boxes Documentation

Release 0.1.0

Anders Jellinggaard

November 11, 2016

1 Introduction to boxes **3**

2 Reference to boxes **5**
2.1 boxes . . . 5
2.2 boxes.box . . . 5
2.3 boxes.cartesian . . . 7
2.4 boxes.constrain . . . 8
2.5 boxes.display . . . 9
2.6 boxes.context . . . 10

3 Introduction to symmath **11**

4 Reference to symmath **13**
4.1 symmath . . . 13
4.2 symmath.expr . . . 13
4.3 symmath.system . . . 14

5 Indices and tables **17**

Python Module Index **19**

The purpose of the boxes library is to align rectangles in 2D based on fairly general *constraints*. For instance, it can be used to make sub-plots in a figure line up nicely, with easy control over margins and dimensions.

The illustration above shows an example of this kind:

- The figure is 6 units wide
- It contains a square and a rectangle.
- The rectangle has an aspect ratio of 2:1.
- All spacings are 0.3 units.

How high should the figure be? — The answer is (6.0 0.3 × 3) / 3 + 0.3 × 2 = 2.3. Clearly, as figures become more complicated, solving these kinds of problems by hand quickly gets complicated. This is where this library comes in.

Below is code solving the above example:

```
from boxes import Context, constrain

ctx = Context()

# A box representing the figure. We only define its width.
fig = ctx.box(width=6)

# Create a square of any size, and a rectangle twice as wide
square = ctx.box(aspect=1)
rectangle = ctx.box(width=2 * square.width)

# Align the square and rectangle next to each other.
row = constrain.row(square, rectangle, spacing=0.3)

# Put them in the figure with 0.3 units of margins.
fig.pad(0.3).fix(row)

fig.solve()
print("The height of the figure is", fig.height)
```

```
The height of the figure is 2.3
```

This library supports any constraint which can ulitmately be expressed as a set of linear equations. See the boxes.constrain module for some examples.

CHAPTER 1

Introduction to boxes

The central class in this library is the *Box* class. To get hold of one, first construct a *Context*.

```
>>> from boxes import Context
>>> ctx = Context()
>>> box1 = ctx.box()
```

A *Box* has two main properties, a location and a size.

```
>>> print(box1.loc)
(x3, x0)
>>> print(box1.size)
(x1 - x3, -x0 + x2)
```

Note that these are symbolic expressions. Constraints in the *boxes* library are handled internally by adding equations to the context. Let us try creating a constraint, by creating second box and putting the two boxes next to each other with some spacing between them using *boxes.constrain.row()*.

```
>>> from boxes import constrain
>>> box2 = ctx.box()
>>> fig = constrain.row(box1, box2, spacing=0.3)
```

We have now added some equations to the context.

```
>>> len(ctx.system.facts) > 0
True
```

But we have not added enough to completely define the layout, therefore calling *solve()* on *fig* (which is the bounding box of *box1* and *box2*) raises an exception.

```
>>> fig.solve()
Traceback (most recent call last):
  ...
AssertionError
```

Lets add a few more constraints until the system is completely defined. First, let us set the total size.

```
>>> constrain.size((6, 2), fig)
```

And now let us set *box1* to be twice as wide as *box2* is tall.

```
>>> ctx.equate(box1.width, 2 * box2.height)
```

That should do it, now we have enough equations.

```
>>> fig.solve()
```

After calling `fig.solve()`, properties that were symbolic before are now regular floats:

```
>>> print(box1.loc)
(0.0, 0.0)
>>> print(box1.size)
(4.0, 2.0)
```

Let us have a look at the output

```
>>> from boxes.display import display
>>> display("figures/intro.svg", fig, (box1, box2))
```

CHAPTER 2

Reference to boxes

Modules in the library:

- boxes — convenience reexports.
- boxes.box — contains the *Box* class and the `context()` function.
- boxes.cartesian — contains the *Vect* and *Rect* classes.
- boxes.constrain — a collection of constraints.
- boxes.display — render some boxes to an image file.
- boxes.context — contains the *Context* class which holds the equations constraining a set of boxes.

2.1 boxes

Convenience reexports. Typing `from boxes import *` imports the following:

- The class `Context` which is the starting point of the library.
- The module *constrain* which contains a bunch of convenient constraints.

2.2 boxes.box

class `boxes.box.`**Box**(*context*, *rect=None*, *aspect=None*, ***kwargs*)

Represents a rectangle in the latout. The position of the rectangle is given by the *rect* property.

All keyword arguments to the constructor are optional, and the same effect can be obtained by later constraining the box.

Parameters

- **context** (boxes.context.Context) – The context the box belongs to.
- **rect** (boxes.cartesian.Rect) – Explicitly locate the box.
- **aspect** (*float*) – Forwarded to *boxes.constrain.aspect()*.

In addition, any property of *Rect* can be used as a keyword argument to the constructor, which will introduce a constraint.

Note, that *Box* instances can also be obtained by calling the *box()* method of a *Context* instance.

Methods

solve(*fix_upper_left=True*)

Calling this method has two effects:

1. If *fix_upper_left* is True (the default), a constraint is added setting `self.loc == (0, 0)`.
2. *`solve()`* is called on *`Box.context`*.

```
>>> from boxes import Context
>>> ctx = Context()
>>> box = ctx.box(size=(6.0, 10.0))
>>> (box.width, box.height)
(Expr(x1 - x3), Expr(-x0 + x2))
>>> box.solve()
>>> (box.width, box.height)
(6.0, 10.0)
```

fix(*other*)

Constrain this box and *other* to have the same size and position.

pad(**args*)

Construct a new box similar to this, but smaller by the given amount.

The method can be called in four different ways (similar to how margins and padding is specified in *css* files):

- `box.pad(`*`all`*`)`
- `box.pad(`*`vert, hor`*`)`
- `box.pad(`*`top, hor, bottom`*`)`
- `box.pad(`*`top, right, bottom, left`*`)`

Simple example

```
>>> from boxes import *
>>> ctx = Context()
>>> outer_box = ctx.box(width=10)
>>> inner_box = ctx.box(height=4)
>>> outer_box.pad(0.5).fix(inner_box)
>>> outer_box.solve()
>>> print(outer_box.loc, outer_box.size)
(0.0, 0.0) (10.0, 5.0)
>>> print(inner_box.loc, inner_box.size)
(0.5, 0.5) (9.0, 4.0)
```

Center a box inside another

```
>>> from boxes import *
>>> ctx = Context()
>>> outer_box = ctx.box(size=(6, 5))
>>> inner_box = ctx.box(size=(4, 4))
>>> outer_box.pad(ctx.sym(), ctx.sym()).fix(inner_box)
>>> outer_box.solve()
>>> print(outer_box.loc, outer_box.size)
(0.0, 0.0) (6.0, 5.0)
```

```
>>> print(inner_box.loc, inner_box.size)
(1.0, 0.5) (4.0, 4.0)
```

surround(**args*)
Similar to *pad()*, but the arguments are negated.

Properties

rect
The position of the box as a *boxes.cartesian.Rect* instance. The dimensions of *rect* are expressions of type *symmath.expr.Expr* until you *solve()* the context.

width, height, top, right, bottom, left, loc, and size
Any property of *rect* is also made a property of this class.

context
The *boxes.context.Context* of this box.

2.3 boxes.cartesian

Vect	Class of two-dimensional vectors.
Rect(top, right, bottom, left)	Holds the dimensions of a rectangle.

class boxes.cartesian.**Vect**(*x*, *y*)
Class of two-dimensional vectors.

This a named tuple (see collections.namedtuple() from the python standard library) with an x and a y attribute. Implements __add__() and __sub__().

class boxes.cartesian.**Rect**(*top*, *right*, *bottom*, *left*)
Holds the dimensions of a rectangle.

Objects of this class cannot be modified after creation

```
>>> from boxes.cartesian import Rect
>>> r = Rect(0, 1, 1, 0)
>>> r.right = 2
Traceback (most recent call last):
  ...
TypeError: 'Rect' objects are immutable
```

top
The *y*-coordinate of the top edge of the rectangle.

right
The *x*-coordinate of the right edge of the rectangle.

bottom
The *y*-coordinate of the bottom edge of the rectangle.

left
The *x*-coordinate of the left edge of the rectangle.

width
The width of the rectangle.

height
The height of the rectangle.

loc
The upper left corner of the rectangle (as a *Vect*).

size
The size of the rectangle (as a *Vect*).

2.4 boxes.constrain

align(edges, *boxes)	Aligns boxes along the specified edges.
row(*boxes[, spacing])	Place the given boxes along a horizontal row with the given *spacing* between them.
column(*boxes[, spacing])	Place the given boxes along a vertical column with the given *spacing* between them.
hcat(*boxes[, spacing])	Similar to *column()* but does not constrain top and bottom edges in any way.
vcat(*boxes[, spacing])	Similar to *row()* but does not constrain left and right edges in any way.
aspect(aspect, *boxes)	Constrain every box in *boxes* to have the given aspect ratio.
width(width, *boxes)	Constrain the width of every box in *boxes*
height(height, *boxes)	Constrain the height of every box in *boxes*
size(size, *boxes)	Constrain the size of every box in *boxes*

boxes.constrain.**align**(*edges*, **boxes*)
Aligns boxes along the specified edges.

This function adds contraints aligning the given boxes along edges specified in the following manner:

- If `'t'` is in *edges*, equate the *y*-coordinates of the top edges of every box in *boxes*.
- If `'r'` is in *edges*, equate the *x*-coordinates of the right edges of every box in *boxes*.
- If `'b'` is in *edges*, equate the *y*-coordinates of the bottom edges of every box in *boxes*.
- If `'l'` is in *edges*, equate the *x*-coordinates of the left edges of every box in *boxes*.

Parameters

- **edges** (*str*) – Any combination of the letters `'t'`, `'r'`, `'b'`, and `'l'`.
- **boxes** – The *Box* objects to align.

boxes.constrain.**row**(**boxes*, *spacing=0*)
Place the given boxes along a horizontal row with the given *spacing* between them. The boxes will align along the top and bottom edges.

Parameters

- **boxes** – The *Box* objects to put in a row.
- **spacing** – The spacing between the boxes (this can be a `float` or an *Expr*).

boxes.constrain.**column**(**boxes*, *spacing=0*)
Place the given boxes along a vertical column with the given *spacing* between them. The boxes will align along the left and right edges.

Parameters

- **boxes** – The *Box* objects to put in a column.

- **spacing** (a `float` or an *Expr*) – The spacing between the boxes.

`boxes.constrain.`**`hcat`**(**boxes*, *spacing=0*)
Similar to *column()* but does not constrain top and bottom edges in any way.

Parameters

- **boxes** – *Box* objects.
- **spacing** (a `float` or an *Expr*) – The spacing between the boxes.

`boxes.constrain.`**`vcat`**(**boxes*, *spacing=0*)
Similar to *row()* but does not constrain left and right edges in any way.

Parameters

- **boxes** – *Box* objects.
- **spacing** (a `float` or an *Expr*) – The spacing between the boxes.

`boxes.constrain.`**`aspect`**(*aspect*, **boxes*)
Constrain every box in *boxes* to have the given aspect ratio.

Parameters

- **aspect** (*float*) – The aspect ratio is defined as the width divided by the height.
- **boxes** – *Box* objects.

`boxes.constrain.`**`width`**(*width*, **boxes*)
Constrain the width of every box in *boxes*

Parameters

- **width** – A `float` or an *Expr* object.
- **boxes** – *Box* objects.

`boxes.constrain.`**`height`**(*height*, **boxes*)
Constrain the height of every box in *boxes*

Parameters

- **height** – A `float` or an *Expr* object.
- **boxes** – *Box* objects.

`boxes.constrain.`**`size`**(*size*, **boxes*)
Constrain the size of every box in *boxes*

Parameters

- **size** – A `float` or an *Expr* object.
- **boxes** – *Box* objects.

2.5 boxes.display

`boxes.display.`**`display`**(*filename*, *figure*, *boxes*, *dots_per_unit=30*)
Output a figure showing the location of the given *boxes* to *filename*. This function calls `figure.solve()` and uses the size of `figure` as the size of the image. This function is meant for quick visualization.

Parameters

- **filename** (*string*) – The output file. Must end in `.svg` or `.png`.

- **figure** (*Box*) – Defines the size of the figure.
- **boxes** (An iterable of *Box* instances) – The boxes to draw.
- **dots_per_unit** (*float*) – Scaling factor.

2.6 boxes.context

class boxes.context.**Context**

system
The underlying *System* holding all equations constraining the layout.

is_solved
A bool indicating whether *solve()* has been called.

box(**args*, ***kwargs*)
Construct a *Box* using this context.

equate(*x*, *y*)
Add a constraint setting *x* == *y*.

solve()
Solve the layout. This function raises an error if the layout is not fully defined.

sym()
Create an expression (of type *symmath.expr.Expr*) representing a fresh symbol unused in this context.

CHAPTER 3

Introduction to symmath

Expressions of type *Expr* emulate numeric types to some extent. Specifically, only linear combinations of symbols are supported.

```
>>> from symmath import Expr, sym
>>> a = sym('a')
>>> b = sym('b')
>>> a + b
Expr(a + b)
>>> 2 * (3 * a + b + 2)
Expr(6 a + 2 b + 4)
>>> a * b
Traceback (most recent call last):
  ...
SymmathError: Symmath expressions cannot be multiplied together
```

To convert an expression to a regular number, use *symmath.expr.Expr.scalar()*, which throws an exception if an expression contains symbols (which obviously cannot be converted).

```
>>> x = a + 3
>>> (x - a).scalar()
3
>>> (x - b).scalar()
Traceback (most recent call last):
  ...
SymmathError: Not a scalar
```

Expressions can be copied and modified in place.

```
>>> x = Expr(a)
>>> x is not a
True
>>> x += 2 * b
>>> x
Expr(a + 2 b)
>>> a
Expr(a)
```

The coeffient in front of each term can be accessed and modified using item accessors.

```
>>> x = 1.5 * a + b
>>> x['a']
1.5
>>> x['a'] += 3
>>> x
```

```
Expr(4.5 a + b)
>>> del x['a']
>>> x['c'] = 2
>>> x
Expr(b + 2 c)
>>> x['d']
0
```

As a special case, *None* can be used to access the scalar part of an expression.

```
>>> x = a + b + 3
>>> x[None]
3
>>> x[None] = 4
>>> x
Expr(a + b + 4)
```

Any object than can be used as key in a dictionary can be used to name a symbol (note, integers get an 'x' prefixed when printing).

```
>>> x = sym(('x', 'y')) + sym(3) + sym(1) + 3
>>> x
Expr(('x', 'y') + x1 + x3 + 3)
```

CHAPTER 4

Reference to symmath

Modules in the library:

- symmath — convenience reexports.
- symmath.expr — contains the *Expr* class and the *sym()* function.
- symmath.system — contains the *System*.

4.1 symmath

Convenience reexports. Typing `from symmath import *` imports everything from *symmath.expr* and *symmath.system*.

4.2 symmath.expr

exception `symmath.expr.`**SymmathError**
Bases: `ValueError`

class `symmath.expr.`**Expr**(*other=None*, *tolerance=1e-08*)
Class used for symbolic computations. Introduction to symmath describes what operations this class supports.

terms
A mapping from symbol names to their coefficients.

tolerance
Numbers smaller than the *tolerance* in absolute value are converted to 0.

scalar()
If *x* contains no symbols, `x.scalar()` returns a regular float equal to the scalar part of *x*. Otherwise an exception is raised.

```
>>> a = sym('a')
>>> x = a + 3
>>> x.scalar()
Traceback (most recent call last):
  ...
SymmathError: Not a scalar
>>> x -= a
>>> x
Expr(3)
```

```
>>> x.scalar()
3
```

substitue(*symbol*, *arg*)
Replace all occurances of *symbol* with *arg*.

```
>>> from symmath import sym
>>> x = sym("x")
>>> y = sym("y")
>>> z = sym("z")
>>> e = 2 * x + y + 1
>>> e.substitue("x", y + z)
>>> print(e)
3 y + 2 z + 1
```

symmath.expr.**sym**(*symbol*)
Create an expression containing just the given symbol.

```
>>> sym('a')
Expr(a)
```

4.3 symmath.system

class symmath.system.**System**

facts
A dictionary mapping symbol names to equivalent expressions. Each value in this dictionary should not contain any references to the symbols used as keys. Consider this attribute read-only, and use *equate()* to add a fact instead of modifying this dictionary.

equate(*a*, *b*)
Eliminate one symbol from the system by using the equation: $a = b$. This adds a fact to *facts* and simplifies all existing facts.

An error is raised if this equation over-constrains the system.

```
>>> x = sym('x')
>>> y = sym('y')
>>> system = System()
>>> system.equate(x + 1, 2*y)
>>> system.equate(2*x, y + 2)
>>> system.equate(x, y)
Traceback (most recent call last):
  ...
SymmathError: System is over-constrained
```

Classes can define _symmath_equate() to customize what it means objects of that class to be equal to something else.

```
>>> class Vector2D:
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...
...   def _symmath_equate(self, f, other):
...     f(self.x, other.x)
```

```
        ...     f(self.y, other.y)

        >>> x = sym('x')
        >>> y = sym('y')
        >>> a = Vector2D(x + 5, 2*y)
        >>> b = Vector2D(4*y, 3*x)

        >>> system = System()
        >>> system.equate(a, b)
        >>> system.eval(x)
        1.0
        >>> system.eval(y)
        1.5
```

eval(*val*)
Substitue all known facts into *val*, recursing into structures that support it. *eval()* calls *symmath.expr.Expr.scalar()* to eliminate all expressions. This method raises an exception if the system is not fully solved.

Classes supporting this function define the method _symmath_eval(), which takes a single argument: the *eval()* method as a bound method, and should return a fully evaluated copy of itself.

rewrite(*expr*)
Substitue all known *facts* into *expr*.

simplify(*expr*)
Same as *rewrite()*, but returns the simplified expression instead of modifying *expr*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

b

boxes, 5
boxes.box, 5
boxes.cartesian, 7
boxes.constrain, 8
boxes.context, 10
boxes.display, 9

s

symmath.__init__, 13
symmath.expr, 13
symmath.system, 14

A

align() (in module boxes.constrain), 8
aspect() (in module boxes.constrain), 9

B

bottom (boxes.cartesian.Rect attribute), 7
Box (class in boxes.box), 5
box() (boxes.context.Context method), 10
boxes (module), 5
boxes.box (module), 5
boxes.cartesian (module), 7
boxes.constrain (module), 8
boxes.context (module), 10
boxes.display (module), 9

C

column() (in module boxes.constrain), 8
context (boxes.box.Box attribute), 7
Context (class in boxes.context), 10

D

display() (in module boxes.display), 9

E

equate() (boxes.context.Context method), 10
equate() (symmath.system.System method), 14
eval() (symmath.system.System method), 15
Expr (class in symmath.expr), 13

F

facts (symmath.system.System attribute), 14
fix() (boxes.box.Box method), 6

H

hcat() (in module boxes.constrain), 9
height (boxes.cartesian.Rect attribute), 7
height() (in module boxes.constrain), 9

I

is_solved (boxes.context.Context attribute), 10

L

left (boxes.cartesian.Rect attribute), 7
loc (boxes.cartesian.Rect attribute), 8

P

pad() (boxes.box.Box method), 6

R

rect (boxes.box.Box attribute), 7
Rect (class in boxes.cartesian), 7
rewrite() (symmath.system.System method), 15
right (boxes.cartesian.Rect attribute), 7
row() (in module boxes.constrain), 8

S

scalar() (symmath.expr.Expr method), 13
simplify() (symmath.system.System method), 15
size (boxes.cartesian.Rect attribute), 8
size() (in module boxes.constrain), 9
solve() (boxes.box.Box method), 6
solve() (boxes.context.Context method), 10
substitue() (symmath.expr.Expr method), 14
surround() (boxes.box.Box method), 7
sym() (boxes.context.Context method), 10
sym() (in module symmath.expr), 14
symmath.__init__ (module), 13
symmath.expr (module), 13
symmath.system (module), 14
SymmathError, 13
system (boxes.context.Context attribute), 10
System (class in symmath.system), 14

T

terms (symmath.expr.Expr attribute), 13
tolerance (symmath.expr.Expr attribute), 13
top (boxes.cartesian.Rect attribute), 7

V

vcat() (in module boxes.constrain), 9
Vect (class in boxes.cartesian), 7

W

width (boxes.cartesian.Rect attribute), 7
width() (in module boxes.constrain), 9