, and then run

```
python setup.py install
```

1.4 Contact

Post your suggestions and questions directly to the pydata mailing list (pydata@googlegroups.com, gmane archive), or to our bug tracker. You could also contact Nathaniel J. Smith directly, but really the mailing list is almost always a better bet, because more people will see your query and others will be able to benefit from any answers you get.

1.5 License

2-clause BSD. See the file COPYING for details.

1.6 Users

We currently know of the following projects using Patsy to provide a high-level interface to their statistical code:

- Statsmodels
- *(your project here!)*

CHAPTER

TWO

QUICKSTART

If you prefer to learn by diving in and getting your feet wet, then here are some cut-and-pasteable examples to play with.

First, let’s import stuff and get some data to work with:

```
In [1]: import numpy as np

In [2]: from patsy import dmatrices, dmatrix, demo_data

In [3]: data = demo_data("a", "b", "x1", "x2", "y")
```

`demo_data()` gives us a mix of categorical and numerical variables:

```
In [1]: data
 Out[1]:
{’a’: [’a1’, ’a1’, ’a2’, ’a2’, ’a1’, ’a1’, ’a2’, ’a2’],
 ’b’: [’b1’, ’b2’, ’b1’, ’b2’, ’b1’, ’b2’, ’b1’, ’b2’],
 ’x1’: array([ 1.76405235,  0.40015721,  0.97873798,  2.2408932 ,  1.86755799,
       -0.97727788,  0.95008842, -0.15135721]),
 ’x2’: array([-0.10321885,  0.4105985 ,  0.14404357,  1.45427351,  0.76103773,
        0.12167502,  0.44386323,  0.33367433]),
 ’y’: array([ 1.49407907, -0.20515826,  0.3130677 , -0.85409574, -2.55298982,
        0.6536186 ,  0.8644362 , -0.74216502])}
```

Of course Patsy doesn’t much care what sort of object you store your data in, so long as it can be indexed like a Python dictionary, `data[varname]`. You may prefer to store your data in a pandas DataFrame, or a numpy record array... whatever makes you happy.

Now, let’s generate design matrices suitable for regressing `y` onto `x1` and `x2`.

```
In [1]: dmatrices("y ~ x1 + x2", data)
 Out[1]:
(DesignMatrix with shape (8, 1)
         y
   1.49408
  -0.20516
   0.31307
  -0.85410
  -2.55299
   0.65362
   0.86444
  -0.74217
  Terms:
    ’y’ (column 0),
 DesignMatrix with shape (8, 3)
```

```
   Intercept         x1        x2
           1    1.76405  -0.10322
           1    0.40016   0.41060
           1    0.97874   0.14404
           1    2.24089   1.45427
           1    1.86756   0.76104
           1   -0.97728   0.12168
           1    0.95009   0.44386
           1   -0.15136   0.33367
   Terms:
     'Intercept' (column 0)
     'x1' (column 1)
     'x2' (column 2))
```

The return value is a Python tuple containing two DesignMatrix objects, the first representing the left-hand side of our formula, and the second representing the right-hand side. Notice that an intercept term was automatically added to the right-hand side. These are just ordinary numpy arrays with some extra metadata and a fancy __repr__ method attached, so we can pass them directly to a regression function like `np.linalg.lstsq()`:

```
In [1]: outcome, predictors = dmatrices("y ~ x1 + x2", data)

In [2]: betas = np.linalg.lstsq(predictors, outcome)[0].ravel()

In [3]: for name, beta in zip(predictors.design_info.column_names, betas):
   ...:     print("%s: %s" % (name, beta))
   ...:
Intercept: 0.579662344123
x1: 0.0885991903554
x2: -1.76479205551
```

Of course the resulting numbers aren't very interesting, since this is just random data.

If you just want the design matrix alone, without the `y` values, use `dmatrix()` and leave off the `y ~` part at the beginning:

```
In [1]: dmatrix("x1 + x2", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept         x1        x2
          1    1.76405  -0.10322
          1    0.40016   0.41060
          1    0.97874   0.14404
          1    2.24089   1.45427
          1    1.86756   0.76104
          1   -0.97728   0.12168
          1    0.95009   0.44386
          1   -0.15136   0.33367
  Terms:
    'Intercept' (column 0)
    'x1' (column 1)
    'x2' (column 2)
```

We'll use dmatrix for the rest of the examples, since seeing the outcome matrix over and over would get boring. This matrix's metadata is stored in an extra attribute called `.design_info`, which is a `DesignInfo` object you can explore at your leisure:

```
In [1]: d = dmatrix("x1 + x2", data)

In [2]: d.design_info.<TAB>
```

```
d.design_info.builder               d.design_info.slice
d.design_info.column_name_indexes   d.design_info.term_name_slices
d.design_info.column_names          d.design_info.term_names
d.design_info.describe              d.design_info.term_slices
d.design_info.linear_constraint     d.design_info.terms
```

Usually the intercept is useful, but if we don't want it we can get rid of it:

```
In [1]: dmatrix("x1 + x2 - 1", data)
 Out[1]:
DesignMatrix with shape (8, 2)
        x1        x2
   1.76405  -0.10322
   0.40016   0.41060
   0.97874   0.14404
   2.24089   1.45427
   1.86756   0.76104
  -0.97728   0.12168
   0.95009   0.44386
  -0.15136   0.33367
  Terms:
    'x1' (column 0)
    'x2' (column 1)
```

We can transform variables using arbitrary Python code:

```
In [1]: dmatrix("x1 + np.log(x2 + 10)", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept        x1  np.log(x2 + 10)
          1   1.76405          2.29221
          1   0.40016          2.34282
          1   0.97874          2.31689
          1   2.24089          2.43836
          1   1.86756          2.37593
          1  -0.97728          2.31468
          1   0.95009          2.34601
          1  -0.15136          2.33541
  Terms:
    'Intercept' (column 0)
    'x1' (column 1)
    'np.log(x2 + 10)' (column 2)
```

Notice that *np.log* is being pulled out of the environment where `dmatrix()` was called – *np.log* is accessible because we did `import numpy as np` up above. Any functions or variables that you could reference when calling `dmatrix()` can also be used inside the formula passed to `dmatrix()`. For example:

```
In [1]: new_x2 = data["x2"] * 100

In [2]: dmatrix("new_x2")
 Out[2]:
DesignMatrix with shape (8, 2)
  Intercept     new_x2
          1  -10.32189
          1   41.05985
          1   14.40436
          1  145.42735
          1   76.10377
          1   12.16750
          1   44.38632
          1   33.36743
  Terms:
    'Intercept' (column 0)
    'new_x2' (column 1)
```

Patsy has some transformation functions "built in", that are automatically accessible to your code:

```
In [1]: dmatrix("center(x1) + standardize(x2)", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept  center(x1)  standardize(x2)
          1     0.87995         -1.21701
          1    -0.48395         -0.07791
          1     0.09463         -0.66885
          1     1.35679          2.23584
          1     0.98345          0.69899
          1    -1.86138         -0.71844
          1     0.06598         -0.00417
          1    -1.03546         -0.24845
  Terms:
    'Intercept' (column 0)
    'center(x1)' (column 1)
    'standardize(x2)' (column 2)
```

See patsy.builtins for a complete list of functions made available to formulas. You can also define your own transformation functions in the ordinary Python way:

```
In [1]: def double(x):
   ...:      return 2 * x
   ...:
```

Arithmetic transformations are also possible, but you'll need to "protect" them by wrapping them in `I()`, so that Patsy knows that you really do want + to mean addition:

```
In [1]: dmatrix("I(x1 + x2)", data)  # compare to "x1 + x2"
 Out[1]:
DesignMatrix with shape (8, 2)
  Intercept  I(x1 + x2)
          1     1.66083
          1     0.81076
          1     1.12278
          1     3.69517
          1     2.62860
          1    -0.85560
          1     1.39395
          1     0.18232
  Terms:
    'Intercept' (column 0)
    'I(x1 + x2)' (column 1)
```

Note that while Patsy goes to considerable efforts to take in data represented using different Python data types and convert them into a standard representation, all this work happens *after* any transformations you perform as part of your formula. So, for example, if your data is in the form of numpy arrays, "+" will perform element-wise addition, but if it is in standard Python lists, it will perform concatentation:

```
In [1]: dmatrix("I(x1 + x2)", {"x1": np.array([1, 2, 3]), "x2": np.array([4, 5, 6])})
 Out[1]:
DesignMatrix with shape (3, 2)
```

```
  Intercept  I(x1 + x2)
          1           5
          1           7
          1           9
  Terms:
    'Intercept' (column 0)
    'I(x1 + x2)' (column 1)

In [2]: dmatrix("I(x1 + x2)", {"x1": [1, 2, 3], "x2": [4, 5, 6]})
 Out[2]:
DesignMatrix with shape (6, 2)
  Intercept  I(x1 + x2)
          1           1
          1           2
          1           3
          1           4
          1           5
          1           6
  Terms:
    'Intercept' (column 0)
    'I(x1 + x2)' (column 1)
```

Patsy becomes particularly useful when you have categorical data. If you use a predictor that has a categorical type (e.g. strings or bools), it will be automatically coded. Patsy automatically chooses an appropriate way to code categorical data to avoid producing a redundant, overdetermined model.

If there is just one categorical variable alone, the default is to dummy code it:

```
In [1]: dmatrix("0 + a", data)
 Out[1]:
DesignMatrix with shape (8, 2)
  a[a1]  a[a2]
      1      0
      1      0
      0      1
      0      1
      1      0
      1      0
      0      1
      0      1
  Terms:
    'a' (columns 0:2)
```

But if you did that and put the intercept back in, you'd get a redundant model. So if the intercept is present, Patsy uses a reduced-rank contrast code (treatment coding by default):

```
In [1]: dmatrix("a", data)
 Out[1]:
DesignMatrix with shape (8, 2)
  Intercept  a[T.a2]
          1        0
          1        0
          1        1
          1        1
          1        0
          1        0
          1        1
          1        1
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
```

The `T.` notation is there to remind you that these columns are treatment coded.

Interactions are also easy – they represent the cartesian product of all the factors involved. Here's a dummy coding of each *combination* of values taken by `a` and `b`:

```
In [1]: dmatrix("0 + a:b", data)
 Out[1]:
DesignMatrix with shape (8, 4)
  a[a1]:b[b1]  a[a2]:b[b1]  a[a1]:b[b2]  a[a2]:b[b2]
            1            0            0            0
            0            0            1            0
            0            1            0            0
            0            0            0            1
            1            0            0            0
            0            0            1            0
            0            1            0            0
            0            0            0            1
  Terms:
    'a:b' (columns 0:4)
```

But interactions also know how to use contrast coding to avoid redundancy. If you have both main effects and interactions in a model, then Patsy goes from lower-order effects to higher-order effects, adding in just enough columns to produce a well-defined model. The result is that each set of columns measures the *additional* contribution of this effect – just what you want for a traditional ANOVA:

```
In [1]: dmatrix("a + b + a:b", data)
 Out[1]:
DesignMatrix with shape (8, 4)
  Intercept  a[T.a2]  b[T.b2]  a[T.a2]:b[T.b2]
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
    'b' (column 2)
    'a:b' (column 3)
```

Since this is so common, there's a convenient short-hand:

```
In [1]: dmatrix("a*b", data)
 Out[1]:
DesignMatrix with shape (8, 4)
  Intercept  a[T.a2]  b[T.b2]  a[T.a2]:b[T.b2]
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
  Terms:
    ’Intercept’ (column 0)
    ’a’ (column 1)
    ’b’ (column 2)
    ’a:b’ (column 3)
```

Of course you can use *other coding schemes* too (or even *define your own*). Here’s `orthogonal polynomial coding`:

```
In [1]: dmatrix("C(c, Poly)", {"c": ["c1", "c1", "c2", "c2", "c3", "c3"]})
 Out[1]:
DesignMatrix with shape (6, 3)
  Intercept  C(c, Poly).Linear  C(c, Poly).Quadratic
          1           -0.70711               0.40825
          1           -0.70711               0.40825
          1           -0.00000              -0.81650
          1           -0.00000              -0.81650
          1            0.70711               0.40825
          1            0.70711               0.40825
  Terms:
    ’Intercept’ (column 0)
    ’C(c, Poly)’ (columns 1:3)
```

You can even write interactions between categorical and numerical variables. Here we fit two different slope coefficients for `x1`; one for the `a1` group, and one for the `a2` group:

```
In [1]: dmatrix("a:x1", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept  a[a1]:x1  a[a2]:x1
          1   1.76405   0.00000
          1   0.40016   0.00000
          1   0.00000   0.97874
          1   0.00000   2.24089
          1   1.86756   0.00000
          1  -0.97728  -0.00000
          1   0.00000   0.95009
          1  -0.00000  -0.15136
  Terms:
    ’Intercept’ (column 0)
    ’a:x1’ (columns 1:3)
```

The same redundancy avoidance code works here, so if you’d rather have treatment-coded slopes (one slope for the `a1` group, and a second for the difference between the `a1` and `a2` group slopes), then you can request it like this:

```
# compare to the difference between "0 + a" and "1 + a"
In [1]: dmatrix("x1 + a:x1", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept        x1  a[T.a2]:x1
          1   1.76405     0.00000
          1   0.40016     0.00000
          1   0.97874     0.97874
          1   2.24089     2.24089
          1   1.86756     0.00000
          1  -0.97728    -0.00000
          1   0.95009     0.95009
          1  -0.15136    -0.15136
  Terms:
    ’Intercept’ (column 0)
    ’x1’ (column 1)
    ’a:x1’ (column 2)
```

And more complex expressions work too:

```
In [1]: dmatrix("C(a, Poly):center(x1)", data)
 Out[1]:
DesignMatrix with shape (8, 3)
  Intercept  C(a, Poly).Constant:center(x1)  C(a, Poly).Linear:center(x1)
          1                         0.87995                      -0.62222
          1                        -0.48395                       0.34220
          1                         0.09463                       0.06691
          1                         1.35679                       0.95939
          1                         0.98345                      -0.69541
          1                        -1.86138                       1.31620
          1                         0.06598                       0.04666
          1                        -1.03546                      -0.73218
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Poly):center(x1)’ (columns 1:3)
```

CHAPTER

THREE

HOW FORMULAS WORK

Now we'll describe the fully nitty-gritty of how formulas are parsed and interpreted. Here's the picture you'll want to keep in mind:

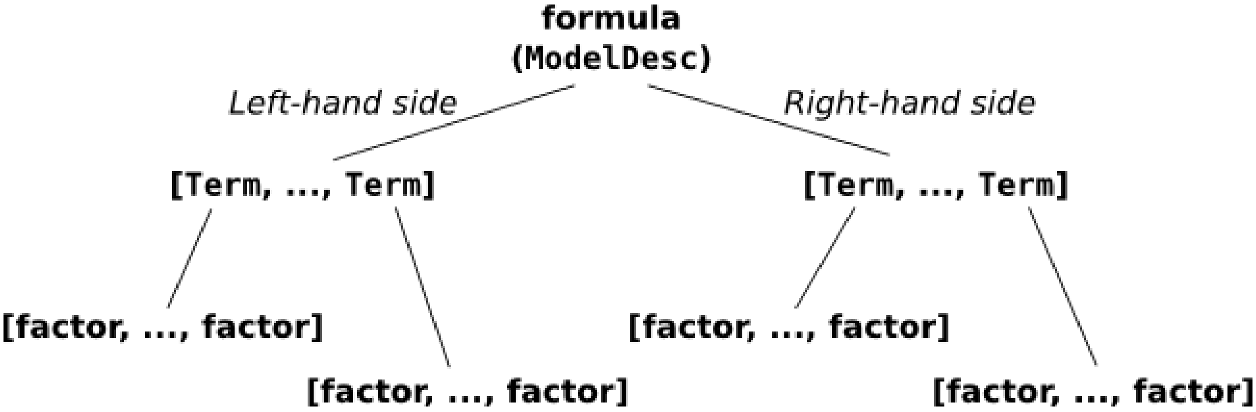


Figure 3.1: The pieces that make up a formula.

Say we have a formula like:

```
y ~ a + a:b + np.log(x)
```

This overall thing is a **formula**, and it's divided into a left-hand side, `y`, and a right-hand side, `a + a:b + np.log(x)`. (Sometimes you want a formula that has no left-hand side, and you can write that as `~ x1 + x2` or even `x1 + x2`.) Each side contains a list of **terms** separated by `+`; on the left there is one term, `y`, and on the right, there are four terms: `a` and `a:b` and `np.log(x)`, plus an invisible intercept term. And finally, each term is the interaction of zero or more **factors**. A factor is the minimal, indivisible unit that each formula is built up out of; the factors here are `y`, `a`, `b`, and `np.log(x)`. Most of these terms have only one factor – for example, the term `y` is a kind of trivial interaction between the factor `y` and, well... and nothing. There's only one factor in that "interaction". The term `a:b` is an interaction between two factors, `a` and `b`. And the intercept term is an interaction between *zero* factors. (This may seem odd, but it turns out that defining the zero-order interaction to produce a column of all ones is very convenient, just like it turns out to be convenient to define the product of an empty list to be `np.prod([]) == 1`.)

Note: In the context of Patsy, the word **factor** does *not* refer specifically to categorical data. What we call a "factor" can represent either categorical or numerical data. Think of factors like in multiplying factors together, not like in factorial design. When we want to refer to categorical data, this manual and the Patsy API use the word "categorical".

To make this more concrete, here's how you could manually construct the same objects that Patsy will construct if given the above formula:

```
from patsy import EvalEnvironment, ModelDesc
env = EvalEnvironment.capture()
ModelDesc([Term([EvalFactor("y", env)])],
          [Term([]),
           Term([EvalFactor("a", env)]),
           Term([EvalFactor("a", env), EvalFactor("b", env)]),
           Term([EvalFactor("np.log(x)", env)])])
```

Compare to what you get from parsing the above formula:

```
ModelDesc.from_formula("y ~ a + a:b + np.log(x)", env)
```

`ModelDesc` represents an overall formula; it just takes two lists of `Term` objects, representing the left-hand side and the right-hand side. And each `Term` object just takes a list of factor objects. In this case our factors are of type `EvalFactor`, which evaluates arbitrary Python code, but in general any object that implements the factor protocol will do – for details see *Model specification for experts and computers*.

Of course as a user you never have to actually touch `ModelDesc`, `Term`, or `EvalFactor` objects by hand – but it's useful to know that this lower layer exists in case you ever want to generate a formula programmatically, and to have an image in your mind of what a formula really is.

3.1 The formula language

Now let's talk about exactly how those magic formula strings are processed.

Since a term is nothing but a set of factors, and a model is nothing but two sets of terms, you can write any Patsy model just using `:` to create interactions, `+` to join terms together into a set, and `~` to separate the left-hand side from the right-hand side. But for convenience, Patsy also understands a number of other short-hand operators, and evaluates them all using a full-fledged parser complete with robust error reporting, etc.

3.1.1 Operators

The built-in binary operators, ordered by precedence, are:

`~`	lowest precedence (binds most loosely)
`+`, `-`	
`*`, `/`	
`:`	
`**`	highest precedence (binds most tightly)

Of course, you can override the order of operations using parentheses. All operations are left-associative (so `a - b - c` means the same as `(a - b) - c`, not `a - (b - c)`). Their meanings are as follows:

~ Separates the left-hand side and right-hand side of a formula. Optional. If not present, then the formula is considered to contain a right-hand side only.

+ Takes the set of terms given on the left and the set of terms given on the right, and returns a set of terms that combines both (i.e., it computes a set union). Note that this means that `a + a` is just `a`.

- Takes the set of terms given on the left and removes any terms which are given on the right (i.e., it computes a set difference).

`*` `a * b` is short-hand for `a + b + a:b`, and is useful for the common case of wanting to include all interactions between a set of variables while partitioning their variance between lower- and higher-order interactions. Standard ANOVA models are of the form `a * b * c * ...`.

`/` This one is a bit quirky. `a / b` is shorthand for `a + a:b`, and is intended to be useful in cases where you want to fit a standard sort of ANOVA model, but `b` is nested within `a`, so `a*b` doesn't make sense. So far so good. Also, if you have multiple terms on the right, then the obvious thing happens: `a / (b + c)` is equivalent to `a + a:b + a:c` (`/` is rightward distributive over `+`). *But,* if you have multiple terms on the left, then there is a surprising special case: `(a + b)/c` is equivalent to `a + b + a:b:c` (and note that this is different from what you'd get out of `a/c + b/c` – `/` is *not* leftward distributive over `+`). Again, this is motivated by the idea of using this for nested variables. It doesn't make sense for `c` to be nested within both `a` and `b` separately, unless `b` is itself nested in `a` – but if that were true, then you'd write `a/b/c` instead. So if we see `(a + b)/c`, we decide that `a` and `b` must be independent factors, but that `c` is nested within each *combination* of levels of `a` and `b`, which is what `a:b:c` gives us. If this is confusing, then my apologies... S has been working this way for >20 years, so it's a bit late to change it now.

`:` This takes two sets of terms, and computes the interaction between each term on the left and each term on the right. So, for example, `(a + b):(c + d)` is the same as `a:c + a:d + b:c + b:d`. Calculating the interaction between two terms is also a kind of set union operation, but `:` takes the union of factors *within* two terms, while `+` takes the union of two sets of terms. Note that this means that `a:a` is just `a`, and `(a:b):(a:c)` is the same as `a:b:c`.

`**` This takes a set of terms on the left, and an integer *n* on the right, and computes the `*` of that set of terms with itself *n* times. This is useful if you want to compute all interactions up to order *n*, but no further. Example:

```
(a + b + c + d) ** 3
```

is expanded to:

```
(a + b + c + d) * (a + b + c + d) * (a + b + c + d)
```

Note that an equivalent way to write this particular expression would be:

```
a*b*c*d - a:b:c:d
```

(*Exercise:* why?)

The parser also understands unary `+` and `-`, though they aren't very useful. `+` is a no-op, and `-` can only be used in the forms `-1` (which means the same as `0`) and `-0` (which means the same as `1`). See *below* for more on `0` and `1`.

3.1.2 Factors and terms

So that explains how the operators work – the verbs in the formula language – but what about the nouns, the terms like `y` and `np.log(x)` that are actually picking out bits of your data?

Individual factors are allowed to be arbitrary Python code. Scanning arbitrary Python code can be quite complicated, but Patsy uses the official Python tokenizer that's built into the standard library, so it's able to do it robustly. There is still a bit of a problem, though, since Patsy operators like `+` are also valid Python operators. When we see a `+`, how do we know which interpretation to use?

The answer is that a Python factor begins whenever we see a token which

- is not a Patsy operator listed in that table up above, and
- is not a parentheses

And then the factor ends whenever we see a token which

- is a Patsy operator listed in that table up above, and
- it not enclosed in any kind of parentheses (where "any kind" includes regular, square, and curly bracket varieties)

This will be clearer with an example:

```
f(x1 + x2) + x3
```

First, we see `f`, which is not an operator or a parentheses, so we know this string begins with a Python-defined factor. Then we keep reading from there. The next Patsy operator we see is the `+` in `x1 + x2`... but since at this point we have seen the opening `(` but not the closing `)`, we know that we're inside parentheses and ignore it. Eventually we come to the second `+`, and by this time we have seen the closing parentheses, so we know that this is the end of the first factor and we interpret the `+` as a Patsy operator.

One side-effect of this is that if you do want to perform some arithmetic inside your formula object, you can hide it from the Patsy parser by putting it inside a function call. To make this more convenient, Patsy provides a builtin function `I()` that simply returns its input. (Hence the name: it's the Identity function.) This means you can use `I(x1 + x2)` inside a formula to represent the sum of `x1` and `x2`.

Note: The above plays a bit fast-and-loose with the distinction between factors and terms. If you want to get more technical, then given something like `a:b`, what's happening is first that we create a factor `a` and then we package it up into a single-factor term. And then we create a factor `b`, and we package it up into a single-factor term. And then we evaluate the `:`, and compute the interaction between these two terms. When we encounter embedded Python code, it's always converted straight to a single-factor term before doing anything else.

3.1.3 Intercept handling

There are two special things about how intercept terms are handled inside the formula parser.

First, since an intercept term is an interaction of zero factors, we have no way to write it down using the parts of the language described so far. Therefore, as a special case, the string `1` is taken to represent the intercept term.

Second, since intercept terms are almost always wanted and remembering to include them by hand all the time is quite tedious, they are always included by default in the right-hand side of any formula. The way this is implemented is exactly as if there is an invisible `1 +` inserted at the beginning of every right-hand side.

Of course, if you don't want an intercept, you can remove it again just like any other unwanted term, using the `-` operator. The only thing that's special about the `1 +` is that it's invisible; otherwise it acts just like any other term. This formula has an intercept:

```
y ~ x
```

because it is processed like `y ~ 1 + x`.

This formula does not have an intercept:

```
y ~ x - 1
```

because it is processed like `y ~ 1 + x - 1`.

Of course if you want to be really explicit you can mention the intercept explicitly:

```
y ~ 1 + x
```

Once the invisible `1 +` is added, this formula is processed like `y ~ 1 + 1 + x`, and as you'll recall from the definition of `+` above, adding the same term twice produces the same result as adding it just once.

For compatibility with S and R, we also allow the magic terms `0` and `-1` which represent the "anti-intercept". Adding one of these terms has exactly the same effect as subtracting the intercept term, and subtracting one of these terms has exactly the same effect as adding the intercept term. That means that all of these formulas are equivalent:

```
y ~ x - 1
y ~ x + -1
y ~ -1 + x
y ~ 0 + x
y ~ x - (-0)
```

3.1.4 Explore!

The formula language is actually fairly simple once you get the hang of it, but if you're ever in doubt as to what some construction means, you can always ask Patsy how it expands.

Here's some code to try out at the Python prompt to get started:

```
from patsy import EvalEnvironment, ModelDesc
# This captures the current Python environment. If a factor refers
# to a variable that doesn't exist in the data (like np.log) then it
# will be looked for here.
env = EvalEnvironment.capture()
ModelDesc.from_formula("y ~ x", env)
ModelDesc.from_formula("y ~ x + x + x", env)
ModelDesc.from_formula("y ~ -1 + x", env)
ModelDesc.from_formula("~ -1", env)
ModelDesc.from_formula("y ~ a:b", env)
ModelDesc.from_formula("y ~ a*b", env)
ModelDesc.from_formula("y ~ (a + b + c + d) ** 2", env)
ModelDesc.from_formula("y ~ (a + b)/(c + d)", env)
ModelDesc.from_formula("np.log(x1 + x2) "
                       "+ (x + {6: x3, 8 + 1: x4}[3 * i])", env)
```

Sometimes it might be easier to read if you put the processed formula back into formula notation using `ModelDesc.describe()`:

```
desc = ModelDesc.from_formula("y ~ (a + b + c + d) ** 2", env)
desc.describe()
```

3.2 From terms to matrices

So at this point, you hopefully understand how a string is parsed into the `ModelDesc` structure shown in the figure at the top of this page. And if you like you can also produce such structures directly without going through the formula parser (see *Model specification for experts and computers*). But these terms and factors objects are still a fairly high-level, symbolic representation of a model. Now we'll talk about how they get converted into actual matrices with numbers in.

There are two core operations here. The first takes a list of `Term` objects (a **termlist**) and some data, and produces a `DesignMatrixBuilder`. The second takes a `DesignMatrixBuilder` and some data, and produces a design matrix. In practice, these operations are implemented by `design_matrix_builders()` and `build_design_matrices()`, respectively, and each of these functions is "vectorized" to process an arbitrary number of matrices together in a single operation. But we'll ignore that for now, and just focus on what happens to a single termlist.

First, each individual factor is given a chance to set up any *Stateful transforms* it may have, and then is evaluated on the data, to determine:

- Whether it is categorical or numerical
- If it is categorical, what levels it has

- If it is numerical, how many columns it has.

Next, we sort terms based on the factors they contain. This is done by dividing terms into groups based on what combination of numerical factors each one contains. The group of terms that have no numerical factors comes first, then the rest of the groups in the order they are first mentioned within the term list. Then within each group, lower-order interactions are ordered to come before higher-order interactions. (Interactions of the same order are left alone.)

Example:

```
In [1]: data = demo_data("a", "b", "x1", "x2")

In [2]: mat = dmatrix("x1:x2 + a:b + b + x1:a:b + a + x2:a:x1", data)

In [3]: mat.design_info.term_names
Out[3]: ['Intercept', 'b', 'a', 'a:b', 'x1:x2', 'x2:a:x1', 'x1:a:b']
```

The non-numerical terms are *Intercept*, *b*, *a*, *a:b* and they come first, sorted from lower-order to higher-order. *b* comes before *a* because it did in the original formula. Next come the terms that involved *x1* and *x2* together, and *x1:x2* comes before *x2:a:x1* because it is a lower-order term. Finally comes the sole term involving *x1* without *x2*.

Note: These ordering rules may seem a bit arbitrary, but will make more sense after our discussion of redundancy below. Basically the motivation is that terms like *b* and *a* represent overlapping vector spaces, which means that the presence of one will affect how the other is coded. So, we group to them together, to make these relationships easier to see in the final analysis. And, a term like *b* represents a sub-space of a term like *a:b*, so if you're including both terms in your model you presumably want the variance represented by *b* to be partitioned out separately from the overall *a:b* term, and for that to happen, *b* should come first in the final model.

After sorting the terms, we determine appropriate coding schemes for categorical factors, as described in the next section. And that's it – we now know exactly how to produce this design matrix, and `design_matrix_builders()` packages this knowledge up into a `DesignMatrixBuilder` and returns it. To get the design matrix itself, we then use `build_design_matrices()`.

3.2.1 Redundancy and categorical factors

Here's the basic idea about how Patsy codes categorical factors: each term that's included means that we want our outcome variable to be able to vary in a certain way – for example, the *a:b* in `y ~ a:b` means that we want our model to be flexible enough to assign *y* a different value for every possible combination of *a* and *b* values. So what Patsy does is build up a design matrix incrementally by working from left to right in the sorted term list, and for each term it adds just the right columns needed to make sure that the model will be flexible enough to include the kind of variation this term represents, while keeping the overall design matrix full rank. The result is that the columns associated with each term always represent the *additional* flexibility that the models gains by adding that term, on top of the terms to its left. Numerical factors are assumed not to be redundant with each other, and are always included "as is"; categorical factors and interactions might be redundant, so Patsy chooses either full-rank or reduced-rank contrast coding for each one to keep the overall design matrix at full rank.

Note: We're only worried here about "structural redundancies", those which occur inevitably no matter what the particular values occur in your data set. If you enter two different factors *x1* and *x2*, but set them to be numerically equal, then Patsy will indeed produce a design matrix that isn't full rank. Avoiding that is your problem.

Okay, now for the more the more detailed explanation. Each term represents a certain space of linear combinations of column vectors:

- A numerical factor represents the vector space spanned by its columns.
- A categorical factor represents the vector space spanned by the columns you get if you apply "dummy coding".

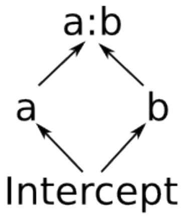
- An interaction between two factors represents the vector space spanned by the element-wise products between vectors in the first factor's space with vectors in the second factor's space. For example, if c_{1a} and c_{1b} are two columns that form a basis for the vector space represented by factor f_1, and likewise c_{2a} and c_{2b} are a basis for the vector space represented by f_2, then $c_{1a} * c_{2a}$, $c_{1b} * c_{2a}$, $c_{1a} * c_{2b}$, $c_{1b} * c_{2b}$ is a basis for the vector space represented by $f_1 : f_2$. Here the $*$ operator represents elementwise multiplication, like numpy `*`. (*Exercise:* show that the choice of basis does not matter.)
- The empty interaction represents the space spanned by the identity element for elementwise multiplication, i.e., the all-ones "intercept" term.

So suppose that *a* is a categorical factor with two levels *a1* and *a2*, and *b* is a categorical factor with two levels *b1* and *b1*. Then:

- *a* represents the space spanned by two vectors: one that has a 1 everywhere that `a == "a1"`, and a zero everywhere else, and another that's similar but for `a == "a2"`. (dummy coding)
- *b* works similarly
- and *a:b* represents the space spanned by *four* vectors: one that has a 1 everywhere that has `a == "a1"` and `b == "b1"`, another that has a 1 everywhere that has `a1 == "a2"` and `b == "b1"`, etc. So if you are familiar with ANOVA terminology, then these are *not* the kinds of interactions you are expecting! They represent a more fundamental idea, that when we write:

  y ~ a:b

  we mean that the value of *y* can vary depending on every possible *combination* of *a* and *b*.



Notice that this means that the space spanned by the intercept term is always a vector subspace of the spaces spanned by *a* and *b*, and these subspaces in turn are always subspaces of the space spanned by *a:b*. (Another way to say this is that *a* and *b* are "marginal to" *a:b*.) The diagram on the right shows these relationships graphically. This reflects the intuition that allowing *y* to depend on every combination of *a* and *b* gives you a more flexible model than allowing it to vary based on just *a* or just *b*.

So what this means is that once you have *a:b* in your model, adding *a* or *b* or the intercept term won't actually give you any additional flexibility; the most they can do is to create redundancies that your linear algebra package will have to somehow detect and remove later. These two models are identical in terms of how flexible they are:

```
y ~ 0 + a:b
y ~ 1 + a + b + a:b
```

And, indeed, we can check that the matrices that Patsy generates for these two formulas have identical column spans:

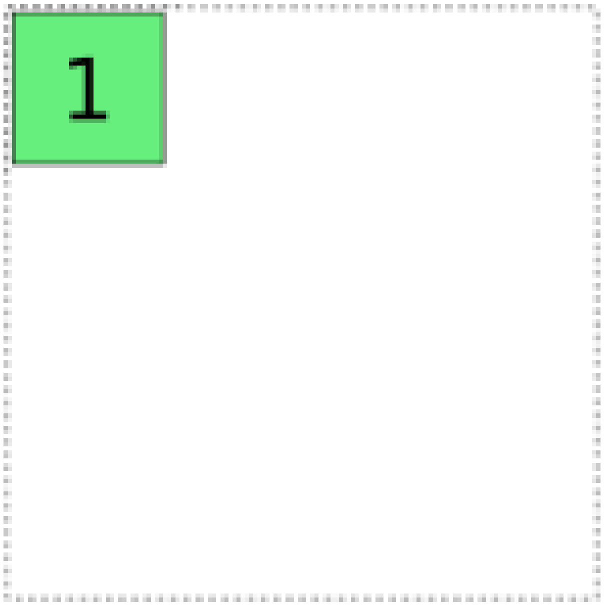
```
In [1]: data = demo_data("a", "b", "y")

In [2]: mat1 = dmatrices("y ~ 0 + a:b", data)[1]

In [3]: mat2 = dmatrices("y ~ 1 + a + b + a:b", data)[1]

In [4]: np.linalg.matrix_rank(mat1)
Out[4]: 4

In [5]: np.linalg.matrix_rank(mat2)
```

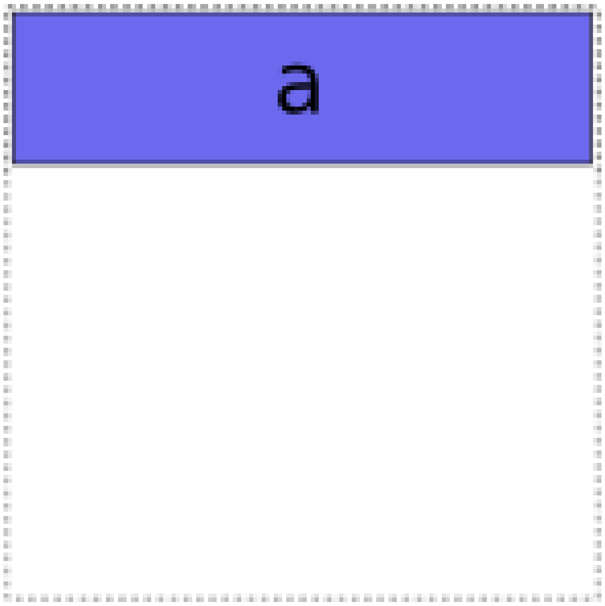
```
Out[5]: 4

In [6]: np.linalg.matrix_rank(np.column_stack((mat1, mat2)))
Out[6]: 4
```

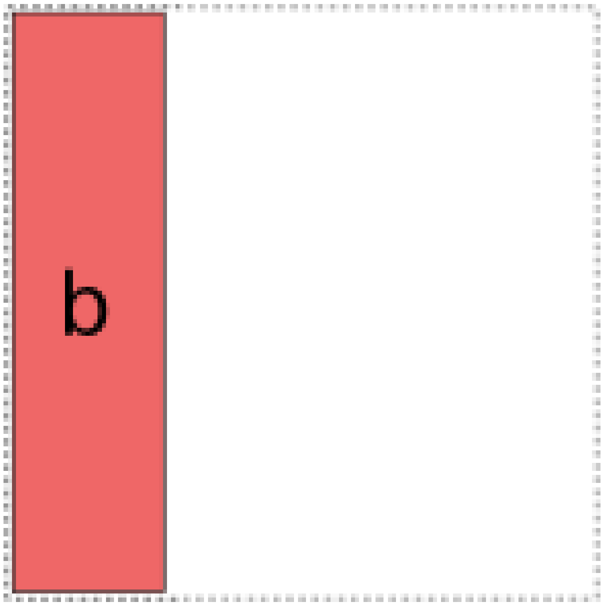
But, of course, their actual contents is different:

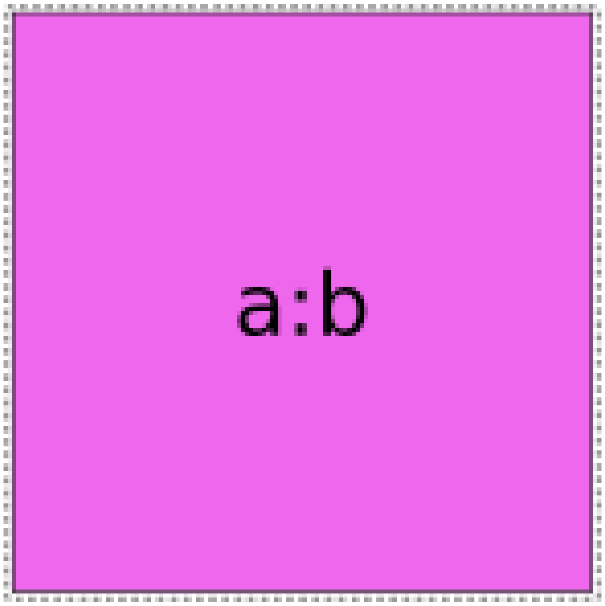
```
In [1]: mat1
Out[1]:
DesignMatrix with shape (8, 4)
  a[a1]:b[b1]  a[a2]:b[b1]  a[a1]:b[b2]  a[a2]:b[b2]
            1            0            0            0
            0            0            1            0
            0            1            0            0
            0            0            0            1
            1            0            0            0
            0            0            1            0
            0            1            0            0
            0            0            0            1
  Terms:
    ’a:b’ (columns 0:4)

In [2]: mat2
Out[2]:
DesignMatrix with shape (8, 4)
  Intercept  a[T.a2]  b[T.b2]  a[T.a2]:b[T.b2]
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
  Terms:
    ’Intercept’ (column 0)
    ’a’ (column 1)
    ’b’ (column 2)
    ’a:b’ (column 3)
```

This happens because Patsy is finding ways to avoid creating redundancy while coding each term. To understand how this works, it’s useful to draw some pictures. Patsy has two general strategies for coding a categorical factor with n levels. The first is to use a full-rank encoding with n columns. Here are some pictures of this style of coding:









Obviously if we lay these images on top of each other, they'll overlap, which corresponds to their overlap when considered as vector spaces. If we try just putting them all into the same model, we get mud:

Patsy avoids this by using its second strategy: coding an n level factor in $n - 1$ columns which, critically, do not span the intercept. We'll call this style of coding *reduced-rank*, and use notation like *a-* to refer to factors coded this way.

Note: Each of the categorical coding schemes included in `patsy` come in both full-rank and reduced-rank flavours. If you ask for, say, `Poly` coding, then this is the mechanism used to decide whether you get full- or reduced-rank `Poly` coding.

For coding *a* there are two options:

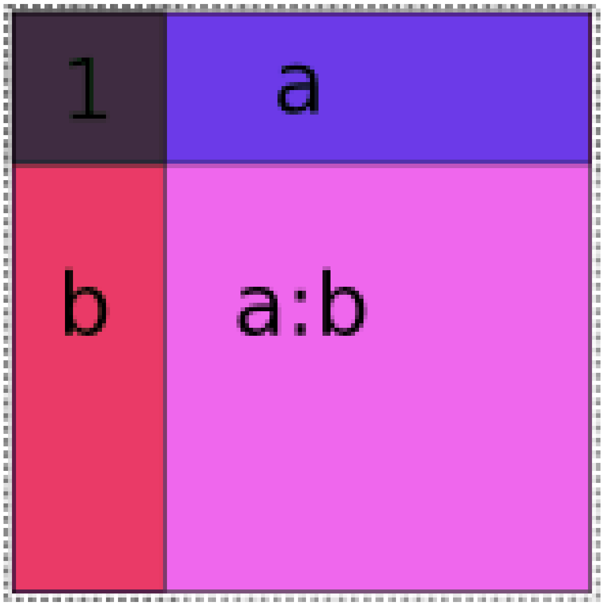
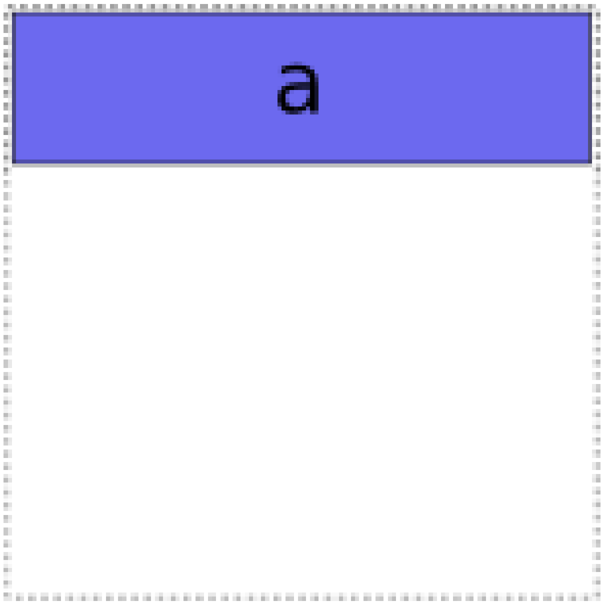
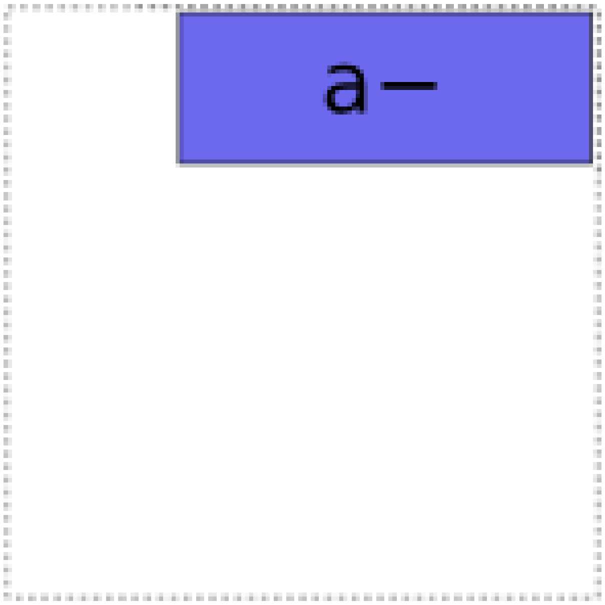
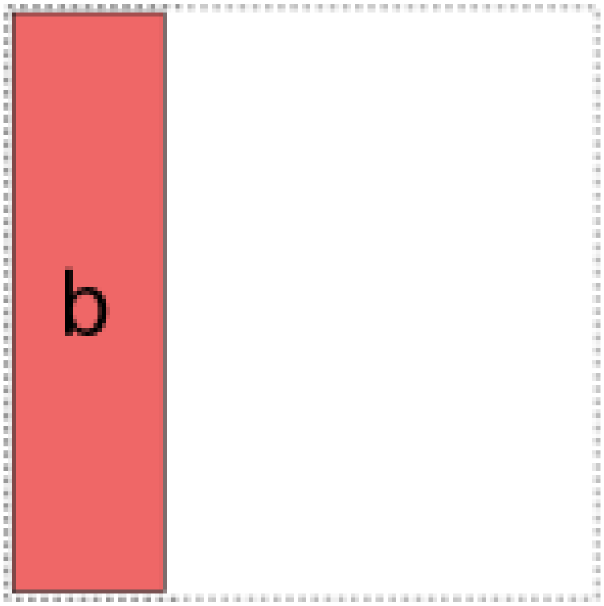


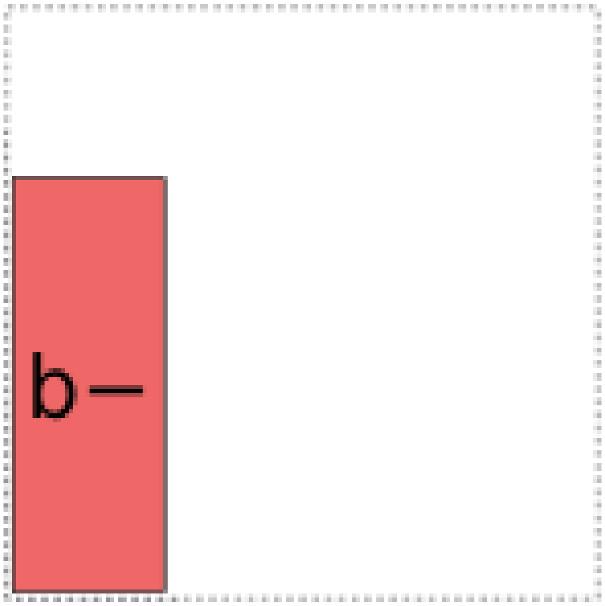
Figure 3.2: Naive *1 + a + b + a:b*



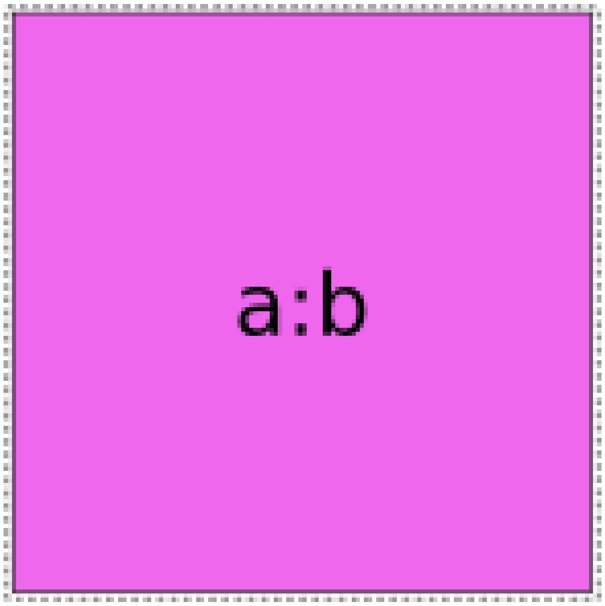


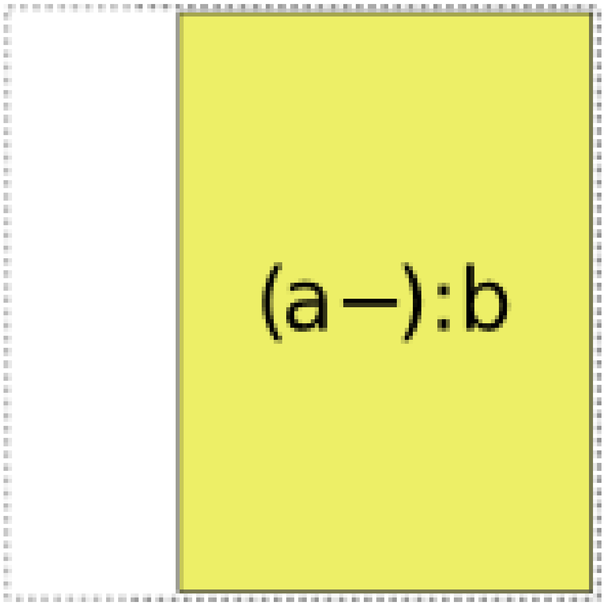
And likewise for *b*:

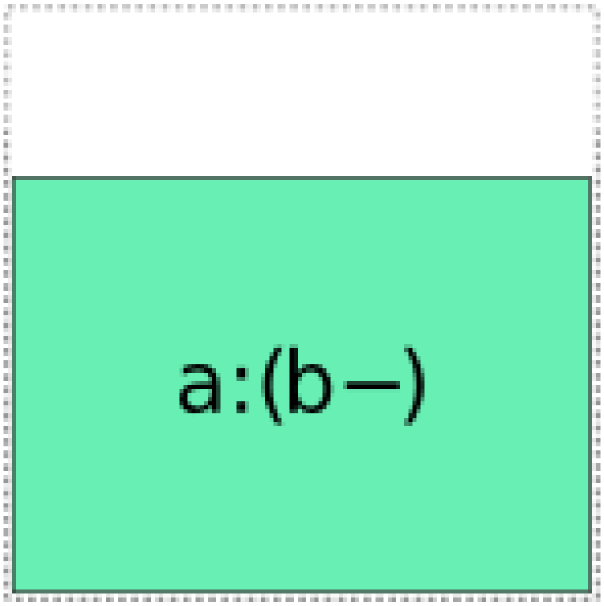


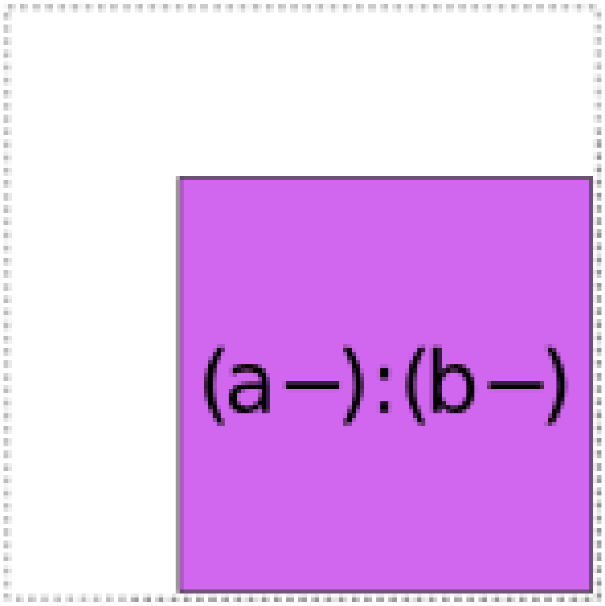


When it comes to *a:b*, things get more interesting: it can choose whether to use a full- or reduced-rank encoding separately for each factor, leading to four choices overall:



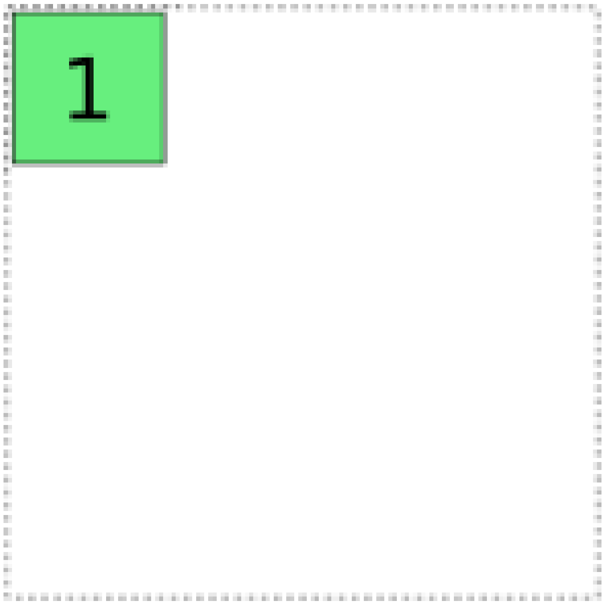






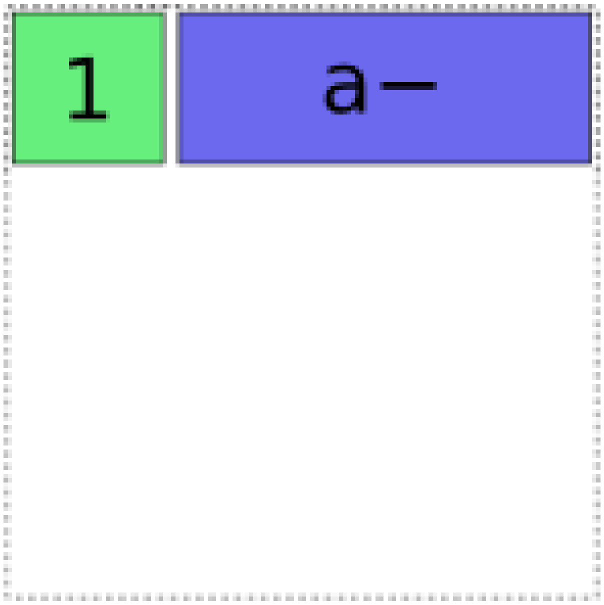
So when interpreting a formula like `1 + a + b + a:b`, Patsy's job is to pick and choose from the above pieces and then assemble them together like a jigsaw puzzle.

Let's walk through the formula `1 + a + b + a:b` to see how this works. First it encodes the intercept:



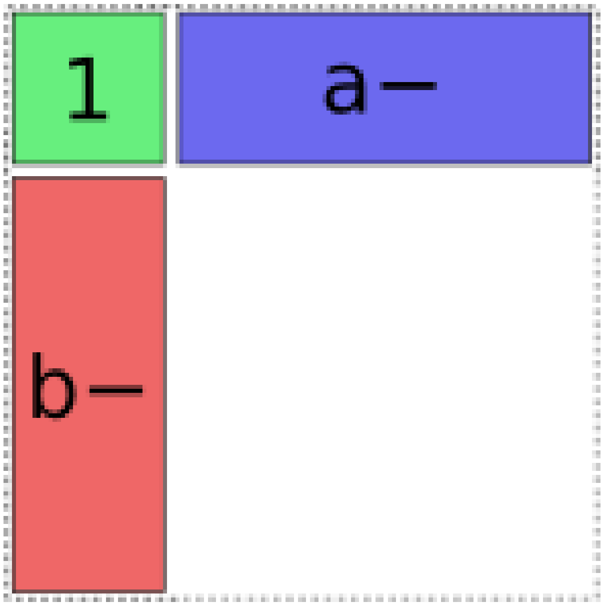
```
In [1]: dmatrices("y ~ 1", data)[1]
Out[1]:
DesignMatrix with shape (8, 1)
  Intercept
          1
          1
          1
          1
          1
          1
          1
          1
  Terms:
    'Intercept' (column 0)
```

Then it adds the *a* term. It has two choices, either the full-rank coding or the reduced rank *a-* coding. Using the full-rank coding would overlap with the already-existing intercept term, though, so it chooses the reduced rank coding:



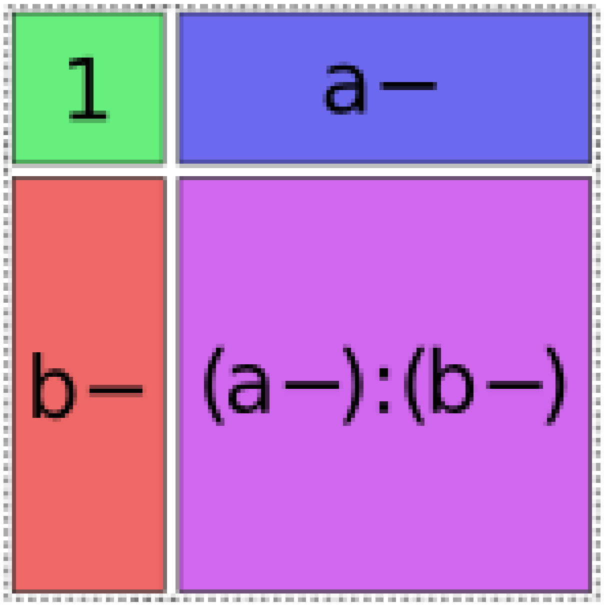
```
In [1]: dmatrices("y ~ 1 + a", data)[1]
Out[1]:
DesignMatrix with shape (8, 2)
  Intercept  a[T.a2]
          1        0
          1        0
          1        1
          1        1
          1        0
          1        0
          1        1
          1        1
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
```

The *b* term is treated similarly:



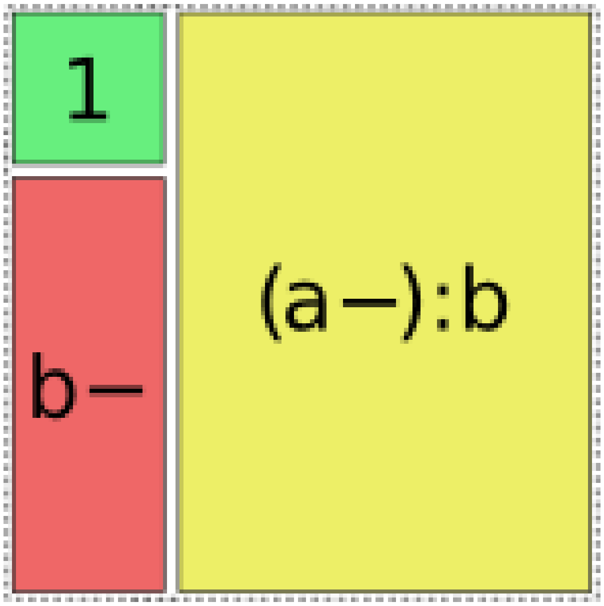
```
In [1]: dmatrices("y ~ 1 + a + b", data)[1]
Out[1]:
DesignMatrix with shape (8, 3)
  Intercept  a[T.a2]  b[T.b2]
          1        0        0
          1        0        1
          1        1        0
          1        1        1
          1        0        0
          1        0        1
          1        1        0
          1        1        1
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
    'b' (column 2)
```

And finally, there are four options for the *a:b* term, but only one of them will fit without creating overlap:



```
In [1]: dmatrices("y ~ 1 + a + b + a:b", data)[1]
Out[1]:
DesignMatrix with shape (8, 4)
  Intercept  a[T.a2]  b[T.b2]  a[T.a2]:b[T.b2]
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
          1        0        0                0
          1        0        1                0
          1        1        0                0
          1        1        1                1
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
    'b' (column 2)
    'a:b' (column 3)
```

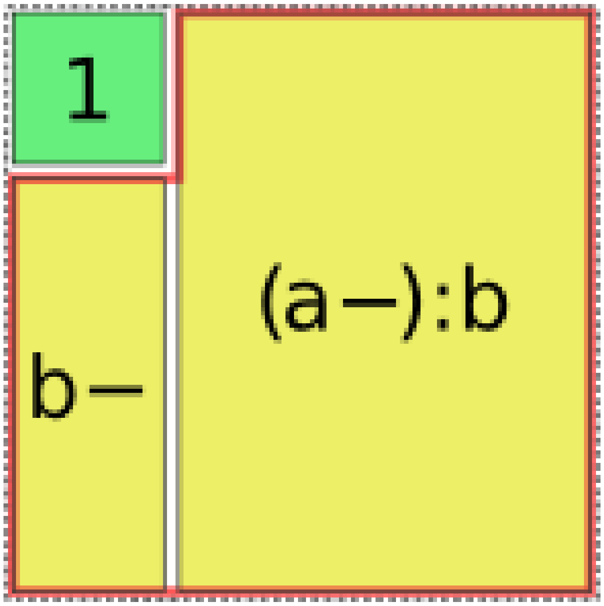
Patsy tries to use the fewest pieces possible to cover the space. For instance, in this formula, the *a:b* term is able to fill the remaining space by using a single piece:



```
In [1]: dmatrices("y ~ 1 + b + a:b", data)[1]
Out[1]:
DesignMatrix with shape (8, 4)
  Intercept  b[T.b2]  a[T.a2]:b[b1]  a[T.a2]:b[b2]
          1        0              0              0
          1        1              0              0
          1        0              1              0
          1        1              0              1
          1        0              0              0
          1        1              0              0
          1        0              1              0
          1        1              0              1
  Terms:
    'Intercept' (column 0)
    'b' (column 1)
    'a:b' (columns 2:4)
```

However, this is not always possible. In such cases, Patsy will assemble multiple pieces to code a single term [1], e.g.:

[1] This is one of the places where Patsy improves on R, which produces incorrect output in this case (see *Differences between R and Patsy formulas*).



```
In [1]: dmatrices("y ~ 1 + a:b", data)[1]
Out[1]:
DesignMatrix with shape (8, 4)
  Intercept  b[T.b2]  a[T.a2]:b[b1]  a[T.a2]:b[b2]
          1        0              0              0
          1        1              0              0
          1        0              1              0
          1        1              0              1
          1        0              0              0
          1        1              0              0
          1        0              1              0
          1        1              0              1
  Terms:
    'Intercept' (column 0)
    'a:b' (columns 1:4)
```

Notice that the matrix entries and column names here are identical to those produced by the previous example, but the association between terms and columns shown at the bottom is different.

In all of these cases, the final model spans the same space; *a:b* is included in the formula, and therefore the final matrix must fill in the full *a:b* square. By including different combinations of lower-order interactions, we can control how this overall variance is partitioned into distinct terms.

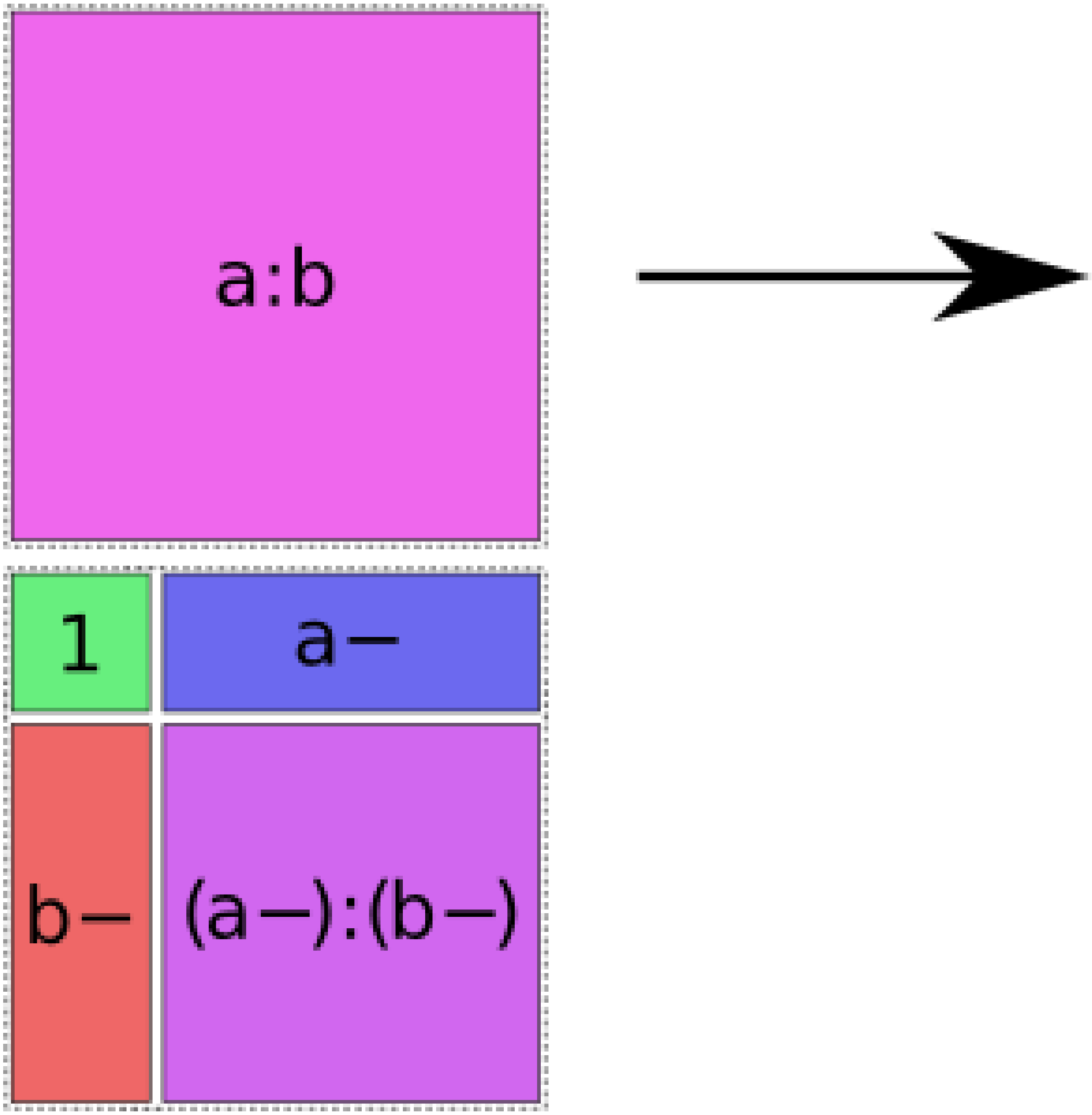
> *Exercise:* create the similar diagram for a formula that includes a three-way interaction, like `1 + a + a:b + a:b:c` or `1 + a:b:c`. Hint: it's a cube. Then, send us your diagram for inclusion in this documentation [2].

Finally, we've so far only discussed purely categorical interactions. Bringing numerical interactions into the mix doesn't make things much more complicated. Each combination of numerical factors is considered to be distinct from all other combinations, so we divide all of our terms into groups based on which numerical factors they contain (just like we do when sorting terms, as described above), and then within each group we separately apply the algorithm described here to the categorical parts of each term.

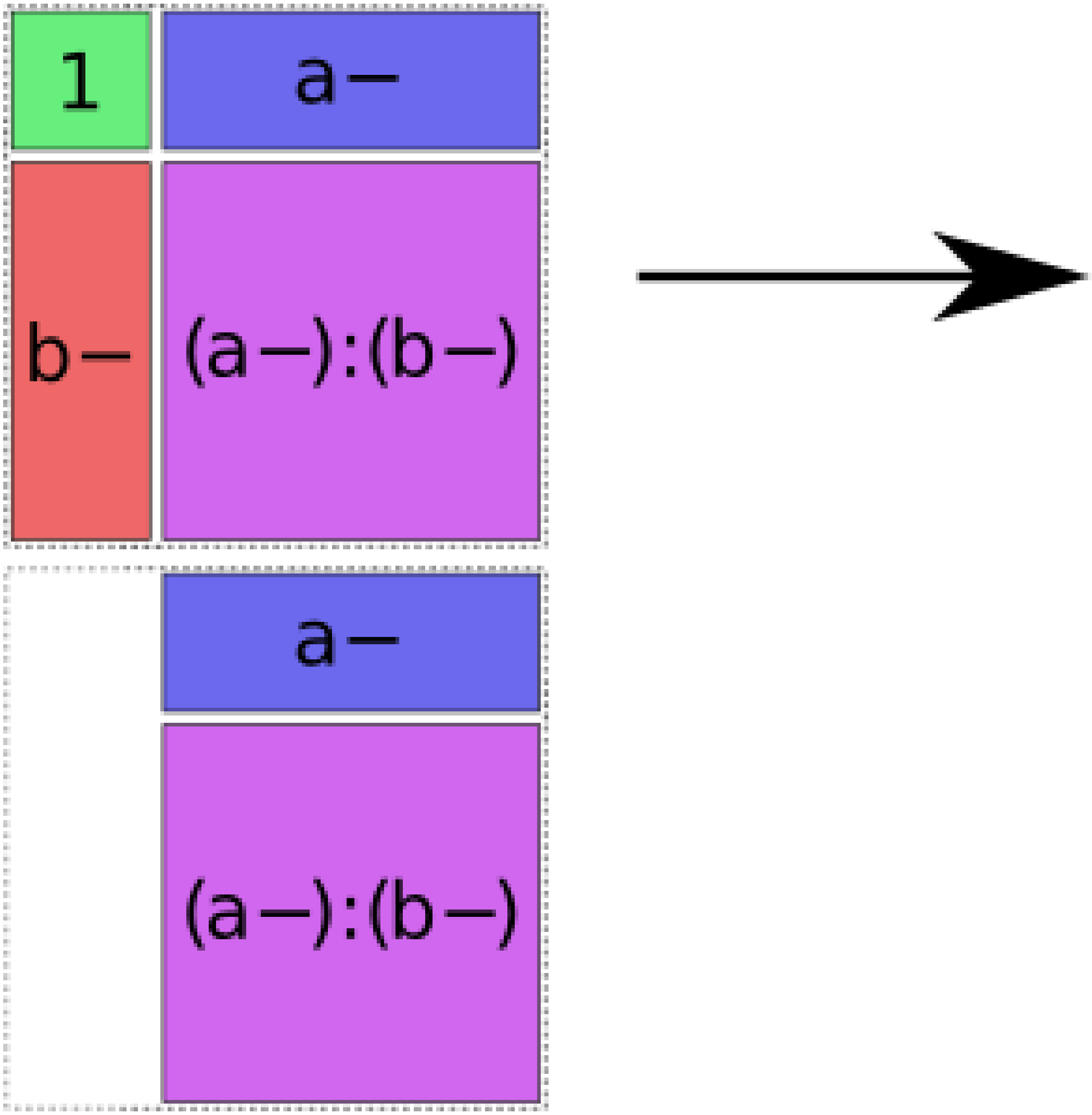
[2] Yes, I'm lazy. And shameless.

3.3 Technical details

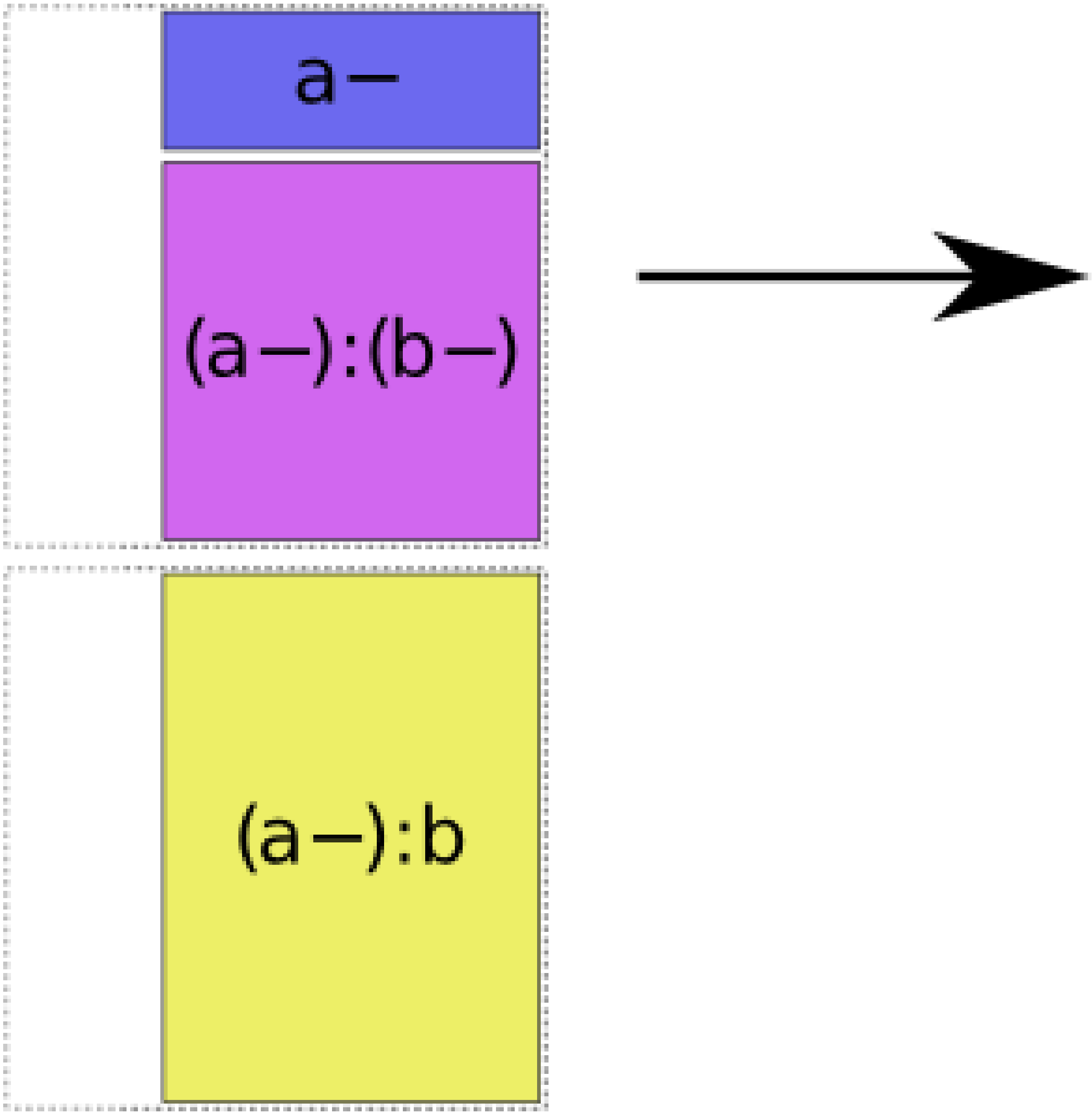
The actual algorithm Patsy uses to produce the above coding is very simple. Within the group of terms associated with each combination of numerical factors, it works from left to right. For each term it encounters, it breaks the categorical part of the interaction down into minimal pieces, e.g. *a:b* is replaced by *1 + (a-) + (b-) + (a-):(b-)*:



(Formally speaking, these "minimal pieces" consist of the set of all subsets of the original interaction.) Then, any of the minimal pieces which were used by a previous term within this group are deleted, since they are redundant:



and then we greedily recombine the pieces that are left by repeatedly merging adjacent pieces according to the rule *ANYTHING + ANYTHING : FACTOR- = ANYTHING : FACTOR*:



> *Exercise:* Prove formally that the space spanned by *ANYTHING* + *ANYTHING : FACTOR-* is identical to the space spanned by *ANYTHING : FACTOR*.
>
> *Exercise:* Either show that the greedy algorithm here is produces optimal encodings in some sense (e.g., smallest number of pieces used), or else find a better algorithm. (Extra credit: implement your algorithm and submit a pull request [3].)

Is this algorithm correct? A full formal proof would be too tedious for this reference manual, but here's a sketch of the analysis.

Recall that our goal is to maintain two invariants: the design matrix column space should include the space associated with each term, and should avoid "structural redundancy", i.e. it should be full rank on at least some data sets. It's easy to see the above algorithm will never "lose" columns, since the only time it eliminates a subspace is when it has previously processed that exact subspace within the same design. But will it always detect all the redundancies that are present?

[3] Yes, still shameless.

That is guaranteed by the following theorem:

Theorem: Let two sets of factors, $F = f_1, \ldots, f_n$ and $G = g_1, \ldots, g_m$ be given, and let $F = F_{\text{num}} \cup F_{\text{categ}}$ be the numerical and categorical factors, respectively (and similarly for $G = G_{\text{num}} \cup G_{\text{categ}}$. Then the space represented by the interaction $f_1 : \cdots : f_n$ has a non-trivial intersection with the space represented by the interaction $g_1 : \cdots : g_m$ whenever:

- $F_{\text{num}} = G_{\text{num}}$, and
- $F_{\text{categ}} \cap G_{\text{categ}} \neq \emptyset$

And, furthermore, whenever this condition does not hold, then there exists some assignment of values to the factors for which the associated vector spaces have only a trivial intersection.

> *Exercise:* Prove it.
>
> *Exercise:* Show that given a sufficient number of rows, the set of factor assignments on which $f_1 : \cdots : f_n$ represents a subspace of $g_1 : \cdots : g_n$ without the above conditions being satisfied is actually a zero set.

Corollary: Patsy's strategy of dividing into groups by numerical factors, and then comparing all subsets of the remaining categorical factors, allows it to precisely identify and avoid structural redundancies.

3.4 Footnotes

CHAPTER

FOUR

CODING CATEGORICAL DATA

Patsy allows great flexibility in how categorical data is coded, via the function `C()`. `C()` marks some data as being categorical (including data which would not automatically be treated as categorical, such as a column of integers), while also optionally setting the preferred coding scheme and level ordering.

Let's get some categorical data to work with:

```
In [1]: from patsy import dmatrix, demo_data, ContrastMatrix, Poly

In [2]: data = demo_data("a", nlevels=3)

In [3]: data
Out[3]: {'a': ['a1', 'a2', 'a3', 'a1', 'a2', 'a3']}
```

As you know, simply giving Patsy a categorical variable causes it to be coded using the default `Treatment` coding scheme. (Strings and booleans are treated as categorical by default.)

```
In [1]: dmatrix("a", data)
Out[1]:
DesignMatrix with shape (6, 3)
  Intercept  a[T.a2]  a[T.a3]
          1        0        0
          1        1        0
          1        0        1
          1        0        0
          1        1        0
          1        0        1
  Terms:
    'Intercept' (column 0)
    'a' (columns 1:3)
```

We can also alter the level ordering, which is useful for, e.g., `Diff` coding:

```
In [1]: l = ["a3", "a2", "a1"]

In [2]: dmatrix("C(a, levels=l)", data)
Out[2]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, levels=l)[T.a2]  C(a, levels=l)[T.a1]
          1                     0                     1
          1                     1                     0
          1                     0                     0
          1                     0                     1
          1                     1                     0
          1                     0                     0
  Terms:
    'Intercept' (column 0)
    'C(a, levels=l)' (columns 1:3)
```

But the default coding is just that – a default. The easiest alternative is to use one of the other built-in coding schemes, like orthogonal polynomial coding:

```
In [1]: dmatrix("C(a, Poly)", data)
Out[1]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, Poly).Linear  C(a, Poly).Quadratic
          1           -0.70711               0.40825
          1           -0.00000              -0.81650
          1            0.70711               0.40825
          1           -0.70711               0.40825
          1           -0.00000              -0.81650
          1            0.70711               0.40825
  Terms:
    'Intercept' (column 0)
    'C(a, Poly)' (columns 1:3)
```

There are a number of built-in coding schemes; for details you can check the *API reference*. But we aren't restricted to those. We can also provide a custom contrast matrix, which allows us to produce all kinds of strange designs:

```
In [1]: contrast = [[1, 2], [3, 4], [5, 6]]

In [2]: dmatrix("C(a, contrast)", data)
Out[2]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, contrast)[custom0]  C(a, contrast)[custom1]
          1                        1                        2
          1                        3                        4
          1                        5                        6
          1                        1                        2
          1                        3                        4
          1                        5                        6
  Terms:
    'Intercept' (column 0)
    'C(a, contrast)' (columns 1:3)

In [3]: dmatrix("C(a, [[1], [2], [-4]])", data)
Out[3]:
DesignMatrix with shape (6, 2)
  Intercept  C(a, [[1], [2], [-4]])[custom0]
          1                                1
          1                                2
          1                               -4
          1                                1
          1                                2
          1                               -4
  Terms:
    'Intercept' (column 0)
    'C(a, [[1], [2], [-4]])' (column 1)
```

Hmm, those `[custom0]`, `[custom1]` names that Patsy auto-generated for us are a bit ugly looking. We can attach names to our contrast matrix by creating a `ContrastMatrix` object, and make things prettier:

```
In [1]: contrast_mat = ContrastMatrix(contrast, ["[pretty0]", "[pretty1]"])
In [2]: dmatrix("C(a, contrast_mat)", data)
Out[2]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, contrast_mat)[pretty0]  C(a, contrast_mat)[pretty1]
          1                            1                            2
          1                            3                            4
          1                            5                            6
          1                            1                            2
          1                            3                            4
          1                            5                            6
  Terms:
    'Intercept' (column 0)
    'C(a, contrast_mat)' (columns 1:3)
```

And, finally, if we want to get really fancy, we can also define our own "smart" coding schemes like `Poly`. Just define a class that has two methods, `code_with_intercept()` and `code_without_intercept()`. They have identical signatures, taking a list of levels as their argument and returning a `ContrastMatrix`. Patsy will automatically choose the appropriate method to call to produce a full-rank design matrix without redundancy; see *Redundancy and categorical factors* for the full details on how Patsy makes this decision.

As an example, here's a simplified version of the built-in `Treatment` coding object:

```
import numpy as np

class MyTreat(object):
    def __init__(self, reference=0):
        self.reference = reference

    def code_with_intercept(self, levels):
        return ContrastMatrix(np.eye(len(levels)),
                              ["[My.%s]" % (level,) for level in levels])

    def code_without_intercept(self, levels):
        eye = np.eye(len(levels) - 1)
        contrasts = np.vstack((eye[:self.reference, :],
                               np.zeros((1, len(levels) - 1)),
                               eye[self.reference:, :]))
        suffixes = ["[MyT.%s]" % (level,) for level in
                    levels[:self.reference] + levels[self.reference + 1:]]
        return ContrastMatrix(contrasts, suffixes)
```

And it can now be used just like the built-in methods:

```
# Full rank:
In [1]: dmatrix("0 + C(a, MyTreat)", data)
Out[1]:
DesignMatrix with shape (6, 3)
  C(a, MyTreat)[My.a1]  C(a, MyTreat)[My.a2]  C(a, MyTreat)[My.a3]
                     1                     0                     0
                     0                     1                     0
                     0                     0                     1
                     1                     0                     0
                     0                     1                     0
                     0                     0                     1
  Terms:
    'C(a, MyTreat)' (columns 0:3)

# Reduced rank:
In [2]: dmatrix("C(a, MyTreat)", data)
```

```
Out[2]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, MyTreat)[MyT.a2]  C(a, MyTreat)[MyT.a3]
          1                      0                      0
          1                      1                      0
          1                      0                      1
          1                      0                      0
          1                      1                      0
          1                      0                      1
  Terms:
    'Intercept' (column 0)
    'C(a, MyTreat)' (columns 1:3)

# With argument:
In [3]: dmatrix("C(a, MyTreat(2))", data)
Out[3]:
DesignMatrix with shape (6, 3)
  Intercept  C(a, MyTreat(2))[MyT.a1]  C(a, MyTreat(2))[MyT.a2]
          1                         1                         0
          1                         0                         1
          1                         0                         0
          1                         1                         0
          1                         0                         1
          1                         0                         0
  Terms:
    'Intercept' (column 0)
    'C(a, MyTreat(2))' (columns 1:3)
```

CHAPTER

FIVE

STATEFUL TRANSFORMS

There's a subtle problem that sometimes bites people when working with formulas. Suppose that I have some numerical data called x, and I would like to center it before fitting. The obvious way would be to write:

```
y ~ I(x - np.mean(x))  # BROKEN! Don't do this!
```

or, even better we could package it up into a function:

```
In [1]: def naive_center(x):  # BROKEN! don't use!
   ...:     x = np.asarray(x)
   ...:     return x - np.mean(x)
   ...:
```

and then write our formula like:

```
y ~ naive_center(x)
```

Why is this a bad idea? Let's set up an example.

```
In [1]: import numpy as np

In [2]: from patsy import dmatrix, build_design_matrices, incr_dbuilder

In [3]: data = {"x": [1, 2, 3, 4]}
```

Now we can build a design matrix and see what we get:

```
In [1]: mat = dmatrix("naive_center(x)", data)

In [2]: mat
 Out[2]:
DesignMatrix with shape (4, 2)
  Intercept  naive_center(x)
          1             -1.5
          1             -0.5
          1              0.5
          1              1.5
  Terms:
    'Intercept' (column 0)
    'naive_center(x)' (column 1)
```

Those numbers look correct, and in fact they are correct. If all we're going to do with this model is call dmatrix() once, then everything is fine – which is what makes this problem so insidious.

Often we want to do more with a model than this. For instance, we might find some new data, and want to feed it into our model to make predictions. To do this, though, we first need to reapply the same transformation, like so:

```
In [1]: new_data = {"x": [5, 6, 7, 8]}

# Broken!
In [2]: build_design_matrices([mat.design_info.builder], new_data)[0]
 Out[2]:
DesignMatrix with shape (4, 2)
  Intercept  naive_center(x)
          1             -1.5
          1             -0.5
          1              0.5
          1              1.5
  Terms:
    'Intercept' (column 0)
    'naive_center(x)' (column 1)
```

So it's clear what's happened here – Patsy has centered the new data, just like it centered the old data. But if you think about what this means statistically, it makes no sense. According to this, the new data point where x is 5 will behave exactly like the old data point where x is 1, because they both produce the same input to the actual model.

The problem is what it means to apply "the same transformation". Here, what we really want to do is to subtract the mean *of the original data* from the new data.

Patsy's solution is called a *stateful transform*. These look like ordinary functions, but they perform a bit of magic to remember the state of the original data, and use it in transforming new data. Several useful stateful transforms are included out of the box, including one called `center()`.

Using `center()` instead of `naive_center()` produces the same correct result for our original matrix. It's used in exactly the same way:

```
In [1]: fixed_mat = dmatrix("center(x)", data)

In [2]: fixed_mat
 Out[2]:
DesignMatrix with shape (4, 2)
  Intercept  center(x)
          1       -1.5
          1       -0.5
          1        0.5
          1        1.5
  Terms:
    'Intercept' (column 0)
    'center(x)' (column 1)
```

But if we then feed in our new data, we also get out the correct result:

```
# Correct!
In [1]: build_design_matrices([fixed_mat.design_info.builder], new_data)[0]
 Out[1]:
DesignMatrix with shape (4, 2)
  Intercept  center(x)
          1        2.5
          1        3.5
          1        4.5
          1        5.5
  Terms:
    'Intercept' (column 0)
    'center(x)' (column 1)
```

Another situation where we need some stateful transform magic is when we are working with data that is too large to fit into memory at once. To handle such cases, Patsy allows you to set up a design matrix while working our way incrementally through the data. But if we use `naive_center()` when building a matrix incrementally, then it centers each *chunk* of data, not the data as a whole. (Of course, depending on how your data is distributed, this might end up being just similar enough for you to miss the problem until it's too late.)

```
In [1]: data_chunked = [{"x": data["x"][:2]},
   ...:                 {"x": data["x"][2:]}]
   ...:

In [2]: builder = incr_dbuilder("naive_center(x)", lambda: iter(data_chunked))

# Broken!
In [3]: np.row_stack([build_design_matrices([builder], chunk)[0]
   ...:               for chunk in data_chunked])
   ...:
 Out[3]:
array([[ 1. , -0.5],
       [ 1. ,  0.5],
       [ 1. , -0.5],
       [ 1. ,  0.5]])
```

But if we use the proper stateful transform, this just works:

```
In [1]: builder = incr_dbuilder("center(x)", lambda: iter(data_chunked))

# Correct!
In [2]: np.row_stack([build_design_matrices([builder], chunk)[0]
   ...:               for chunk in data_chunked])
   ...:
 Out[2]:
array([[ 1. , -1.5],
       [ 1. , -0.5],
       [ 1. ,  0.5],
       [ 1. ,  1.5]])
```

Note: Under the hood, the way this works is that `incr_dbuilder()` iterates through the data once to calculate the mean, and then we use `build_design_matrices()` to iterate through it a second time creating our design matrix. While taking two passes through a large data set may be slow, there's really no other way to accomplish what the user asked for. The good news is that Patsy is smart enough to make only the minimum number of passes necessary. For example, in our example with `naive_center()` above, `incr_dbuilder()` would not have done a full pass through the data at all. And if you have multiple stateful transforms in the same formula, then Patsy will process them in parallel in a single pass.

And, of course, we can use the resulting builder for prediction as well:

```
# Correct!
In [1]: build_design_matrices([builder], new_data)[0]
 Out[1]:
DesignMatrix with shape (4, 2)
  Intercept  center(x)
          1        2.5
          1        3.5
          1        4.5
          1        5.5
  Terms:
    'Intercept' (column 0)
    'center(x)' (column 1)
```

In fact, Patsy's stateful transform handling is clever enough that it can support arbitrary mixing of stateful transforms with other Python code. E.g., if `center()` and `spline()` were both stateful transforms, then even a silly a formula like this will be handled 100% correctly:

```
y ~ I(spline(center(x1)) + center(x2))
```

However, it isn't perfect – there are two things you have to be careful of. Let's put them in red:

Warning: If you are unwise enough to ignore this section, write a function like *naive_center* above, and use it in a formula, then Patsy will not notice. If you use that formula with `incr_dbuilders()` or for predictions, then you will just silently get the wrong results. We have a plan to detect such cases, but it isn't implemented yet (and in any case can never be 100% reliable). So be careful!

Warning: Even if you do use a "real" stateful transform like `center()` or `standardize()`, still have to make sure that Patsy can "see" that you are using such a transform. Currently the rule is that you must access the stateful transform function using a simple, bare variable reference, without any dots or other lookups:

```
dmatrix("y ~ center(x)", data)  # okay
asdf = patsy.center
dmatrix("y ~ asdf(x)", data)  # okay
dmatrix("y ~ patsy.center(x)", data)  # BROKEN! DON'T DO THIS!
funcs = {"center": patsy.center}
dmatrix("y ~ funcs['center'](x)", data)  # BROKEN! DON'T DO THIS!
```

5.1 Builtin stateful transforms

There are a number of builtin stateful transforms beyond `center()`; see *stateful transforms* in the API reference for a complete list.

5.2 Defining a stateful transform

You can also easily define your own stateful transforms. The first step is to define a class which fulfills the stateful transform protocol. The lifecycle of a stateful transform object is as follows:

1. An instance of your type will be constructed.
2. `memorize_chunk()` will be called one or more times.
3. `memorize_finish()` will be called once.
4. `transform()` will be called one or more times, on either the same or different data to what was initially passed to `memorize_chunk()`. You can trust that any non-data arguments will be identical between calls to `memorize_chunk()` and `transform()`.

And here are the methods and call signatures you need to define:

class `patsy.`**`stateful_transform_protocol`**

`__init__``()`
    It must be possible to create an instance of the class by calling the constructor with no arguments.

`memorize_chunk``(`**args*, ***kwargs*`)`
    Update any internal state, based on the data passed into *memorize_chunk*.

memorize_finish()

Do any housekeeping you want to do between the last call to `memorize_chunk()` and the first call to `transform()`. For example, if you are computing some summary statistic that cannot be done incrementally, then your `memorize_chunk()` method might just store the data that's passed in, and then `memorize_finish()` could compute the summary statistic and delete the stored data to free up the associated memory.

transform(**args*, ***kwargs*)

This method should transform the input data passed to it. It should be deterministic, and it should be "point-wise", in the sense that when passed an array it performs an independent transformation on each data point that is not affected by any other data points passed to `transform()`.

Then once you have created your class, pass it to `stateful_transform()` to create a callable stateful transform object suitable for use inside or outside formulas.

Here's a simple example of how you might implement a working version of `center()` (though it's less robust and featureful than the real builtin):

```
class MyExampleCenter(object):
    def __init__(self):
        self._total = 0
        self._count = 0
        self._mean = None

    def memorize_chunk(self, x):
        self._total += np.sum(x)
        self._count += len(x)

    def memorize_finish(self):
        self._mean = self.total * 1. / self._count

    def transform(self, x):
        return x - self._mean

my_example_center = patsy.stateful_transform(MyExampleCenter)
print(my_example_center(np.array([1, 2, 3])))
```

But of course, if you come up with any useful ones, please let us know so we can incorporate them into patsy itself!

CHAPTER

SIX

MODEL SPECIFICATION FOR EXPERTS AND COMPUTERS

While the formula language is great for interactive model-fitting and exploratory data analysis, there are times when we want a different or more systematic interface for creating design matrices. If you ever find yourself writing code that pastes together bits of strings to create a formula, then stop! And read this chapter.

Our first option, of course, is that we can go ahead and write some code to construct our design matrices directly, just like we did in the old days. Since this is supported directly by `dmatrix()` and `dmatrices()`, it also works with any third-party library functions that use Patsy internally. Just pass in an array_like or a tuple `(y_array_like, X_array_like)` in place of the formula.

```
In [1]: from patsy import dmatrix

In [2]: X = [[1, 10], [1, 20], [1, -2]]

In [3]: dmatrix(X)
Out[3]: 
DesignMatrix with shape (3, 2)
  x0  x1
   1  10
   1  20
   1  -2
  Terms:
    ’x0’ (column 0)
    ’x1’ (column 1)
```

By using a `DesignMatrix` with `DesignInfo` attached, we can also specify custom names for our custom matrix (or even term slices and so forth), so that we still get the nice output and such that Patsy would otherwise provide:

```
In [1]: from patsy import DesignMatrix, DesignInfo

In [2]: design_info = DesignInfo(["Intercept!", "Not intercept!"])

In [3]: X_dm = DesignMatrix(X, design_info)

In [4]: dmatrix(X_dm)
Out[4]: 
DesignMatrix with shape (3, 2)
  Intercept!  Not intercept!
           1              10
           1              20
           1              -2
  Terms:
    'Intercept!' (column 0)
    'Not intercept!' (column 1)
```

Or if all we want to do is to specify column names, we could also just use a `pandas.DataFrame`:

```
In [1]: import pandas

In [2]: df = pandas.DataFrame([[1, 10], [1, 20], [1, -2]],
   ...:                        columns=["Intercept!", "Not intercept!"])
   ...:

In [3]: dmatrix(df)
Out[3]:
DesignMatrix with shape (3, 2)
  Intercept!  Not intercept!
           1              10
           1              20
           1              -2
  Terms:
    'Intercept!' (column 0)
    'Not intercept!' (column 1)
```

However, there is also a middle ground between pasting together strings and going back to putting together design matrices out of string and baling wire. Patsy has a straightforward Python interface for representing the result of parsing formulas, and you can use it directly. This lets you keep Patsy's normal advantages – handling of categorical data and interactions, predictions, term tracking, etc. – while using a nice high-level Python API. An example of somewhere this might be useful is if, say, you had a GUI with a tick box next to each variable in your data set, and wanted to construct a formula containing all the variables that had been checked, and letting Patsy deal with categorical data handling. Or this would be the approach you'd take for doing stepwise regression, where you need to programatically add and remove terms.

Whatever your particular situation, the strategy is this:

1. Construct some factor objects (probably using `LookupFactor` or `EvalFactor`
2. Put them into some `Term` objects,
3. Put the `Term` objects into two lists, representing the left- and right-hand side of your formula,
4. And then wrap the whole thing up in a `ModelDesc`.

(See *How formulas work* if you need a refresher on what each of these things are.)

```
In [1]: import numpy as np

In [2]: from patsy import (ModelDesc, EvalEnvironment, Term, EvalFactor,
   ...:                     LookupFactor, demo_data, dmatrix)
   ...:

In [3]: data = demo_data("a", "x")

In [4]: env = EvalEnvironment.capture()

# LookupFactor takes a dictionary key:
In [5]: a_lookup = LookupFactor("a")

# EvalFactor takes arbitrary Python code:
In [6]: x_transform = EvalFactor("np.log(x ** 2)", env)

# First argument is empty list for dmatrix; we would need to put
# something there if we were calling dmatrices.
In [7]: desc = ModelDesc([],
   ...:                   [Term([a_lookup]),
   ...:                    Term([x_transform]),
   ...:                    Term([a_lookup, x_transform])])
   ...:

# Create the matrix (or pass 'desc' to any statistical library
# function that uses patsy.dmatrix internally):
In [8]: dmatrix(desc, data)
Out[8]:
DesignMatrix with shape (6, 4)
  a[a1]  a[a2]  np.log(x ** 2)  a[T.a2]:np.log(x ** 2)
      1      0         1.13523                 0.00000
      0      1        -1.83180                -1.83180
      1      0        -0.04298                -0.00000
      0      1         1.61375                 1.61375
      1      0         1.24926                 0.00000
      0      1        -0.04597                -0.04597
  Terms:
    'a' (columns 0:2)
    'np.log(x ** 2)' (column 2)
    'a:np.log(x ** 2)' (column 3)
```

Notice that no intercept term is included. Implicit intercepts are a feature of the formula parser, not the underlying represention. If you want an intercept, include the constant `INTERCEPT` in your list of terms (which is just sugar for `Term([])`).

Note: Another option is to just pass your term lists directly to `design_matrix_builders()`, and skip the `ModelDesc` entirely – all of the highlevel API functions like `dmatrix()` accept `DesignMatrixBuilder` objects as well as `ModelDesc` objects.

Example: say our data has 100 different numerical columns that we want to include in our design – and we also have a few categorical variables with a more complex interaction structure. Here's one solution:

```
def add_predictors(base_formula, extra_predictors):
    # Interpret formula in caller's environment:
    env = EvalEnvironment.capture(1)
    desc = ModelDesc.from_formula(base_formula, env)
    # Using LookupFactor here ensures that everything will work correctly even
    # if one of the column names in extra_columns is named like "weight.in.kg"
    # or "sys.exit()" or "LittleBobbyTables()".
    desc.rhs_termlist += [Term([LookupFactor(p)]) for p in extra_predictors]
    return desc
```

```
In [1]: extra_predictors = ["x%s" % (i,) for i in range(10)]

In [2]: desc = add_predictors("np.log(y) ~ a*b + c:d", extra_predictors)

In [3]: desc.describe()
Out[3]: 'np.log(y) ~ a + b + a:b + c:d + x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9'
```

6.1 The factor protocol

If `LookupFactor` and `EvalFactor` aren't enough for you, then you can define your own factor class.

The full interface looks like this:

class `patsy.`**`factor_protocol`**

`name``()`

This must return a short string describing this factor. It will be used to create column names, among other things.

`__eq__``(`*obj*`)`

`__ne__``(`*obj*`)`

`__hash__``()`

If your factor will ever contain categorical data or participate in interactions, then it's important to make sure you've defined `__eq__()` and `__ne__()` and that your type is *hashable*. These methods will determine which factors Patsy considers equal for purposes of redundancy elimination.

`memorize_passes_needed``(`*state*`)`

Return the number of passes through the data that this factor will need in order to set up any *Stateful transforms*.

If you don't want to support stateful transforms, just return 0. In this case, `memorize_chunk()` and `memorize_finish()` will never be called.

state is an (initially) empty dict which is maintained by the builder machinery, and that we can do whatever we like with. It will be passed back in to all memorization and evaluation methods.

`memorize_chunk``(`*state*, *which_pass*, *data*`)`

Called repeatedly with each 'chunk' of data produced by the *data_iter_maker* passed to `design_matrix_builders()`.

state is the state dictionary. *which_pass* will be zero on the first pass through the data, and eventually reach the value you returned from `memorize_passes_needed()`, minus one.

Return value is ignored.

`memorize_finish``(`*state*, *which_pass*`)`

Called once after each pass through the data.

Return value is ignored.

`eval``(`*state*, *data*`)`

Evaluate this factor on the given *data*. Return value should ideally be a 1-d or 2-d array or `Categorical()` object, but this will be checked and converted as needed.

Warning: Do not store evaluation-related state in attributes of your factor object! The same factor object may appear in two totally different formulas, or if you have two factor objects which compare equally, then only one may be executed, and which one this is may vary randomly depending on how `build_design_matrices()` is called! Use only the *state* dictionary for storing state.

The lifecycle of a factor object therefore looks like:

1. Initialized.
2. `memorize_passes_needed()` is called.
3. `for i in range(passes_needed):`
   (a) `memorize_chunk()` is called one or more times
   (b) `memorize_finish()` is called
4. `eval()` is called zero or more times.

6.2 Alternative formula implementations

Even if you hate Patsy's formulas all together, to the extent that you're going to go and implement your own competing mechanism for defining formulas, you can still Patsy-based interfaces. Unfortunately, this isn't *quite* as clean as we'd like, because for now there's no way to define a custom `DesignMatrixBuilder`. So you do still have to go through Patsy's formula-building machinery. But, this machinery simply passes numerical data through unchanged, so in extremis you can:

- Define a special factor object that simply defers to your existing machinery
- Define the magic `__patsy_get_model_desc__` method on your formula object. `dmatrix()` and friends check for the presence of this method on any object that is passed in, and if found, it is called (passing in the `EvalEnvironment`), and expected to return a `ModelDesc`. And your `ModelDesc` can, of course, include your special factor object(s).

Put together, it looks something like this:

```
class MyAlternativeFactor(object):
    # A factor object that simply returns the design
    def __init__(self, alternative_formula, side):
        self.alternative_formula = alternative_formula
        self.side = side

    def name(self):
        return self.side

    def memorize_passes_needed(self, state):
        return 0

    def eval(self, state, data):
        return self.alternative_formula.get_matrix(self.side, data)

class MyAlternativeFormula(object):
    ...

    def __patsy_get_model_desc__(self, eval_env):
        return ModelDesc([Term([MyAlternativeFactor(self, side="left")])],
                         [Term([MyAlternativeFactor(self, side="right")])],


my_formula = MyAlternativeFormula(...)
dmatrix(my_formula, data)
```

The only downside to this approach is that you can't control the names of individual columns. (A workaround would be to create multiple terms each with its own factor that returns a different pieces of your overall matrix.) If this is a problem for you, though, then let's talk – we can probably work something out.

CHAPTER

SEVEN

USING PATSY IN YOUR LIBRARY

Our goal is to make Patsy the de facto standard for describing models in Python, regardless of the underlying package in use – just as formulas are the standard interface to all R packages. Therefore we've tried to make it as easy as possible for you to build Patsy support into your libraries.

Patsy is a good houseguest:

- Pure Python, no compilation necessary.
- Exhaustive tests (>98% statement coverage at time of writing) and documentation (you're looking at it).
- No dependencies besides numpy (and we even test against numpy 1.2.1, as distributed by RHEL 5).
- Tested and supported on every version of Python since 2.4.

So you can be pretty confident that adding a dependency on Patsy won't create much hassle for your users.

And, of course, the fundamental design is very conservative – the formula mini-language in S was first described in Chambers and Hastie (1992), more than two decades ago. It's still in heavy use today in R, which is one of the most popular environments for statistical programming. Many of your users may already be familiar with it. So we can be pretty certain that it will hold up to real-world usage.

7.1 Using the high-level interface

If you have a function whose signature currently looks like this:

```
def mymodel2(X, y, ...):
    ...
```

or this:

```
def mymodel1(X, ...):
    ...
```

then adding Patsy support is extremely easy (though of course like any other API change, you may have to deprecate the old interface, or provide two interfaces in parallel, depending on your situation). Just write something like:

```
def mymodel2_patsy(formula_like, data={}, ...):
    y, X = patsy.dmatrices(formula_like, data, 1)
    ...
```

or:

```
def mymodel1_patsy(formula_like, data={}, ...):
    X = patsy.dmatrix(formula_like, data, 1)
    ...
```

(See dmatrices() and dmatrix() for details.) This won't force your users to switch to formulas immediately; they can replace code that looks like this:

```
X, y = build_matrices_laboriously()
result = mymodel2(X, y, ...)
other_result = mymodel1(X, ...)
```

with code like this:

```
X, y = build_matrices_laboriously()
result = mymodel2((y, X), data=None, ...)
other_result = mymodel1(X, data=None, ...)
```

Of course in the long run they might want to throw away that `build_matrices_laboriously()` function and start using formulas, but they aren't forced to just to start using your new interface.

7.1.1 Working with metadata

Once you've started using Patsy to handle formulas, you'll probably want to take advantage of the metadata that Patsy provides, so that you can display regression coefficients by name and so forth. Design matrices processed by Patsy always have a `.design_info` attribute which contains lots of information about the design: see DesignInfo for details.

7.1.2 Predictions

Another nice feature is making predictions on new data. But this requires that we can take in new data, and transform it to create a new *X* matrix. Or if we want to compute the likelihood of our model on new data, we need both new *X* and *y* matrices.

This is also easily done with Patsy – first fetch the relevant DesignMatrixBuilder objects by doing `input_data.design_info.builder`, and then pass them to build_design_matrices() along with the new data.

7.1.3 Example

Here's a simplified class for doing ordinary least-squares regression, demonstrating the above techniques:

Warning: This code has not been validated for numerical correctness.

```
import numpy as np
from patsy import dmatrices, build_design_matrices

class LM(object):
    """An example ordinary least squares linear model class, analogous to R's
    lm() function. Don't use this in real life, it isn't properly tested."""
    def __init__(self, formula_like, data={}):
        y, x = dmatrices(formula_like, data, 1)
        self.nobs = x.shape[0]
        self.betas, self.rss, _, _ = np.linalg.lstsq(x, y)
        self._y_design_info = y.design_info
        self._x_design_info = x.design_info

    def __repr__(self):
        summary = ("Ordinary least-squares regression\n"
                   "  Model: %s ~ %s\n"
                   "  Regression (beta) coefficients:\n"
                   % (self._y_design_info.describe(),
                      self._x_design_info.describe()))
        for name, value in zip(self._x_design_info.column_names, self.betas):
            summary += "    %s:  %0.3g\n" % (name, value[0])
        return summary

    def predict(self, new_data):
        (new_x,) = build_design_matrices([self._x_design_info.builder],
                                         new_data)
        return np.dot(new_x, self.betas)

    def loglik(self, new_data):
        (new_y, new_x) = build_design_matrices([self._y_design_info.builder,
                                                self._x_design_info.builder],
                                               new_data)
        new_pred = np.dot(new_x, self.betas)
        sigma2 = self.rss / self.nobs
        # It'd be more elegant to use scipy.stats.norm.logpdf here, but adding
        # a dependency on scipy makes the docs build more complicated:
        Z = -0.5 * np.log(2 * np.pi * sigma2)
        return Z + -0.5 * (new_y - new_x) ** 2/sigma2
```

And here's how it can be used:

```
In [1]: from patsy import demo_data

In [2]: data = demo_data("x", "y", "a")

# Old and boring approach (but it still works):
In [3]: X = np.column_stack(([1] * len(data["y"]), data["x"]))

In [4]: LM((data["y"], X))
Out[4]:
Ordinary least-squares regression
  Model: y0 ~ x0 + x1
  Regression (beta) coefficients:
    x0:  0.677
    x1:  -0.217


# Fancy new way:
In [5]: m = LM("y ~ x", data)

In [6]: m
Out[6]:
Ordinary least-squares regression
  Model: y ~ 1 + x
  Regression (beta) coefficients:
    Intercept:  0.677
    x:  -0.217


In [7]: m.predict({"x": [10, 20, 30]})
Out[7]:
array([[-1.48944498],
       [-3.65620297],
       [-5.82296096]])

In [8]: m.loglik(data)
Out[8]:
array([[ -0.28884193,  -1.46289596],
       [ -2.64235743,  -0.8254485 ],
       [ -2.44930737,  -2.36666465],
       [ -0.90233651,  -6.24317017],
       [ -1.58762894,  -5.56817766],
       [ -0.65148056, -10.80114045]])

In [9]: m.loglik({"x": [10, 20, 30], "y": [-1, -2, -3]})
Out[9]:
array([[   -7.39939265,  -215.51261221],
       [  -16.29311998,  -861.19721649],
       [  -28.74433824, -1937.33822362]])

# Your users get support for categorical predictors for free:
In [10]: LM("y ~ a", data)
Out[10]:
Ordinary least-squares regression
  Model: y ~ 1 + a
  Regression (beta) coefficients:
    Intercept:  0.33
    a[T.a2]:  0.241


# And variable transformations too:
In [11]: LM("y ~ np.log(x ** 2)", data)
Out[11]:
Ordinary least-squares regression
  Model: y ~ 1 + np.log(x ** 2)
  Regression (beta) coefficients:
    Intercept:  0.399
    np.log(x ** 2):  0.148
```

7.1.4 Other cool tricks

If you want to compute ANOVAs, then check out `DesignInfo.term_name_slices`, `DesignInfo.slice()`.

If you support linear hypothesis tests or otherwise allow your users to specify linear constraints on model parameters, consider taking advantage of `DesignInfo.linear_constraint()`.

7.2 Extending the formula syntax

The above documentation assumes that you have a relatively simple model that can be described by one or two matrices (plus whatever other arguments you take). This covers many of the most popular models, but it's definitely not sufficient for every model out there.

Internally, Patsy is designed to be very flexible – for example, it's quite straightforward to add custom operators to the formula parser, or otherwise extend the formula evaluation machinery. (Heck, it only took an hour or two to repurpose it for a totally different purpose, parsing linear constraints.) But extending Patsy in a more fundamental way this will require just a wee bit more complicated API than just calling `dmatrices()`, and for this initial release, we've been busy enough getting the basics working that we haven’t yet taken the time to pin down a public extension API we can support.

So, if you want something fancier – please give us a nudge, it’s entirely likely we can work something out.

CHAPTER

EIGHT

DIFFERENCES BETWEEN R AND PATSY FORMULAS

Patsy has a very high degree of compatibility with R. Almost any formula you would use in R will also work in Patsy – with a few caveats.

Note: All R quirks described herein were last verified with R 2.15.0.

Differences from R:

- Most obviously, we both support using arbitrary code to perform variable transformations, but in Patsy this code is written in Python, not R.
- Patsy has no `%in%`. In R, `a %in% b` is identical to `b:a`. Patsy only supports the `b:a` version of this syntax.
- In Patsy, only `**` can be used for exponentiation. In R, both `^` and `**` can be used for exponentiation, i.e., you can write either `(a + b)^2` or `(a + b)**2`. In Patsy (as in Python generally), only `**` indicates exponentiation; `^` is ignored by the parser (and if present, will be interpreted as a call to the Python binary XOR operator).
- In Patsy, the left-hand side of a formula uses the same evaluation rules as the right-hand side. In R, the left hand side is treated as R code, so a formula like `y1 + y2 ~ x1 + x2` actually regresses the *sum* of `y1` and `y2` onto the *set of predictors* `x1` and `x2`. In Patsy, the only difference between the left-hand side and the right-hand side is that there is no automatic intercept added to the left-hand side. (In this regard Patsy is similar to the R enhanced formula package Formula.)
- Patsy produces a different column ordering for formulas involving numeric predictors. In R, there are two rules for term ordering: first, lower-order interactions are sorted before higher-order interactions, and second, interactions of the same order are listed in whatever order they appeared in the formula. In Patsy, we add another rule: terms are first grouped together based on which numeric factors they include. Then within each group, we use the same ordering as R.
- Patsy has more rigorous handling of the presence or absence of the intercept term. In R, the rules for when deciding whether to include an intercept are somewhat idiosyncratic and can ignore things like parentheses. To understand the difference, first consider the formula `a + (b - a)`. In both Patsy and R, we first evaluate the `(b - a)` part; since there is no `a` term to remove, this simplifies to just `b`. We then evaluate `a + b`: the end result is a model which contains an `a` term in it.

  Now consider the formula `1 + (b - 1)`. In Patsy, this is analogous to the case above: first `(b - 1)` is reduced to just `b`, and then `1 + b` produces a model with intercept included. In R, the parentheses are ignored, and `1 + (b - 1)` gives a model that does *not* include the intercept.

  This can be slightly more confusing when it comes to the implicit intercept term. In Patsy, this is handled exactly as if the right-hand side of each formula has an invisible `"1 +"` inserted at the beginning. Therefore in Patsy, these formulas are different:

```
# Python:
dmatrices("y ~ b - 1")    # equivalent to 1 + b - 1: no intercept
dmatrices("y ~ (b - 1)") # equivalent to 1 + (b - 1): has intercept
```

In R, these two formulas are equivalent.

- Patsy has a more accurate algorithm for deciding whether to use a full- or reduced-rank coding scheme for categorical factors. There are two situations in which R's coding algorithm for categorical variables can become confused and produce over- or under-specified model matrices. Patsy, so far as we are aware, produces correctly specified matrices in all cases. It's unlikely that you'll run into these in actual usage, but they're worth mentioning. To illustrate, let's define `a` and `b` as categorical predictors, each with 2 levels:

```
# R:
> a <- factor(c("a1", "a1", "a2", "a2"))
> b <- factor(c("b1", "b2", "b1", "b2"))
```

The first problem occurs for formulas like `1 + a:b`. This produces a model matrix with rank 4, just like many other formulas that include `a:b`, such as `0 + a:b`, `1 + a + a:b`, and `a*b`:

```
# R:
> qr(model.matrix(~ 1 + a:b))$rank
[1] 4
```

However, the matrix produced for this formula has 5 columns, meaning that it contains redundant overspecification:

```
# R:
> mat <- model.matrix(~ 1 + a:b)
> ncol(mat)
[1] 5
```

The underlying problem is that R's algorithm does not pay attention to 'non-local' redundancies – it will adjust its coding to avoid a redundancy between two terms of degree-n, or a term of degree-n and one of degree-(n+1), but it is blind to a redundancy between a term of degree-n and one of degree-(n+2), as we have here.

Patsy's algorithm has no such limitation:

```
# Python:
In [1]: a = ["a1", "a1", "a2", "a2"]

In [2]: b = ["b1", "b2", "b1", "b2"]

In [3]: mat = dmatrix("1 + a:b")

In [4]: mat.shape[1]
Out[4]: 4
```

To produce this result, it codes `a:b` uses the same columns that would be used to code `b + a:b` in the formula `"1 + b + a:b"`.

The second problem occurs for formulas involving numeric predictors. Effectively, when determining coding schemes, R assumes that all factors are categorical. So for the formula `0 + a:c + a:b`, R will notice that if it used a full-rank coding for the `c` and `b` factors, then both terms would be collinear with `a`, and thus each other. Therefore, it encodes `c` with a full-rank encoding, and uses a reduced-rank encoding for `b`. (And the `0 +` lets it avoid the previous bug.) So far, so good.

But now consider the formula `0 + a:x + a:b`, where `x` is numeric. Here, `a:x` and `a:b` will not be collinear, even if we do use a full-rank encoding for `b`. Therefore, we *should* use a full-rank encoding for `b`, and produce a model matrix with 6 columns. But in fact, R gives us only 4:

```
# R:
> x <- c(1, 2, 3, 4)
> mat <- model.matrix(~ 0 + a:x + a:b)
> ncol(mat)
[1] 4
```

The problem is that it cannot tell the difference between `0 + a:x + a:b` and `0 + a:c + a:b`: it uses the same coding for both, whether it's appropriate or not.

(The alert reader might wonder whether this bug could be triggered by a simpler formula, like `0 + x + b`. It turns out that R's code `do_modelmatrix` function has a special-case where for first-order interactions only, it *will* peek at the type of the data before deciding on a coding scheme.)

Patsy always checks whether each factor is categorical or numeric before it makes coding decisions, and thus handles this case correctly:

```
# Python:
In [1]: x = [1, 2, 3, 4]

In [2]: mat = dmatrix("0 + a:x + a:b")

In [3]: mat.shape[1]
Out[3]: 6
```

CHAPTER

NINE

PYTHON 2 VERSUS PYTHON 3

The biggest difference between Python 2 and Python 3 is in their string handling, and this is particularly relevant to Patsy since it parses user input. We follow a simple rule: input to Patsy should always be of type *str*. That means that on Python 2, you should pass byte-strings (not unicode), and on Python 3, you should pass unicode strings (not byte-strings). Similarly, when Patsy passes text back (e.g. `DesignInfo.column_names`), it's always in the form of a *str*.

In addition to this being the most convenient for users (you never need to use any b"weird" u"prefixes" when writing a formula string), it's actually a necessary consequence of a deeper change in the Python language: in Python 2, Python code itself is represented as byte-strings, and that's the only form of input accepted by the `tokenize` module. On the other hand, Python 3's tokenizer and parser use unicode, and since Patsy processes Python code, it has to follow suit.

CHAPTER

TEN

PATSY API REFERENCE

This is a complete reference for everything you get when you *import patsy*.

10.1 Basic API

`patsy.`**`dmatrix`**(*formula_like*, *data={}*, *eval_env=0*, *NA_action='drop'*, *return_type='matrix'*)

Construct a single design matrix given a formula_like and data.

Parameters

- **formula_like** – An object that can be used to construct a design matrix. See below.
- **data** – A dict-like object that can be used to look up variables referenced in *formula_like*.
- **eval_env** – Either a `EvalEnvironment` which will be used to look up any variables referenced in *formula_like* that cannot be found in *data*, or else a depth represented as an integer which will be passed to `EvalEnvironment.capture()`. `eval_env=0` means to use the context of the function calling `dmatrix()` for lookups. If calling this function from a library, you probably want `eval_env=1`, which means that variables should be resolved in *your* caller's namespace.
- **NA_action** – What to do with rows that contain missing values. You can `"drop"` them, `"raise"` an error, or for customization, pass an `NAAction` object. See `NAAction` for details on what values count as 'missing' (and how to alter this).
- **return_type** – Either `"matrix"` or `"dataframe"`. See below.

The *formula_like* can take a variety of forms. You can use any of the following:

- (The most common option) A formula string like `"x1 + x2"` (for `dmatrix()`) or `"y ~ x1 + x2"` (for `dmatrices()`). For details see *How formulas work*.
- A `ModelDesc`, which is a Python object representation of a formula. See *How formulas work* and *Model specification for experts and computers* for details.
- A `DesignMatrixBuilder`.
- An object that has a method called `__patsy_get_model_desc__()`. For details see *Model specification for experts and computers*.
- A numpy array_like (for `dmatrix()`) or a tuple (array_like, array_like) (for `dmatrices()`). These will have metadata added, representation normalized, and then be returned directly. In this case *data* and *eval_env* are ignored. There is special handling for two cases:
  - `DesignMatrix` objects will have their `DesignInfo` preserved. This allows you to set up custom column names and term information even if you aren't using the rest of the patsy machinery.

–pandas.DataFrame or pandas.Series objects will have their (row) indexes checked. If two are passed in, their indexes must be aligned. If return_type="dataframe", then their indexes will be preserved on the output.

Regardless of the input, the return type is always either:

- A DesignMatrix, if return_type="matrix" (the default)
- A pandas.DataFrame, if return_type="dataframe".

The actual contents of the design matrix is identical in both cases, and in both cases a DesignInfo object will be available in a .design_info attribute on the return value. However, for return_type="dataframe", any pandas indexes on the input (either in *data* or directly passed through *formula_like*) will be preserved, which may be useful for e.g. time-series models. New in version 0.2.0: The NA_action argument.

patsy.**dmatrices** (*formula_like*, *data={}*, *eval_env=0*, *NA_action='drop'*, *return_type='matrix'*)

Construct two design matrices given a formula_like and data.

This function is identical to dmatrix(), except that it requires (and returns) two matrices instead of one. By convention, the first matrix is the "outcome" or "y" data, and the second is the "predictor" or "x" data.

See dmatrix() for details.

patsy.**incr_dbuilders** (*formula_like*, *data_iter_maker*, *eval_env=0*, *NA_action='drop'*)

Construct two design matrix builders incrementally from a large data set.

incr_dbuilders() is to incr_dbuilder() as dmatrices() is to dmatrix(). See incr_dbuilder() for details.

patsy.**incr_dbuilder** (*formula_like*, *data_iter_maker*, *eval_env=0*, *NA_action='drop'*)

Construct a design matrix builder incrementally from a large data set.

Parameters

- **formula_like** – Similar to dmatrix(), except that explicit matrices are not allowed. Must be a formula string, a ModelDesc, a DesignMatrixBuilder, or an object with a __patsy_get_model_desc__ method.
- **data_iter_maker** – A zero-argument callable which returns an iterator over dict-like data objects. This must be a callable rather than a simple iterator because sufficiently complex formulas may require multiple passes over the data (e.g. if there are nested stateful transforms).
- **eval_env** – Either a EvalEnvironment which will be used to look up any variables referenced in *formula_like* that cannot be found in *data*, or else a depth represented as an integer which will be passed to EvalEnvironment.capture(). eval_env=0 means to use the context of the function calling incr_dbuilder() for lookups. If calling this function from a library, you probably want eval_env=1, which means that variables should be resolved in *your* caller's namespace.
- **NA_action** – An NAAction object or string, used to determine what values count as 'missing' for purposes of determining the levels of categorical factors.

Returns A DesignMatrixBuilder

Tip: for *data_iter_maker*, write a generator like:

```
def iter_maker():
    for data_chunk in my_data_store:
        yield data_chunk
```

and pass *iter_maker* (*not iter_maker()*). New in version 0.2.0: The NA_action argument.

exception `patsy.PatsyError`(*message*, *origin=None*)

This is the main error type raised by Patsy functions.

In addition to the usual Python exception features, you can pass a second argument to this function specifying the origin of the error; this is included in any error message, and used to help the user locate errors arising from malformed formulas. This second argument should be an `Origin` object, or else an arbitrary object with a `.origin` attribute. (If it is neither of these things, then it will simply be ignored.)

For ordinary display to the user with default formatting, use `str(exc)`. If you want to do something cleverer, you can use the `.message` and `.origin` attributes directly. (The latter may be None.)

10.2 Convenience utilities

`patsy.balanced`(*factor_name=num_levels*[, *factor_name=num_levels*, ..., *repeat=1*])

Create simple balanced factorial designs for testing.

Given some factor names and the number of desired levels for each, generates a balanced factorial design in the form of a data dictionary. For example:

```
In [1]: balanced(a=2, b=3)
Out[1]:
{'a': ['a1', 'a1', 'a1', 'a2', 'a2', 'a2'],
 'b': ['b1', 'b2', 'b3', 'b1', 'b2', 'b3']}
```

By default it produces exactly one instance of each combination of levels, but if you want multiple replicates this can be accomplished via the *repeat* argument:

```
In [1]: balanced(a=2, b=2, repeat=2)
Out[1]:
{'a': ['a1', 'a1', 'a2', 'a2', 'a1', 'a1', 'a2', 'a2'],
 'b': ['b1', 'b2', 'b1', 'b2', 'b1', 'b2', 'b1', 'b2']}
```

`patsy.demo_data`(**names*, *nlevels=2*, *min_rows=5*)

Create simple categorical/numerical demo data.

Pass in a set of variable names, and this function will return a simple data set using those variable names.

Names whose first letter falls in the range "a" through "m" will be made categorical (with *nlevels* levels). Those that start with a "p" through "z" are numerical.

We attempt to produce a balanced design on the categorical variables, repeating as necessary to generate at least *min_rows* data points. Categorical variables are returned as a list of strings.

Numerical data is generated by sampling from a normal distribution. A fixed random seed is used, so that identical calls to demo_data() will produce identical results. Numerical data is returned in a numpy array.

Example:

10.3 Design metadata

class `patsy.DesignInfo`(*column_names*, *term_slices=None*, *term_name_slices=None*, *builder=None*)

A DesignInfo object holds metadata about a design matrix.

This is the main object that Patsy uses to pass information to statistical libraries. Usually encountered as the *.design_info* attribute on design matrices.

Here's an example of the most common way to get a `DesignInfo`:

```
In [1]: mat = dmatrix("a + x", demo_data("a", "x", nlevels=3))

In [2]: di = mat.design_info
```

column_names

The names of each column, represented as a list of strings in the proper order. Guaranteed to exist.

```
In [1]: di.column_names
Out[1]: ['Intercept', 'a[T.a2]', 'a[T.a3]', 'x']
```

column_name_indexes

An `OrderedDict` mapping column names (as strings) to column indexes (as integers). Guaranteed to exist and to be sorted from low to high.

```
In [1]: di.column_name_indexes
Out[1]: OrderedDict([('Intercept', 0), ('a[T.a2]', 1), ('a[T.a3]', 2), ('x', 3)])
```

term_names

The names of each term, represented as a list of strings in the proper order. Guaranteed to exist. There is a one-to-many relationship between columns and terms – each term generates one or more columns.

```
In [1]: di.term_names
Out[1]: ['Intercept', 'a', 'x']
```

term_name_slices

An `OrderedDict` mapping term names (as strings) to Python `slice()` objects indicating which columns correspond to each term. Guaranteed to exist. The slices are guaranteed to be sorted from left to right and to cover the whole range of columns with no overlaps or gaps.

```
In [1]: di.term_name_slices
Out[1]: OrderedDict([('Intercept', slice(0, 1, None)), ('a', slice(1, 3, None)), ('x', slice
```

terms

A list of `Term` objects representing each term. May be None, for example if a user passed in a plain preassembled design matrix rather than using the Patsy machinery.

```
In [1]: di.terms
Out[1]: [Term([]), Term([EvalFactor('a')]), Term([EvalFactor('x')])]

In [2]: [term.name() for term in di.terms]
Out[2]: ['Intercept', 'a', 'x']
```

term_slices

An `OrderedDict` mapping `Term` objects to Python `slice()` objects indicating which columns correspond to which terms. Like `terms`, this may be None.

```
In [1]: di.term_slices
Out[1]: OrderedDict([(Term([]), slice(0, 1, None)), (Term([EvalFactor('a')]), slice(1, 3, No
```

builder

A `DesignMatrixBuilder` object that can be used to generate more design matrices of this type (e.g. for prediction). May be None.

A number of convenience methods are also provided that take advantage of the above metadata:

describe()

Returns a human-readable string describing this design info.

Example:

```
In [1]: y, X = dmatrices("y ~ x1 + x2", demo_data("y", "x1", "x2"))

In [2]: y.design_info.describe()
Out[2]: ’y’

In [3]: X.design_info.describe()
Out[3]: ’1 + x1 + x2’
```

Warning: There is no guarantee that the strings returned by this function can be parsed as formulas. They are best-effort descriptions intended for human users.

linear_constraint (*constraint_likes*)

Construct a linear constraint in matrix form from a (possibly symbolic) description.

Possible inputs:

- A dictionary which is taken as a set of equality constraint. Keys can be either string column names, or integer column indexes.
- A string giving a arithmetic expression referring to the matrix columns by name.
- A list of such strings which are ANDed together.
- A tuple (A, b) where A and b are array_likes, and the constraint is Ax = b. If necessary, these will be coerced to the proper dimensionality by appending dimensions with size 1.

The string-based language has the standard arithmetic operators, / * + - and parentheses, plus “=” is used for equality and ”,” is used to AND together multiple constraint equations within a string. You can If no = appears in some expression, then that expression is assumed to be equal to zero. Division is always float-based, even if `__future__.true_division` isn’t in effect.

Returns a `LinearConstraint` object.

Examples:

```
di = DesignInfo(["x1", "x2", "x3"])

# Equivalent ways to write x1 == 0:
di.linear_constraint({"x1": 0})  # by name
di.linear_constraint({0: 0})  # by index
di.linear_constraint("x1 = 0")  # string based
di.linear_constraint("x1")  # can leave out "= 0"
di.linear_constraint("2 * x1 = (x1 + 2 * x1) / 3")
di.linear_constraint(([1, 0, 0], 0))  # constraint matrices

# Equivalent ways to write x1 == 0 and x3 == 10
di.linear_constraint({"x1": 0, "x3": 10})
di.linear_constraint({0: 0, 2: 10})
di.linear_constraint({0: 0, "x3": 10})
di.linear_constraint("x1 = 0, x3 = 10")
di.linear_constraint("x1, x3 = 10")
di.linear_constraint(["x1", "x3 = 0"])  # list of strings
di.linear_constraint("x1 = 0, x3 - 10 = x1")
di.linear_constraint([[1, 0, 0], [0, 0, 1]], [0, 10])

# You can also chain together equalities, just like Python:
di.linear_constraint("x1 = x2 = 3")
```

slice (*columns_specifier*)

Locate a subset of design matrix columns, specified symbolically.

A patsy design matrix has two levels of structure: the individual columns (which are named), and the *terms* in the formula that generated those columns. This is a one-to-many relationship: a single term may span several columns. This method provides a user-friendly API for locating those columns.

(While we talk about columns here, this is probably most useful for indexing into other arrays that are derived from the design matrix, such as regression coefficients or covariance matrices.)

The *columns_specifier* argument can take a number of forms:

- A term name
- A column name
- A `Term` object
- An integer giving a raw index
- A raw slice object

In all cases, a Python `slice()` object is returned, which can be used directly for indexing.

Example:

```
y, X = dmatrices("y ~ a", demo_data("y", "a", nlevels=3))
betas = np.linalg.lstsq(X, y)[0]
a_betas = betas[X.design_info.slice("a")]
```

(If you want to look up a single individual column by name, use `design_info.column_name_indexes[name]`.)

classmethod **`from_array`**(*array_like*, *default_column_prefix='column'*)

Find or construct a DesignInfo appropriate for a given array_like.

If the input *array_like* already has a `.design_info` attribute, then it will be returned. Otherwise, a new DesignInfo object will be constructed, using names either taken from the *array_like* (e.g., for a pandas DataFrame with named columns), or constructed using *default_column_prefix*.

This is how `dmatrix()` (for example) creates a DesignInfo object if an arbitrary matrix is passed in.

Parameters

- **array_like** – An ndarray or pandas container.
- **default_column_prefix** – If it's necessary to invent column names, then this will be used to construct them.

Returns a DesignInfo object

class `patsy.`**`DesignMatrix`**

A simple numpy array subclass that carries design matrix metadata.

`design_info`

A `DesignInfo` object containing metadata about this design matrix.

This class also defines a fancy __repr__ method with labeled columns. Otherwise it is identical to a regular numpy ndarray.

> **Warning:** You should never check for this class using `isinstance()`. Limitations of the numpy API mean that it is impossible to prevent the creation of numpy arrays that have type DesignMatrix, but that are not actually design matrices (and such objects will behave like regular ndarrays in every way). Instead, check for the presence of a `.design_info` attribute – this will be present only on "real" DesignMatrix objects.

static `__new__`(*input_array*, *design_info=None*, *default_column_prefix='column'*)

Create a DesignMatrix, or cast an existing matrix to a DesignMatrix.

A call like:

```
DesignMatrix(my_array)
```

will convert an arbitrary array_like object into a DesignMatrix.

The return from this function is guaranteed to be a two-dimensional ndarray with a real-valued floating point dtype, and a `.design_info` attribute which matches its shape. If the *design_info* argument is not given, then one is created via `DesignInfo.from_array()` using the given *default_column_prefix*.

Depending on the input array, it is possible this will pass through its input unchanged, or create a view.

10.4 Stateful transforms

Patsy comes with a number of *stateful transforms* built in:

`patsy.center(x)`

A stateful transform that centers input data, i.e., subtracts the mean.

If input has multiple columns, centers each column separately.

Equivalent to `standardize(x, rescale=False)`

`patsy.standardize(x, center=True, rescale=True, ddof=0)`

A stateful transform that standardizes input data, i.e. it subtracts the mean and divides by the sample standard deviation.

Either centering or rescaling or both can be disabled by use of keyword arguments. The *ddof* argument controls the delta degrees of freedom when computing the standard deviation (cf. `numpy.std()`). The default of `ddof=0` produces the maximum likelihood estimate; use `ddof=1` if you prefer the square root of the unbiased estimate of the variance.

If input has multiple columns, standardizes each column separately.

Note: This function computes the mean and standard deviation using a memory-efficient online algorithm, making it suitable for use with large incrementally processed data-sets.

`patsy.scale(x, center=True, rescale=True, ddof=0)`

An alias for `standardize()`, for R compatibility.

Finally, this is not itself a stateful transform, but it's useful if you want to define your own:

`patsy.stateful_transform(class_)`

Create a stateful transform callable object from a class that fulfills the *stateful transform protocol*.

10.5 Handling categorical data

class `patsy.Treatment(reference=None)`

Treatment coding (also known as dummy coding).

This is the default coding.

For reduced-rank coding, one level is chosen as the “reference”, and its mean behaviour is represented by the intercept. Each column of the resulting matrix represents the difference between the mean of one level and this reference level.

For full-rank coding, classic “dummy” coding is used, and each column of the resulting matrix represents the mean of the corresponding level.

The reference level defaults to the first level, or can be specified explicitly.

```
# reduced rank
In [1]: dmatrix("C(a, Treatment)", balanced(a=3))
Out[1]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment)[T.a2]  C(a, Treatment)[T.a3]
          1                      0                      0
          1                      1                      0
          1                      0                      1
  Terms:
    'Intercept' (column 0)
    'C(a, Treatment)' (columns 1:3)

# full rank
In [2]: dmatrix("0 + C(a, Treatment)", balanced(a=3))
Out[2]:
DesignMatrix with shape (3, 3)
  C(a, Treatment)[a1]  C(a, Treatment)[a2]  C(a, Treatment)[a3]
                    1                    0                    0
                    0                    1                    0
                    0                    0                    1
  Terms:
    'C(a, Treatment)' (columns 0:3)

# Setting a reference level
In [3]: dmatrix("C(a, Treatment(1))", balanced(a=3))
Out[3]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment(1))[T.a1]  C(a, Treatment(1))[T.a3]
          1                         1                         0
          1                         0                         0
          1                         0                         1
  Terms:
    'Intercept' (column 0)
    'C(a, Treatment(1))' (columns 1:3)

In [4]: dmatrix("C(a, Treatment('a2'))", balanced(a=3))
Out[4]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment('a2'))[T.a1]  C(a, Treatment('a2'))[T.a3]
          1                            1                            0
          1                            0                            0
          1                            0                            1
  Terms:
    'Intercept' (column 0)
    "C(a, Treatment('a2'))" (columns 1:3)
```

Equivalent to R `contr.treatment`. The R documentation suggests that using `Treatment(reference=-1)` will produce contrasts that are “equivalent to those produced by many (but not all) SAS procedures”.

class `patsy.`**`Diff`**

Backward difference coding.

This coding scheme is useful for ordered factors, and compares the mean of each level with the preceding level. So you get the second level minus the first, the third level minus the second, etc.

For full-rank coding, a standard intercept term is added (which gives the mean value for the first level).

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Diff)", balanced(a=3))
Out[1]: 
DesignMatrix with shape (3, 3)
  Intercept  C(a, Diff)[D.a1]  C(a, Diff)[D.a2]
          1          -0.66667          -0.33333
          1           0.33333          -0.33333
          1           0.33333           0.66667
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Diff)’ (columns 1:3)

# Full rank
In [2]: dmatrix("0 + C(a, Diff)", balanced(a=3))
Out[2]: 
DesignMatrix with shape (3, 3)
  C(a, Diff)[D.a1]  C(a, Diff)[D.a2]  C(a, Diff)[D.a3]
                 1          -0.66667          -0.33333
                 1           0.33333          -0.33333
                 1           0.33333           0.66667
  Terms:
    ’C(a, Diff)’ (columns 0:3)
```

class `patsy.`**`Poly`**(*scores=None*)

Orthogonal polynomial contrast coding.

This coding scheme treats the levels as ordered samples from an underlying continuous scale, whose effect takes an unknown functional form which is Taylor-decomposed into the sum of a linear, quadratic, etc. components.

For reduced-rank coding, you get a linear column, a quadratic column, etc., up to the number of levels provided.

For full-rank coding, the same scheme is used, except that the zero-order constant polynomial is also included. I.e., you get an intercept column included as part of your categorical term.

By default the levels are treated as equally spaced, but you can override this by providing a value for the *scores* argument.

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Poly)", balanced(a=4))
Out[1]: 
DesignMatrix with shape (4, 4)
  Intercept  C(a, Poly).Linear  C(a, Poly).Quadratic  C(a, Poly).Cubic
          1          -0.67082                   0.5          -0.22361
          1          -0.22361                  -0.5           0.67082
          1           0.22361                  -0.5          -0.67082
          1           0.67082                   0.5           0.22361
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Poly)’ (columns 1:4)

# Full rank
```

```
In [2]: dmatrix("0 + C(a, Poly)", balanced(a=3))
Out[2]:
DesignMatrix with shape (3, 3)
  C(a, Poly).Constant  C(a, Poly).Linear  C(a, Poly).Quadratic
                    1           -0.70711               0.40825
                    1           -0.00000              -0.81650
                    1            0.70711               0.40825
  Terms:
    'C(a, Poly)' (columns 0:3)

# Explicit scores
In [3]: dmatrix("C(a, Poly([1, 2, 10]))", balanced(a=3))
Out[3]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Poly([1, 2, 10])).Linear  C(a, Poly([1, 2, 10])).Quadratic
          1                       -0.47782                           0.66208
          1                       -0.33447                          -0.74485
          1                        0.81229                           0.08276
  Terms:
    'Intercept' (column 0)
    'C(a, Poly([1, 2, 10]))' (columns 1:3)
```

This is equivalent to R's `contr.poly`. (But note that in R, reduced rank encodings are always dummy-coded, regardless of what contrast you have set.)

class `patsy.`**`Sum`**`(`*omit=None*`)`

Deviation coding (also known as sum-to-zero coding).

Compares the mean of each level to the mean-of-means. (In a balanced design, compares the mean of each level to the overall mean.)

For full-rank coding, a standard intercept term is added.

One level must be omitted to avoid redundancy; by default this is the last level, but this can be adjusted via the *omit* argument.

> **Warning:** There are multiple definitions of 'deviation coding' in use. Make sure this is the one you expect before trying to interpret your results!

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Sum)", balanced(a=4))
Out[1]:
DesignMatrix with shape (4, 4)
  Intercept  C(a, Sum)[S.a1]  C(a, Sum)[S.a2]  C(a, Sum)[S.a3]
          1                1                0                0
          1                0                1                0
          1                0                0                1
          1               -1               -1               -1
  Terms:
    'Intercept' (column 0)
    'C(a, Sum)' (columns 1:4)

# Full rank
In [2]: dmatrix("0 + C(a, Sum)", balanced(a=4))
Out[2]:
DesignMatrix with shape (4, 4)
  C(a, Sum)[mean]  C(a, Sum)[S.a1]  C(a, Sum)[S.a2]  C(a, Sum)[S.a3]
```

```
                1                1                0                0
                1                0                1                0
                1                0                0                1
                1               -1               -1               -1
  Terms:
    ’C(a, Sum)’ (columns 0:4)

# Omit a different level
In [3]: dmatrix("C(a, Sum(1))", balanced(a=3))
Out[3]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Sum(1))[S.a1]  C(a, Sum(1))[S.a3]
          1                   1                   0
          1                  -1                  -1
          1                   0                   1
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Sum(1))’ (columns 1:3)

In [4]: dmatrix("C(a, Sum(’a1’))", balanced(a=3))
Out[4]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Sum(’a1’))[S.a2]  C(a, Sum(’a1’))[S.a3]
          1                     -1                     -1
          1                      1                      0
          1                      0                      1
  Terms:
    ’Intercept’ (column 0)
    "C(a, Sum(’a1’))" (columns 1:3)
```

This is equivalent to R’s *contr.sum*.

class `patsy.`**`Helmert`**

Helmert contrasts.

Compares the second level with the first, the third with the average of the first two, and so on.

For full-rank coding, a standard intercept term is added.

> **Warning:** There are multiple definitions of ‘Helmert coding’ in use. Make sure this is the one you expect before trying to interpret your results!

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Helmert)", balanced(a=4))
Out[1]:
DesignMatrix with shape (4, 4)
  Intercept  C(a, Helmert)[H.a2]  C(a, Helmert)[H.a3]  C(a, Helmert)[H.a4]
          1                   -1                   -1                   -1
          1                    1                   -1                   -1
          1                    0                    2                   -1
          1                    0                    0                    3
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Helmert)’ (columns 1:4)

# Full rank
In [2]: dmatrix("0 + C(a, Helmert)", balanced(a=4))
```

```
Out[2]:
DesignMatrix with shape (4, 4)
  Columns:
    ['C(a, Helmert)[H.intercept]',
     'C(a, Helmert)[H.a2]',
     'C(a, Helmert)[H.a3]',
     'C(a, Helmert)[H.a4]']
  Terms:
    'C(a, Helmert)' (columns 0:4)
  (to view full data, use np.asarray(this_obj))
```

This is equivalent to R's *contr.helmert*.

class `patsy.`**`ContrastMatrix`**(*matrix*, *column_suffixes*)

A simple container for a matrix used for coding categorical factors.

Attributes:

`matrix`

A 2d ndarray, where each column corresponds to one column of the resulting design matrix, and each row contains the entries for a single categorical variable level. Usually n-by-n for a full rank coding or n-by-(n-1) for a reduced rank coding, though other options are possible.

`column_suffixes`

A list of strings to be appended to the factor name, to produce the final column names. E.g. for treatment coding the entries will look like `"[T.level1]"`.

10.6 Spline regression

`patsy.`**`bs`**(*x*, *df=None*, *knots=None*, *degree=3*, *include_intercept=False*, *lower_bound=None*, *upper_bound=None*)

Generates a B-spline basis for `x`, allowing non-linear fits. The usual usage is something like:

```
y ~ 1 + bs(x, 4)
```

to fit `y` as a smooth function of `x`, with 4 degrees of freedom given to the smooth.

Parameters

- **df** – The number of degrees of freedom to use for this spline. The return value will have this many columns. You must specify at least one of `df` and `knots`.
- **knots** – The interior knots to use for the spline. If unspecified, then equally spaced quantiles of the input data are used. You must specify at least one of `df` and `knots`.
- **degree** – The degree of the spline to use.
- **include_intercept** – If `True`, then the resulting spline basis will span the intercept term (i.e., the constant function). If `False` (the default) then this will not be the case, which is useful for avoiding overspecification in models that include multiple spline terms and/or an intercept term.
- **lower_bound** – The lower exterior knot location.
- **upper_bound** – The upper exterior knot location.

A spline with `degree=0` is piecewise constant with breakpoints at each knot, and the default knot positions are quantiles of the input. So if you find yourself in the situation of wanting to quantize a continuous variable into equal-sized bins with a constant effect across each bin, you can use `bs(x, num_bins, degree=0)`.

Similarly, a spline with `degree=1` is piecewise linear with breakpoints at each knot.

The default is `degree=3`, which gives a cubic b-spline.

This is a stateful transform (for details see *Stateful transforms*). If `knots`, `lower_bound`, or `upper_bound` are not specified, they will be calculated from the data and then the chosen values will be remembered and re-used for prediction from the fitted model.

Using this function requires scipy be installed.

Note: This function is very similar to the R function of the same name. In cases where both return output at all (e.g., R's `bs` will raise an error if `degree=0`, while patsy's will not), they should produce identical output given identical input and parameter settings.

Warning: I'm not sure on what the proper handling of points outside the lower/upper bounds is, so for now attempting to evaluate a spline basis at such points produces an error. Patches gratefully accepted.

New in version 0.2.0.

10.7 Working with formulas programmatically

class `patsy.`**`Term`**(*factors*)

The interaction between a collection of factor objects.

This is one of the basic types used in representing formulas, and corresponds to an expression like `"a:b:c"` in a formula string. For details, see *How formulas work* and *Model specification for experts and computers*.

Terms are hashable and compare by value.

Attributes:

`factors`

A tuple of factor objects.

`patsy.`**`INTERCEPT`**

This is a pre-instantiated zero-factors `Term` object representing the intercept, useful for making your code clearer. Do remember though that this is not a singleton object, i.e., you should compare against it using `==`, not `is`.

class `patsy.`**`LookupFactor`**(*varname*, *force_categorical=False*, *contrast=None*, *levels=None*, *origin=None*)

A simple factor class that simply looks up a named entry in the given data.

Useful for programatically constructing formulas, and as a simple example of the factor protocol. For details see *Model specification for experts and computers*.

Example:

```
dmatrix(ModelDesc([], [Term([LookupFactor("x")])]), {"x": [1, 2, 3]})
```

Parameters

- **varname** – The name of this variable; used as a lookup key in the passed in data dictionary/DataFrame/whatever.
- **force_categorical** – If True, then treat this factor as categorical. (Equivalent to using `C()` in a regular formula, but of course you can't do that with a `LookupFactor`.

- **contrast** – If given, the contrast to use; see `C()`. (Requires `force_categorical=True`.)
- **levels** – If given, the categorical levels; see `C()`. (Requires `force_categorical=True`.)
- **origin** – Either `None`, or the `Origin` of this factor for use in error reporting.

New in version 0.2.0: The `force_categorical` and related arguments.

class `patsy.`**`EvalFactor`**(*code*, *eval_env*, *origin=None*)

A factor class that executes arbitrary Python code and supports stateful transforms.

Parameters

- **code** – A string containing a Python expression, that will be evaluated to produce this factor's value.
- **eval_env** – The `EvalEnvironment` where *code* will be evaluated.

This is the standard factor class that is used when parsing formula strings and implements the standard stateful transform processing. See *Stateful transforms* and *Model specification for experts and computers*.

Two EvalFactor's are considered equal (e.g., for purposes of redundancy detection) if they use the same evaluation environment and they contain the same token stream. Basically this means that the source code must be identical except for whitespace:

```
env = EvalEnvironment.capture()
assert EvalFactor("a + b", env) == EvalFactor("a+b", env)
assert EvalFactor("a + b", env) != EvalFactor("b + a", env)
```

class `patsy.`**`ModelDesc`**(*lhs_termlist*, *rhs_termlist*)

A simple container representing the termlists parsed from a formula.

This is a simple container object which has exactly the same representational power as a formula string, but is a Python object instead. You can construct one by hand, and pass it to functions like `dmatrix()` or `incr_dbuilder()` that are expecting a formula string, but without having to do any messy string manipulation. For details see *Model specification for experts and computers*.

Attributes:

`lhs_termlist`
`rhs_termlist`

Two termlists representing the left- and right-hand sides of a formula, suitable for passing to `design_matrix_builders()`.

10.8 Working with the Python execution environment

class `patsy.`**`EvalEnvironment`**(*namespaces*, *flags=0*)

Represents a Python execution environment.

Encapsulates a namespace for variable lookup and set of __future__ flags.

`add_outer_namespace`(*namespace*)

Expose the contents of a dict-like object to the encapsulated environment.

The given namespace will be checked last, after all existing namespace lookups have failed.

classmethod **`capture`**(*eval_env=0*, *reference=0*)

Capture an execution environment from the stack.

If *eval_env* is already an `EvalEnvironment`, it is returned unchanged. Otherwise, we walk up the stack by `eval_env + reference` steps and capture that function's evaluation environment.

For `eval_env=0` and `reference=0`, the default, this captures the stack frame of the function that calls `capture()`. If `eval_env + reference` is 1, then we capture that function's caller, etc.

This somewhat complicated calling convention is designed to be convenient for functions which want to capture their caller's environment by default, but also allow explicit environments to be specified. See the second example.

Example:

```
x = 1
this_env = EvalEnvironment.capture()
assert this_env["x"] == 1
def child_func():
    return EvalEnvironment.capture(1)
this_env_from_child = child_func()
assert this_env_from_child["x"] == 1
```

Example:

```
# This function can be used like:
#   my_model(formula_like, data)
#     -> evaluates formula_like in caller's environment
#   my_model(formula_like, data, eval_env=1)
#     -> evaluates formula_like in caller's caller's environment
#   my_model(formula_like, data, eval_env=my_env)
#     -> evaluates formula_like in environment 'my_env'
def my_model(formula_like, data, eval_env=0):
    eval_env = EvalEnvironment.capture(eval_env, reference=1)
    return model_setup_helper(formula_like, data, eval_env)
```

This is how `dmatrix()` works.

`eval`(*expr*, *source_name='<string>'*, *inner_namespace={}*)

Evaluate some Python code in the encapsulated environment.

Parameters

- **expr** – A string containing a Python expression.
- **source_name** – A name for this string, for use in tracebacks.
- **inner_namespace** – A dict-like object that will be checked first when *expr* attempts to access any variables.

Returns The value of *expr*.

`namespace`

A dict-like object that can be used to look up variables accessible from the encapsulated environment.

10.9 Building design matrices

`patsy.`**`design_matrix_builders`**(*termlists*, *data_iter_maker*, *NA_action='drop'*)

Construct several `DesignMatrixBuilders` from termlists.

This is one of Patsy's fundamental functions. This function and `build_design_matrices()` together form the API to the core formula interpretation machinery.

Parameters

- **termlists** – A list of termlists, where each termlist is a list of `Term` objects which together specify a design matrix.
- **data_iter_maker** – A zero-argument callable which returns an iterator over dict-like data objects. This must be a callable rather than a simple iterator because sufficiently complex formulas may require multiple passes over the data (e.g. if there are nested stateful transforms).
- **NA_action** – An `NAAction` object or string, used to determine what values count as 'missing' for purposes of determining the levels of categorical factors.

Returns A list of `DesignMatrixBuilder` objects, one for each termlist passed in.

This function performs zero or more iterations over the data in order to sniff out any necessary information about factor types, set up stateful transforms, pick column names, etc.

See *How formulas work* for details. New in version 0.2.0: The `NA_action` argument.

class `patsy.`**`DesignMatrixBuilder`**

This is an opaque class that represents Patsy's knowledge about how to build a design matrix. You get these objects from `design_matrix_builders()`, and you pass them to `build_design_matrices()`.

`design_info`

This attribute gives metadata about the matrices that this builder object can produce, in the form of a `DesignInfo` object.

`subset`(*which_terms*)

Create a new `DesignMatrixBuilder` that includes only a subset of the terms that this object does.

For example, if *builder* has terms *x*, *y*, and *z*, then:

```
builder2 = builder.subset(["x", "z"])
```

will return a new builder that will return design matrices with only the columns corresponding to the terms *x* and *z*. After we do this, then in general these two expressions will return the same thing (here we assume that *x*, *y*, and *z* each generate a single column of the output):

```
build_design_matrix([builder], data)[0][:, [0, 2]]
build_design_matrix([builder2], data)[0]
```

However, a critical difference is that in the second case, *data* need not contain any values for *y*. This is very useful when doing prediction using a subset of a model, in which situation R usually forces you to specify dummy values for *y*.

If using a formula to specify the terms to include, remember that like any formula, the intercept term will be included by default, so use *0* or *-1* in your formula if you want to avoid this.

Parameters which_terms – The terms which should be kept in the new `DesignMatrixBuilder`. If this is a string, then it is parsed as a formula, and then the names of the resulting terms are taken as the terms to keep. If it is a list, then it can contain a mixture of term names (as strings) and `Term` objects.

`patsy.`**`build_design_matrices`**(*builders*, *data*, *NA_action='drop'*, *return_type='matrix'*, *dtype=dtype('float64')*)

Construct several design matrices from `DesignMatrixBuilder` objects.

This is one of Patsy's fundamental functions. This function and `design_matrix_builders()` together form the API to the core formula interpretation machinery.

Parameters

- **builders** – A list of `DesignMatrixBuilders` specifying the design matrices to be built.

- **data** – A dict-like object which will be used to look up data.
- **NA_action** – What to do with rows that contain missing values. You can `"drop"` them, `"raise"` an error, or for customization, pass an `NAAction` object. See `NAAction` for details on what values count as 'missing' (and how to alter this).
- **return_type** – Either `"matrix"` or `"dataframe"`. See below.
- **dtype** – The dtype of the returned matrix. Useful if you want to use single-precision or extended-precision.

This function returns either a list of `DesignMatrix` objects (for `return_type="matrix"`) or a list of `pandas.DataFrame` objects (for `return_type="dataframe"`). In the latter case, the DataFrames will preserve any (row) indexes that were present in the input, which may be useful for time-series models etc. In any case, all returned design matrices will have `.design_info` attributes containing the appropriate `DesignInfo` objects.

Unlike `design_matrix_builders()`, this function takes only a simple data argument, not any kind of iterator. That's because this function doesn't need a global view of the data – everything that depends on the whole data set is already encapsulated in the *builders*. If you are incrementally processing a large data set, simply call this function for each chunk. New in version 0.2.0: The `NA_action` argument.

10.10 Missing values

class `patsy.`**`NAAction`**(*on_NA='drop', NA_types=['None', 'NaN']*)

An `NAAction` object defines a strategy for handling missing data.

"NA" is short for "Not Available", and is used to refer to any value which is somehow unmeasured or unavailable. In the long run, it is devoutly hoped that numpy will gain first-class missing value support. Until then, we work around this lack as best we're able.

There are two parts to this: First, we have to determine what counts as missing data. For numerical data, the default is to treat NaN values (e.g., `numpy.nan`) as missing. For categorical data, the default is to treat NaN values, and also the Python object None, as missing. (This is consistent with how pandas does things, so if you're already using None/NaN to mark missing data in your pandas DataFrames, you're good to go.)

Second, we have to decide what to do with any missing data when we encounter it. One option is to simply discard any rows which contain missing data from our design matrices (`drop`). Another option is to raise an error (`raise`). A third option would be to simply let the missing values pass through into the returned design matrices. However, this last option is not yet implemented, because of the lack of any standard way to represent missing values in arbitrary numpy matrices; we're hoping numpy will get this sorted out before we standardize on anything ourselves.

You can control how patsy handles missing data through the `NA_action=` argument to functions like `build_design_matrices()` and `dmatrix()`. If all you want to do is to choose between `drop` and `raise` behaviour, you can pass one of those strings as the `NA_action=` argument directly. If you want more fine-grained control over how missing values are detected and handled, then you can create an instance of this class, or your own object that implements the same interface, and pass that as the `NA_action=` argument instead.

The `NAAction` constructor takes the following arguments:

Parameters

- **on_NA** – How to handle missing values. The default is `"drop"`, which removes all rows from all matrices which contain any missing values. Also available is `"raise"`, which raises an exception when any missing values are encountered.

- **NA_types** – Which rules are used to identify missing values, as a list of strings. Allowed values are:
  - `"None"`: treat the `None` object as missing in categorical data.
  - `"NaN"`: treat floating point NaN values as missing in categorical and numerical data.

New in version 0.2.0.

`handle_NA`(*values*, *is_NAs*, *origins*)

Takes a set of factor values that may have NAs, and handles them appropriately.

Parameters

- **values** – A list of *ndarray* objects representing the data. These may be 1- or 2-dimensional, and may be of varying dtype. All will have the same number of rows (or entries, for 1-d arrays).
- **is_NAs** – A list with the same number of entries as *values*, containing boolean *ndarray* objects that indicate which rows contain NAs in the corresponding entry in *values*.
- **origins** – A list with the same number of entries as *values*, containing information on the origin of each value. If we encounter a problem with some particular value, we use the corresponding entry in *origins* as the origin argument when raising a `PatsyError`.

Returns A list of new values (which may have a differing number of rows.)

`is_categorical_NA`(*obj*)

Return True if *obj* is a categorical NA value.

Note that here *obj* is a single scalar value.

`is_numerical_NA`(*arr*)

Returns a 1-d mask array indicating which rows in an array of numerical values contain at least one NA value.

Note that here *arr* is a numpy array or pandas DataFrame.

10.11 Linear constraints

class `patsy.`**`LinearConstraint`**(*variable_names*, *coefs*, *constants=None*)

A linear constraint in matrix form.

This object represents a linear constraint of the form $Ax = b$.

Usually you won't be constructing these by hand, but instead get them as the return value from `DesignInfo.linear_constraint()`.

`coefs`

A 2-dimensional ndarray with float dtype, representing A.

`constants`

A 2-dimensional single-column ndarray with float dtype, representing b.

`variable_names`

A list of strings giving the names of the variables being constrained. (Used only for consistency checking.)

10.12 Origin tracking

class `patsy.Origin`(*code*, *start*, *end*)

This represents the origin of some object in some string.

For example, if we have an object `x1_obj` that was produced by parsing the `x1` in the formula `"y ~ x1:x2"`, then we conventionally keep track of that relationship by doing:

```
x1_obj.origin = Origin("y ~ x1:x2", 4, 6)
```

Then later if we run into a problem, we can do:

```
raise PatsyError("invalid factor", x1_obj)
```

and we'll produce a nice error message like:

```
PatsyError: invalid factor
    y ~ x1:x2
        ^^
```

Origins are compared by value, and hashable.

`caretize`(*indent=0*)

Produces a user-readable two line string indicating the origin of some code. Example:

```
y ~ x1:x2
    ^^
```

If optional argument 'indent' is given, then both lines will be indented by this much. The returned string does not have a trailing newline.

classmethod **`combine`**(*origin_objs*)

Class method for combining a set of Origins into one large Origin that spans them.

Example usage: if we wanted to represent the origin of the "x1:x2" term, we could do `Origin.combine([x1_obj, x2_obj])`.

Single argument is an iterable, and each element in the iterable should be either:

- An Origin object
- `None`
- An object that has a `.origin` attribute which fulfills the above criteria.

Returns either an Origin object, or None.

`relevant_code`()

Extracts and returns the span of the original code represented by this Origin. Example: `x1`.

CHAPTER

ELEVEN

PATSY.BUILTINS API REFERENCE

This module defines some tools that are automatically made available to code evaluated in formulas. You can also access it directly; use `from patsy.builtins import *` to import the same variables that formula code receives automatically.

`patsy.builtins.`**`I`**`(`*x*`)`

The identity function. Simply returns its input unchanged.

Since Patsy's formula parser ignores anything inside a function call syntax, this is useful to 'hide' arithmetic operations from it. For instance:

```
y ~ x1 + x2
```

has `x1` and `x2` as two separate predictors. But in:

```
y ~ I(x1 + x2)
```

we instead have a single predictor, defined to be the sum of `x1` and `x2`.

`patsy.builtins.`**`Q`**`(`*name*`)`

A way to 'quote' variable names, especially ones that do not otherwise meet Python's variable name rules.

If `x` is a variable, `Q("x")` returns the value of `x`. (Note that `Q` takes the *string* `"x"`, not the value of `x` itself.) This works even if instead of `x`, we have a variable name that would not otherwise be legal in Python.

For example, if you have a column of data named *weight.in.kg*, then you can't write:

```
y ~ weight.in.kg
```

because Python will try to find a variable named `weight`, that has an attribute named `in`, that has an attribute named `kg`. (And worse yet, `in` is a reserved word, which makes this example doubly broken.) Instead, write:

```
y ~ Q("weight.in.kg")
```

and all will be well. Note, though, that this requires embedding a Python string inside your formula, which may require some care with your quote marks. Some standard options include:

```
my_fit_function("y ~ Q('weight.in.kg')", ...)
my_fit_function('y ~ Q("weight.in.kg")', ...)
my_fit_function("y ~ Q(\"weight.in.kg\")", ...)
```

Note also that `Q` is an ordinary Python function, which means that you can use it in more complex expressions. For example, this is a legal formula:

```
y ~ np.sqrt(Q("weight.in.kg"))
```

class `patsy.builtins.`**`ContrastMatrix`**(*matrix*, *column_suffixes*)

A simple container for a matrix used for coding categorical factors.

Attributes:

`matrix`

A 2d ndarray, where each column corresponds to one column of the resulting design matrix, and each row contains the entries for a single categorical variable level. Usually n-by-n for a full rank coding or n-by-(n-1) for a reduced rank coding, though other options are possible.

`column_suffixes`

A list of strings to be appended to the factor name, to produce the final column names. E.g. for treatment coding the entries will look like `"[T.level1]"`.

class `patsy.builtins.`**`Treatment`**(*reference=None*)

Treatment coding (also known as dummy coding).

This is the default coding.

For reduced-rank coding, one level is chosen as the “reference”, and its mean behaviour is represented by the intercept. Each column of the resulting matrix represents the difference between the mean of one level and this reference level.

For full-rank coding, classic “dummy” coding is used, and each column of the resulting matrix represents the mean of the corresponding level.

The reference level defaults to the first level, or can be specified explicitly.

```
# reduced rank
In [1]: dmatrix("C(a, Treatment)", balanced(a=3))
Out[1]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment)[T.a2]  C(a, Treatment)[T.a3]
          1                      0                      0
          1                      1                      0
          1                      0                      1
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Treatment)’ (columns 1:3)

# full rank
In [2]: dmatrix("0 + C(a, Treatment)", balanced(a=3))
Out[2]:
DesignMatrix with shape (3, 3)
  C(a, Treatment)[a1]  C(a, Treatment)[a2]  C(a, Treatment)[a3]
                    1                    0                    0
                    0                    1                    0
                    0                    0                    1
  Terms:
    ’C(a, Treatment)’ (columns 0:3)

# Setting a reference level
In [3]: dmatrix("C(a, Treatment(1))", balanced(a=3))
Out[3]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment(1))[T.a1]  C(a, Treatment(1))[T.a3]
          1                         1                         0
          1                         0                         0
          1                         0                         1
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Treatment(1))’ (columns 1:3)

In [4]: dmatrix("C(a, Treatment(’a2’))", balanced(a=3))
Out[4]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Treatment(’a2’))[T.a1]  C(a, Treatment(’a2’))[T.a3]
          1                            1                            0
          1                            0                            0
          1                            0                            1
  Terms:
    ’Intercept’ (column 0)
    "C(a, Treatment(’a2’))" (columns 1:3)
```

Equivalent to R `contr.treatment`. The R documentation suggests that using `Treatment(reference=-1)` will produce contrasts that are “equivalent to those produced by many (but not all) SAS procedures”.

`code_with_intercept`(*levels*)

`code_without_intercept`(*levels*)

class `patsy.builtins.`**`Poly`**(*scores=None*)

Orthogonal polynomial contrast coding.

This coding scheme treats the levels as ordered samples from an underlying continuous scale, whose effect takes an unknown functional form which is Taylor-decomposed into the sum of a linear, quadratic, etc. components.

For reduced-rank coding, you get a linear column, a quadratic column, etc., up to the number of levels provided.

For full-rank coding, the same scheme is used, except that the zero-order constant polynomial is also included. I.e., you get an intercept column included as part of your categorical term.

By default the levels are treated as equally spaced, but you can override this by providing a value for the *scores* argument.

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Poly)", balanced(a=4))
Out[1]:
DesignMatrix with shape (4, 4)
  Intercept  C(a, Poly).Linear  C(a, Poly).Quadratic  C(a, Poly).Cubic
          1          -0.67082                   0.5          -0.22361
          1          -0.22361                  -0.5           0.67082
          1           0.22361                  -0.5          -0.67082
          1           0.67082                   0.5           0.22361
  Terms:
    ’Intercept’ (column 0)
    ’C(a, Poly)’ (columns 1:4)

# Full rank
In [2]: dmatrix("0 + C(a, Poly)", balanced(a=3))
Out[2]:
DesignMatrix with shape (3, 3)
  C(a, Poly).Constant  C(a, Poly).Linear  C(a, Poly).Quadratic
                    1          -0.70711               0.40825
                    1          -0.00000              -0.81650
                    1           0.70711               0.40825
  Terms:
    ’C(a, Poly)’ (columns 0:3)
```

```
# Explicit scores
In [3]: dmatrix("C(a, Poly([1, 2, 10]))", balanced(a=3))
Out[3]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Poly([1, 2, 10])).Linear  C(a, Poly([1, 2, 10])).Quadratic
          1                       -0.47782                          0.66208
          1                       -0.33447                         -0.74485
          1                        0.81229                          0.08276
  Terms:
    'Intercept' (column 0)
    'C(a, Poly([1, 2, 10]))' (columns 1:3)
```

This is equivalent to R's `contr.poly`. (But note that in R, reduced rank encodings are always dummy-coded, regardless of what contrast you have set.)

code_with_intercept(*levels*)

code_without_intercept(*levels*)

class patsy.builtins.**Sum**(*omit=None*)

Deviation coding (also known as sum-to-zero coding).

Compares the mean of each level to the mean-of-means. (In a balanced design, compares the mean of each level to the overall mean.)

For full-rank coding, a standard intercept term is added.

One level must be omitted to avoid redundancy; by default this is the last level, but this can be adjusted via the *omit* argument.

> **Warning:** There are multiple definitions of 'deviation coding' in use. Make sure this is the one you expect before trying to interpret your results!

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Sum)", balanced(a=4))
Out[1]:
DesignMatrix with shape (4, 4)
  Intercept  C(a, Sum)[S.a1]  C(a, Sum)[S.a2]  C(a, Sum)[S.a3]
          1                1                0                0
          1                0                1                0
          1                0                0                1
          1               -1               -1               -1
  Terms:
    'Intercept' (column 0)
    'C(a, Sum)' (columns 1:4)

# Full rank
In [2]: dmatrix("0 + C(a, Sum)", balanced(a=4))
Out[2]:
DesignMatrix with shape (4, 4)
  C(a, Sum)[mean]  C(a, Sum)[S.a1]  C(a, Sum)[S.a2]  C(a, Sum)[S.a3]
                1                1                0                0
                1                0                1                0
                1                0                0                1
                1               -1               -1               -1
  Terms:
    'C(a, Sum)' (columns 0:4)
```

```
# Omit a different level
In [3]: dmatrix("C(a, Sum(1))", balanced(a=3))
Out[3]: 
DesignMatrix with shape (3, 3)
  Intercept  C(a, Sum(1))[S.a1]  C(a, Sum(1))[S.a3]
          1                   1                   0
          1                  -1                  -1
          1                   0                   1
  Terms:
    'Intercept' (column 0)
    'C(a, Sum(1))' (columns 1:3)

In [4]: dmatrix("C(a, Sum('a1'))", balanced(a=3))
Out[4]: 
DesignMatrix with shape (3, 3)
  Intercept  C(a, Sum('a1'))[S.a2]  C(a, Sum('a1'))[S.a3]
          1                     -1                     -1
          1                      1                      0
          1                      0                      1
  Terms:
    'Intercept' (column 0)
    "C(a, Sum('a1'))" (columns 1:3)
```

This is equivalent to R's *contr.sum*.

code_with_intercept (*levels*)

code_without_intercept (*levels*)

class patsy.builtins.**Helmert**

Helmert contrasts.

Compares the second level with the first, the third with the average of the first two, and so on.

For full-rank coding, a standard intercept term is added.

Warning: There are multiple definitions of 'Helmert coding' in use. Make sure this is the one you expect before trying to interpret your results!

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Helmert)", balanced(a=4))
Out[1]: 
DesignMatrix with shape (4, 4)
  Intercept  C(a, Helmert)[H.a2]  C(a, Helmert)[H.a3]  C(a, Helmert)[H.a4]
          1                   -1                   -1                   -1
          1                    1                   -1                   -1
          1                    0                    2                   -1
          1                    0                    0                    3
  Terms:
    'Intercept' (column 0)
    'C(a, Helmert)' (columns 1:4)

# Full rank
In [2]: dmatrix("0 + C(a, Helmert)", balanced(a=4))
Out[2]: 
DesignMatrix with shape (4, 4)
  Columns:
    ['C(a, Helmert)[H.intercept]',
     'C(a, Helmert)[H.a2]',
     'C(a, Helmert)[H.a3]',
     'C(a, Helmert)[H.a4]']
  Terms:
    'C(a, Helmert)' (columns 0:4)
  (to view full data, use np.asarray(this_obj))
```

This is equivalent to R's *contr.helmert*.

code_with_intercept(*levels*)

code_without_intercept(*levels*)

class patsy.builtins.**Diff**

Backward difference coding.

This coding scheme is useful for ordered factors, and compares the mean of each level with the preceding level. So you get the second level minus the first, the third level minus the second, etc.

For full-rank coding, a standard intercept term is added (which gives the mean value for the first level).

Examples:

```
# Reduced rank
In [1]: dmatrix("C(a, Diff)", balanced(a=3))
Out[1]:
DesignMatrix with shape (3, 3)
  Intercept  C(a, Diff)[D.a1]  C(a, Diff)[D.a2]
          1          -0.66667          -0.33333
          1           0.33333          -0.33333
          1           0.33333           0.66667
  Terms:
    'Intercept' (column 0)
    'C(a, Diff)' (columns 1:3)

# Full rank
In [2]: dmatrix("0 + C(a, Diff)", balanced(a=3))
Out[2]:
DesignMatrix with shape (3, 3)
  C(a, Diff)[D.a1]  C(a, Diff)[D.a2]  C(a, Diff)[D.a3]
                 1          -0.66667          -0.33333
                 1           0.33333          -0.33333
                 1           0.33333           0.66667
  Terms:
    'C(a, Diff)' (columns 0:3)
```

code_with_intercept(*levels*)

code_without_intercept(*levels*)

patsy.builtins.**C**(*data*, *contrast=None*, *levels=None*)

Marks some *data* as being categorical, and specifies how to interpret it.

This is used for three reasons:

- To explicitly mark some data as categorical. For instance, integer data is by default treated as numerical. If you have data that is stored using an integer type, but where you want patsy to treat each different value as a different level of a categorical factor, you can wrap it in a call to *C* to accomplish this. E.g., compare:

  ```
  dmatrix("a", {"a": [1, 2, 3]})
  dmatrix("C(a)", {"a": [1, 2, 3]})
  ```

- To explicitly set the levels or override the default level ordering for categorical data, e.g.:

  ```
  dmatrix("C(a, levels=["a2", "a1"])", balanced(a=2))
  ```

- To override the default coding scheme for categorical data. The *contrast* argument can be any of:
  - A `ContrastMatrix` object
  - A simple 2d ndarray (which is treated the same as a ContrastMatrix object except that you can't specify column names)
  - An object with methods called *code_with_intercept* and *code_without_intercept*, like the built-in contrasts (`Treatment`, `Diff`, `Poly`, etc.). See *Coding categorical data* for more details.
  - A callable that returns one of the above.

`patsy.builtins.`**`center`**`(`*x*`)`

A stateful transform that centers input data, i.e., subtracts the mean.

If input has multiple columns, centers each column separately.

Equivalent to `standardize(x, rescale=False)`

`patsy.builtins.`**`standardize`**`(`*x*, *center=True*, *rescale=True*, *ddof=0*`)`

A stateful transform that standardizes input data, i.e. it subtracts the mean and divides by the sample standard deviation.

Either centering or rescaling or both can be disabled by use of keyword arguments. The *ddof* argument controls the delta degrees of freedom when computing the standard deviation (cf. `numpy.std()`). The default of `ddof=0` produces the maximum likelihood estimate; use `ddof=1` if you prefer the square root of the unbiased estimate of the variance.

If input has multiple columns, standardizes each column separately.

Note: This function computes the mean and standard deviation using a memory-efficient online algorithm, making it suitable for use with large incrementally processed data-sets.

`patsy.builtins.`**`scale`**`(`**args*, ***kwargs*`)`

standardize(x, center=True, rescale=True, ddof=0)

A stateful transform that standardizes input data, i.e. it subtracts the mean and divides by the sample standard deviation.

Either centering or rescaling or both can be disabled by use of keyword arguments. The *ddof* argument controls the delta degrees of freedom when computing the standard deviation (cf. `numpy.std()`). The default of `ddof=0` produces the maximum likelihood estimate; use `ddof=1` if you prefer the square root of the unbiased estimate of the variance.

If input has multiple columns, standardizes each column separately.

Note: This function computes the mean and standard deviation using a memory-efficient online algorithm, making it suitable for use with large incrementally processed data-sets.

`patsy.builtins.`**`bs`**`(`*x*, *df=None*, *knots=None*, *degree=3*, *include_intercept=False*, *lower_bound=None*, *upper_bound=None*`)`

Generates a B-spline basis for `x`, allowing non-linear fits. The usual usage is something like:

```
y ~ 1 + bs(x, 4)
```

to fit `y` as a smooth function of `x`, with 4 degrees of freedom given to the smooth.

Parameters

- **df** – The number of degrees of freedom to use for this spline. The return value will have this many columns. You must specify at least one of `df` and `knots`.
- **knots** – The interior knots to use for the spline. If unspecified, then equally spaced quantiles of the input data are used. You must specify at least one of `df` and `knots`.
- **degree** – The degree of the spline to use.
- **include_intercept** – If `True`, then the resulting spline basis will span the intercept term (i.e., the constant function). If `False` (the default) then this will not be the case, which is useful for avoiding overspecification in models that include multiple spline terms and/or an intercept term.
- **lower_bound** – The lower exterior knot location.
- **upper_bound** – The upper exterior knot location.

A spline with `degree=0` is piecewise constant with breakpoints at each knot, and the default knot positions are quantiles of the input. So if you find yourself in the situation of wanting to quantize a continuous variable into equal-sized bins with a constant effect across each bin, you can use `bs(x, num_bins, degree=0)`.

Similarly, a spline with `degree=1` is piecewise linear with breakpoints at each knot.

The default is `degree=3`, which gives a cubic b-spline.

This is a stateful transform (for details see *Stateful transforms*). If `knots`, `lower_bound`, or `upper_bound` are not specified, they will be calculated from the data and then the chosen values will be remembered and re-used for prediction from the fitted model.

Using this function requires scipy be installed.

Note: This function is very similar to the R function of the same name. In cases where both return output at all (e.g., R's `bs` will raise an error if `degree=0`, while patsy's will not), they should produce identical output given identical input and parameter settings.

Warning: I'm not sure on what the proper handling of points outside the lower/upper bounds is, so for now attempting to evaluate a spline basis at such points produces an error. Patches gratefully accepted.

New in version 0.2.0.

CHAPTER

TWELVE

CHANGES

12.1 v0.2.0

Warnings:

- The lowest officially supported Python version is now 2.5. So far as I know everything still works with Python 2.4, but as everyone else has continued to drop support for 2.4, testing on 2.4 has become so much trouble that I've given up.

New features:

- New support for automatically detecting and (optionally) removing missing values (see `NAAction`).
- New stateful transform for B-spline regression: `bs()`. (Requires scipy.)
- Added a core API to make it possible to run predictions on only a subset of model terms. (This is particularly useful for e.g. plotting the isolated effect of a single fitted spline term.) See `DesignMatrixBuilder.subset()`.
- `LookupFactor` now allows users to mark variables as categorical directly.
- `pandas.Categorical` objects are now recognized as representing categorical data and handled appropriately.
- Better error reporting for exceptions raised by user code inside formulas. We now, whenever possible, tag the generated exception with information about which factor's code raised it, and use this information to give better error reporting.
- `EvalEnvironment.capture()` now takes a *reference* argument, to make it easier to implement new `dmatrix()`-like functions.

Other: miscellaneous doc improvements and bug fixes.

12.2 v0.1.0

First public release.

CHAPTER

THIRTEEN

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX