builders

Release 1.2.8

March 16, 2015

`builders` is a small and lightweight framework created to facilitate creation of complex test data for systems with complicated and intermingled data model.

See *Basic tutorial* for a jump-start.

Contents:

Frequently Asked Questions

This is a list of Frequently Asked Questions about builders. Feel free to suggest new entries!

1.1 How do I...

... set backrefs? Use `builders.construct.Uplink`.

... simplify `InstanceModifier`? Use `builders.modifiers.ValuesMixin`.

... inherit model classes from other model classes? At your own risk.

... make sure my random ID's dont collide? Use `builders.construct.Key` around your `Random`

... reuse the modifiers? They can be placed in a list and fed to the builder like this:

```
big_engine = InstanceModifier(Engine).thatSets(hp=1500)
big_wheels = InstanceModifier(Wheel).thatSets(size=25)

monster_car = [big_engine, big_wheels, InstanceModifier(Body).thatSets(color='red')]

my_monster = Builder(Car).withA(monster_car).build()  # indeed it is
```

... build something with a circular dependency? Add a proper `InstanceModifier().thatDoes()` to set non-tree-like references.

Basic tutorial

`builders` is intended to facilitate test data creation and achieves it per two major capabilities:

- *Describing* data structure via class-like model
- *Building* of a particular *finely-configured* data set

2.1 Describing data model

Data models are commonly considered as large trees of crossreferenced objects.

For example, to describe a decomposition of a convenient automobile one might draw something like that:
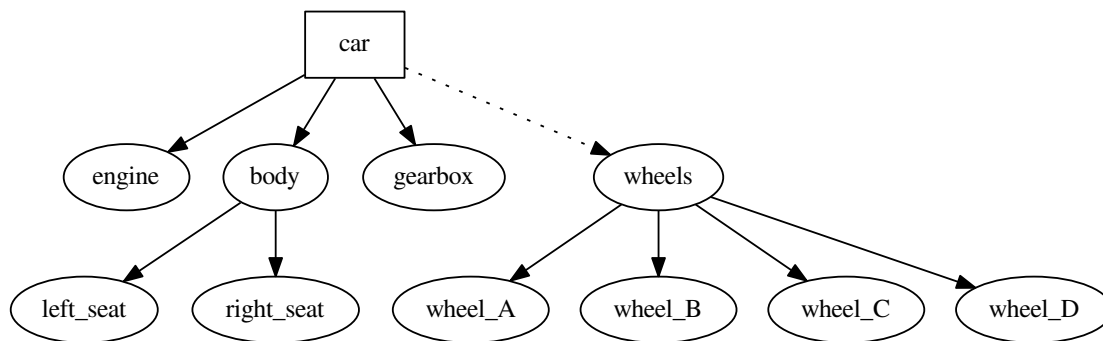
The diagram declares that car consists of engine, gearbox, body and a set of wheels, completely ommiting the properties of each object.

Same can be described with `builders` as follows:

```python
from builders.construct import Unique, Collection, Random

class Engine:
    hp = 100
    type = 'steam'

class Gearbox:
    gears = 4
    type = 'manual'
```

```python
class Seat:
    material = 'leather'

class Body:
    color = 'blue'
    left_seat = Unique(Seat)
    right_seat = Unique(Seat)

class Wheel:
    size = 14
    threading = 'diagonal'
    type = 'winter_spiked'

class Car:
    make = 'ford'
    year_of_make = Random(1990, 2005)

    engine = Unique(Engine)
    gearbox = Unique(Gearbox)
    body = Unique(Body)
    wheels = Collection(Wheel, number=4)
```

The example is mostly self-describing. However, note:

- each data model type has its own python class as a representation

- default attribute values are given in the classes in primitives

- references to other model types are declared via `Construct` attributes

- there is no explicit **root** element or mandatory base classes

2.2 Building model instance

Long story short, building the model is as easy as:

```python
from builders.builder import Builder

my_car = Builder(Car).build()

isinstance(my_car, Car)   # True
my_car.engine.hp == 100   # True
len(my_car.wheels)   # 4
type(my_car.wheels)   # list
my_car.wheels[0] == my_car.wheels[1]   # False, these are different wheels
1990 <= my_car.year_of_make <= 2005   # True, exact value of year_of_make varies
```

How this works? `Builder` recursevily walks over the tree starting with `Car` and instantiates model classes.

When a class instance is created, each attribute that is a `Construct` has its `build` method called. The resulting value is then assigned to that attribute of a built instance.

The `Unique` builds a single new instance of given type thus performing recursion step. `Collection` builds a number of new instances and puts them in a list.

There are several other useful constructs:

- `builders.construct.Random` generate a random number or string

- `builders.construct.Uid` generates a new UUID
- `builders.construct.Reused` works like `Unique`, but caches built values
- `builders.construct.Maybe` builds a nested construct in a certain conditions
- `builders.construct.Lambda` runs passed function with instance being constructed as parameter every time object is built

All the built-in constructs can be found at `builders.construct`. Custom constructs may be derived from `builders.construct.Construct`.

2.3 Modifying a tree

To build non-default model (and thats what you need most of the time) just apply some `Modifiers` to the tree like this:

```python
from builders.modifiers import InstanceModifier, NumberOf

my_car = Builder(Car).withA(NumberOf(Car.wheels, 5)).build()

len(my_car.wheels)  # 5, we told it to be so

my_car = Builder(Car).withA(InstanceModifier(Seat).thatSets(material='fabric')).build()

my_car.body.left_seat.material  # 'fabric'
my_car.body.right_seat.material  # 'fabric'
```

The `withA` method accepts a number of modifiers and returns same `Builder` for the sake of chaining:

```python
from builders.modifiers import InstanceModifier, NumberOf

Builder(Car).withA(NumberOf(Car.wheels, 5)).withA(InstanceModifier(Engine).thatSets(hp='over_9000'))
```

Obviously, configured `builder` can be used again to produce a another one similar car.

Useful built-in modifiers are:

- `builders.modifiers.InstanceModifier` factory that makes fancy `thatDoes`, `thatSets` and `thatSetsCarefully` modifiers,
- `builders.modifiers.NumberOf` that sets `Collection` sizes
- `builders.modifiers.OneOf` that modifies a `Collection` entry
- `builders.modifiers.Enabled` that turns on `builders.construct.Maybe`
- `builders.modifiers.LambdaModifier` replaces default function in `builders.construct.Lambda` with a given one
- `builders.modifiers.Another` adds one more element to a `Collection` with given modifiers

All the built-in modifiers can be found in `builders.modifiers`.

builders Package

3.1 `builder` Module

class `builders.builder.`**`Builder`**(*clazzToBuild*)

> Main interface class for the `builders` package.
>
> For example:
>
> ```
> class Bar:
>     bar = 1
>
>
> class Foo:
>     baz = 10
>     bars = Collection(Bar)
>
>
> my_foo = Builder(Foo).withA(NumberOf(Foo.bars, 5)).build()
> ```
>
> **`build`**()
>
> > Build the resulting instance with the respect of all of the previously applied modifiers.
>
> **`withA`**(**modifiers*)
>
> > **Parameters modifiers** – list of modifiers to apply
> >
> > Apply a number of modifiers to this builder. Each modifier can be either a single `builders.modifiers.Modifier` or a nested list structure of them.
> >
> > Modifiers are stored in the builder and executed on the `build` call.

`builders.builder.`**`flatten`**(*l*)

> > **Parameters l** – iterable to flatten
>
> Generator that flattens iterable infinitely. If an item is iterable, `flatten` descends on it. If it is callable, it descends on the call result (with no arguments), and it yields the item itself otherwise.

3.2 `construct` Module

class `builders.construct.`**`Construct`**

> Bases: `builders.construct.Link`
>
> Base class for build-generated attributes. Subclasses should implement *doBuild* method.

build(**args*, ***kwargs*)
>   Called by `builders.builder.Builder` on the model construction. Returns actual pre-set value (via `Link` mechanism) or a newly built one.

class `builders.construct.`**Predefined**(*value*)
>   Bases: `builders.construct.Construct`

>   Builds to a predefined `value`.

class `builders.construct.`**Unique**(*typeToBuild*)
>   Bases: `builders.construct.Construct`

>   Builds a new instance of `type` with a separate `builders.Builder` with respect to currently active modifiers.

class `builders.construct.`**Collection**(*typeToBuild*, *number=1*)
>   Bases: `builders.construct.Unique`

>   Builds a `list` of new `typeToBuild` objects. With no modifiers, list will contain `number` entries.

class `builders.construct.`**Reused**(*typeToBuild*, *local=False*, *keys=*[ ])
>   Bases: `builders.construct.Unique`

>   Like `Unique`, but with caching.

>   Stores all the built instances within a dictionary. If the would-be-new-instance has key equal to some of the objects in cache, cached object is returned.

>   Key is a tuple of `typeToBuild` and selected attribute values.

>>   **Parameters**
>>
>>   - **local** – keep cache in the *Reused* instance. If false, cache is global (eww).
>>
>>   - **keys** – list of attributes that are considered key components along with the *typeToBuild*.

class `builders.construct.`**Random**(*start=1*, *end=100500*, *pattern=None*)
>   Bases: `builders.construct.Construct`

>>   **Parameters**
>>
>>   - **start** – random interval start
>>
>>   - **end** – random interval end
>>
>>   - **pattern** – a string %-pattern with single non-positional argument

>   A construct that results in a random integer or random string. If `pattern` is present, it is formatted with the random value.

class `builders.construct.`**Maybe**(*construct*, *enabled=False*)
>   Bases: `builders.construct.Construct`

>   Returns result of nested `construct` if `enabled`.

>   See `builders.modifiers.Enabled` to turn it on.

class `builders.construct.`**Uplink**(*reusing_by=*[ ])
>   Bases: `builders.construct.Construct`

>   Becomes a value of another `Construct` when it is build.

>   Call `linksTo` on `Uplink` object to set destination.

>   Supplying `reusing_by` emulates `Reused` behavior with given `keys`.

> **Warning:** reusing_by is not fully operational at the moment, use at your own risk. See test_uplink.test_reuse – there are commented checks.

class builders.construct.**Uid**
    Bases: builders.construct.Construct

    Builds to a string with a fresh uuid.uuid4()

class builders.construct.**Key**(*value_construct*)
    Bases: builders.construct.Construct

    Tries to obtain fresh items from value_construct upon build via checking new item against all the previously built ones.

        Raises Exception if it fails to get a non-used value after a meaningful number of attempts.

    Intended to be used with Random to prevent key collisions like:

```python
class MyFoo:
    id = Key(Random())
```

class builders.construct.**Lambda**(*functionToExecute*)
    Bases: builders.construct.Construct

    Function, executed during each build with an instance being constructed passed in as parameter

3.3 modifiers Module

class builders.modifiers.**Modifier**
    Bases: object

    Base class for build process modifiers. Child classes should implement apply method.

    apply(**args*, ***kwargs*)
        Perform the actual modification. kwargs can contain different parameters – modifier is encouraged to check actual values supplied. See builders.builder.Builder to find out how this is invoked.

    shouldRun(**args*, ***kwargs*)
        Determines if the modifier should run on this particular occasion

        Parameters are similar to the apply method

class builders.modifiers.**InstanceModifier**(*classToRunOn*)
    Modifier factory that builds new modifiers to act upon instances of classToRunOn.

    InstanceModifier(foo).thatDoes(bar) returns modifier that calls bar(x) on the foo istances x

    InstanceModifier(foo).thatSets(a=1, b=2) returns modifier that sets foo instance attributes a to 1 and b to 2

    InstanceModifier(foo).thatCarefullySets(c=2) returns modifier that sets foo instance attributes c to 2 if that instance already has c attribute and raises exception if it does not

    thatCarefullySets(***kwargs*)
        as *thatSets* factory method, but asserts that attribute exists

    thatDoes(*action*)
        factory method that builds an instance backed by a given callable action

    thatSets(***kwargs*)
        factory method that builds a modifier that sets given kwargs as attributes for the instance

class builders.modifiers.**ValuesMixin**

> Bases: object

> Syntactic sugar for InstanceModifier.thatCarefullySets. Use it like:

```
class Foo(ValuesMixin):
  bar = 0

class Baz:
  foo = Unique(Foo)

baz = Builder(Baz).withA(Foo.values(bar=2)).build()
```

class builders.modifiers.**ClazzModifier**

> Bases: builders.modifiers.Modifier

> Base class for Modifier siblings that act at classes.

> Siblings should implement do method.

> See builders.builder.Builder to see the actual invocation.

class builders.modifiers.**ConstructModifier**(*construct*)

> Bases: builders.modifiers.ClazzModifier

> Base class for ClazzModifier that work on a particular construct object within a class

> Siblings should implement doApply method.

class builders.modifiers.**Given**(*construct*, *value*)

> Bases: builders.modifiers.ConstructModifier

> Supplied pre-defined value for a given construct.

class builders.modifiers.**NumberOf**(*what*, *amount*)

> Bases: builders.modifiers.ConstructModifier

> Sets the target number of builders.constructs.Collection elements to a given amount

class builders.modifiers.**HavingIn**(*what*, **instances*)

> Bases: builders.modifiers.ConstructModifier

> Adds instances to a given builders.constructs.Collection.

> If instance is a number, **that much** new instances are added to the Collection target number.

> Else, that instance is added to the Collection as a pre-built one.

class builders.modifiers.**OneOf**(*what*, **modifiers*)

> Bases: builders.modifiers.ConstructModifier

> Applies given modifiers to one of objects build by builders.construct.Collection.

class builders.modifiers.**Enabled**(*what*)

> Bases: builders.modifiers.ConstructModifier

> Turns on given builders.construct.Maybe once.

class builders.modifiers.**Disabled**(*what*)

> Bases: builders.modifiers.Enabled

> Like Enabled, but the other way around.

class builders.modifiers.**LambdaModifier**(*construct*, *new_lambda*)

> Bases: builders.modifiers.ConstructModifier

> Replaces function in builders.construct.Lambda with given new_lambda

builders.modifiers.**Another**(*collection*, **modifiers*)
    Add another instance to given `collection` with given `mod`

Indices and tables

- *genindex*
- *modindex*
- *search*

b

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W