
django-model-values

Release 1.2

Aug 05, 2020

Contents

1	Introduction	3
2	Updates	5
3	Selects	7
4	Aggregation	9
5	Expressions	11
6	Conditionals	13
7	Contents	15
8	Indices and tables	25
	Python Module Index	27
	Index	29

Taking the **O** out of **ORM**.

CHAPTER 1

Introduction

Provides Django model utilities for encouraging direct data access instead of unnecessary object overhead. Implemented through compatible method and operator extensions¹ to `QuerySets` and `Managers`.

The primary motivation is the experiential observation that the active record pattern - specifically `Model.save` - is the root of all evil. The secondary goal is to provide a more intuitive data layer, similar to PyData projects such as `pandas`.

Usage: instantiate the `custom manager` in your models.

¹ The only incompatible changes are edge cases which aren't documented behavior, such as queryset comparison.

CHAPTER 2

Updates

The Bad:

```
book = Book.objects.get(pk=pk)
book.rating = 5.0
book.save()
```

This example is ubiquitous and even encouraged in many django circles. It's also an epic fail.

- Runs an unnecessary select query, as no fields need to be read.
- Updates all fields instead of just the one needed.
- Therefore also suffers from race conditions.
- And is relatively verbose, without addressing errors yet.

The solution is relatively well-known, and endorsed by [django's own docs](#), but remains under-utilized.

The Ugly:

```
Book.objects.filter(pk=pk).update(rating=5.0)
```

So why not provide syntactic support for the better approach. The [Manager](#) supports filtering by primary key, since that's so common. The [QuerySet](#) supports column updates.

The Good:

```
Book.objects[pk]['rating'] = 5.0
```

But one might posit...

- “Isn't the encapsulation `save` provides worth it in principle?”
- “Doesn't the new `update_fields` option fix this in practice?”
- “What if the object is cached or has custom logic in the `save` method?”

No, no, and good luck with that.² Consider a more realistic example which addresses these concerns.

The Bad:

```
try:
    book = Book.objects.get(pk=pk)
except Book.DoesNotExist:
    changed = False
else:
    changed = book.publisher != publisher
    if changed:
        book.publisher = publisher
        book.pubdate = today
        book.save(update_fields=['publisher', 'pubdate'])
```

This solves the most severe problem, though with more verbosity and still an unnecessary read.³ Note handling `pubdate` in the `save` implementation would only spare the caller one line of code. But the real problem is how to handle custom logic when `update_fields` *isn't* specified. There's no one obvious correct behavior, which is why projects like [django-model-utils](#) have to track the changes on the object itself.⁴

A better approach would be an `update_publisher` method which does all and only what is required. So what would such an implementation be? A straight-forward update won't work, yet only a minor tweak is needed.

The Ugly:

```
changed = Book.objects.filter(pk=pk).exclude(publisher=publisher) \
    .update(publisher=publisher, pubdate=today)
```

Now the update is only executed if necessary. And this can be generalized with a little inspiration from `{get, update}_or_create`.

The Good:

```
changed = Book.objects[pk].change({'pubdate': today}, publisher=publisher)
```

² In the *vast* majority of instances of that idiom, the object is immediately discarded and no custom logic is necessary. Furthermore the dogma of a model knowing how to serialize itself doesn't inherently imply a single all-purpose instance method. Specialized classmethods or manager methods would be just as encapsulated.

³ Premature optimization? While debatable with respect to general object overhead, nothing good can come from running superfluous database queries.

⁴ Supporting `update_fields` with custom logic also results in complex conditionals, ironic given that OO methodology ostensibly favors separate methods over large switch statements.

CHAPTER 3

Selects

Direct column access has some of the clunkiest syntax: `values_list(..., flat=True)`. `QuerySets` override `__getitem__`, as well as comparison operators for simple filters. Both are common syntax in panel data layers.

The Bad:

```
{book.pk: book.name for book in qs}

(book.name for book in qs.filter(name__isnull=False))

if qs.filter(author=author):
```

The Ugly:

```
dict(qs.values_list('pk', 'name'))

qs.exclude(name=None).values_list('name', flat=True)

if qs.filter(author=author).exists():
```

The Good:

```
dict(qs['pk', 'name'])

qs['name'] != None

if author in qs['author']:
```


CHAPTER 4

Aggregation

Once accustomed to working with data values, a richer set of aggregations becomes possible. Again the method names mirror projects like `pandas` whenever applicable.

The Bad:

```
collections.Counter(book.author for book in qs)

sum(book.rating for book in qs) / len(qs)

counts = collections.Counter()
for book in qs:
    counts[book.author] += book.quantity
```

The Ugly:

```
dict(qs.values_list('author').annotate(model.Count('author'))))

qs.aggregate(models.Avg('rating'))['rating__avg']

dict(qs.values_list('author').annotate(models.Sum('quantity')))
```

The Good:

```
dict(qs['author'].value_counts())

qs['rating'].mean()

dict(qs['quantity'].groupby('author').sum())
```


CHAPTER 5

Expressions

F expressions are similarly extended to easily create Q, Func, and OrderBy objects. Note they can be used directly even without a custom manager.

The Bad:

```
(book for book in qs if book.author.startswith('A') or book.author.startswith('B'))

(book.title[:10] for book in qs)

for book in qs:
    book.rating += 1
    book.save()
```

The Ugly:

```
qs.filter(Q(author__startswith='A') | Q(author__startswith='B'))

qs.values_list(functions.Substr('title', 1, 10), flat=True)

qs.update(rating=models.F('rating') + 1)
```

The Good:

```
qs[F.any(map(F.author.startswith, 'AB'))]

qs[F.title[:10]]

qs['rating'] += 1
```


CHAPTER 6

Conditionals

Annotations and updates with Case and When expressions. See also [bulk_changed](#) and [bulk_change](#) for efficient bulk operations on primary keys.

The Bad:

```
collections.Counter('low' if book.quantity < 10 else 'high' for book in qs).items()

for author, quantity in items:
    for book in qs.filter(author=author):
        book.quantity = quantity
        book.save()
```

The Ugly:

```
qs.values_list(models.Case(
    models.When(quantity__lt=10, then=models.Value('low')),
    models.When(quantity__gte=10, then=models.Value('high')),
    output_field=models.CharField(),
)).annotate(count=models.Count('*'))

cases = (models.When(author=author, then=models.Value(quantity)) for author, quantity_
↪ in items)
qs.update(quantity=models.Case(*cases, default='quantity'))
```

The Good:

```
qs[F.quantity < 10: 'low', F.quantity >= 10: 'high'].value_counts()

qs['quantity'] = {F.author == author: quantity for author, quantity in items}
```


7.1 Lookup

class `model_values.Lookup`
Mixin for field lookups.

Note: Spatial lookups require `gis` to be enabled.

`__ge__` (*value*)
gte

`__gt__` (*value*)
gt

`__le__` (*value*)
lte

`__lshift__` (*value*)
left

`__lt__` (*value*)
lt

`__ne__` (*value*)
ne

`__rshift__` (*value*)
right

above (*value*)
strictly_above

below (*value*)
strictly_below

contained (*value*)

contains (*value*, *properly=False*, *bb=False*)

Return whether field *contains* the value. Options apply only to geom fields.

Parameters

- **properly** – *contains_properly*
- **bb** – bounding box, *bbcontains*

coveredby (*value*)

covers (*value*)

crosses (*value*)

disjoint (*value*)

endswith (*value*)

equals (*value*)

icontains (*value*)

iendswith (*value*)

iexact (*value*)

intersects (*value*)

iregex (*value*)

is_valid

Whether field *isvalid*.

isin (*value*)
in

startswith (*value*)

left (*value*)

overlaps (*geom*, *position=""*, *bb=False*)

Return whether field *overlaps* with geometry .

Parameters

- **position** – *overlaps_{left, right, above, below}*
- **bb** – bounding box, *bboverlaps*

range (**values*)

regex (*value*)

relate (**values*)

right (*value*)

startswith (*value*)

touches (*value*)

within (*geom*, *distance=None*)

Return whether field is *within* geometry.

Parameters **distance** – *dwithin*

7.2 F

class `model_values.F(name)`

Bases: `django.db.models.expressions.F`, `model_values.Lookup`

Create F, Q, and Func objects with expressions.

F creation supported as attributes: `F.user == F('user')`, `F.user.created == F('user__created')`.

Q lookups supported as methods or operators: `F.text.iexact(...) == Q(text__iexact=...)`, `F.user.created >= ... == Q(user__created__gte=...)`.

Func objects also supported as methods: `F.user.created.min() == Min('user__created')`.

Note: Since attributes are used for constructing *F* objects, there may be collisions between field names and methods. For example, `name` is a reserved attribute, but the usual constructor can still be used: `F('name')`.

Note: See source for available spatial functions if *gis* is configured.

lookups

mapping of potentially [registered lookups](#) to transform functions

`__abs__`

Abs

`__call__(*args, **extra) → django.db.models.expressions.Func`

Call self as a function.

`__ceil__`

Ceil

`__eq__(value, lookup: str = "") → django.db.models.query_utils.Q`

Return Q object with lookup.

`__floor__`

Floor

`__getattr__(name: str) → model_values.F`

Return new *F* object with chained attribute.

`__getitem__(slc: slice) → django.db.models.expressions.Func`

Return field Substr or Right.

`__hash__()`

Return hash(self).

`__mod__`

Mod

`__ne__(value) → django.db.models.query_utils.Q`

Allow `__ne=None` lookup without custom queryset.

`__pow__`

Power

`__reversed__`

Reverse

round
Round

cast
Coerce an expression to a new field type.

coalesce
Return, from left to right, the first non-null expression.

concat
Concatenate text fields together. Backends that result in an entire null expression when any arguments are null will wrap each argument in coalesce functions to ensure a non-null result.

count ()
Return Count with optional field.

cume_dist
CumeDist

dense_rank
DenseRank

extract
Extract

find (sub, **extra) → django.db.models.expressions.Expression
Return StrIndex with str.find semantics.

first_value
FirstValue

greatest
Return the maximum expression.

If any expression is null the return value is database-specific: On PostgreSQL, the maximum not-null expression is returned. On MySQL, Oracle, and SQLite, if any expression is null, null is returned.

lag
Lag

last_value
LastValue

lead
Lead

least
Return the minimum expression.

If any expression is null the return value is database-specific: On PostgreSQL, return the minimum not-null expression. On MySQL, Oracle, and SQLite, if any expression is null, return null.

ljust (width: int, fill=' ', **extra) → django.db.models.expressions.Func
Return LPad with wrapped values.

log (base=2.718281828459045, **extra) → django.db.models.expressions.Func
Return Log, by default Ln.

lstrip
LTrim

max
Max

mean
Avg

min
Min

now
alias of `django.db.models.functions.datetime.Now`

nth_value
NthValue

ntile
alias of `django.db.models.functions.window.Ntile`

nullif
NullIf

percent_rank
PercentRank

rank
Rank

repeat
Repeat

replace (*old*, *new*=', **extra') → `django.db.models.expressions.Func`
Return Replace with wrapped values.

rjust (*width*: int, *fill*=' ', **extra') → `django.db.models.expressions.Func`
Return RPad with wrapped values.

row_number
RowNumber

rstrip
RTrim

sha1
SHA1

sha224
SHA224

sha256
SHA256

sha384
SHA384

sha512
SHA512

std
StdDev

strip
Trim

sum
Sum

trunc
Trunc

var
Variance

7.3 QuerySet

class `model_values.QuerySet` (*model=None, query=None, using=None, hints=None*)
 Bases: `django.db.models.query.QuerySet`, `model_values.Lookup`

Note: See source for available aggregate spatial functions if `gis` is configured.

__add__ (*value*)
add

__contains__ (*value*)
Return whether value is present using `exists`.

__eq__ (*value, lookup: str = ''*) → `model_values.QuerySet`
Return *QuerySet* filtered by comparison to given value.

__getitem__ (*key*)
Allow column access by field names, expressions, or F objects.

`qs[field]` returns flat `values_list`

`qs[field, ...]` returns tupled `values_list`

`qs[Q_obj]` provisionally returns filtered *QuerySet*

__iter__ ()
Iteration extended to support `groupby()`.

__mod__ (*value*)
mod

__mul__ (*value*)
mul

__pow__ (*value*)
pow

__setitem__ (*key, value*)
Update a single column.

__sub__ (*value*)
sub

__truediv__ (*value*)
truediv

annotate (**args, **kwargs*) → `model_values.QuerySet`
 Annotate extended to also handle mapping values, as a *Case* expression.

Parameters **kwargs** – `field={Q_obj: value, ...}, ...`

As a provisional feature, an optional `default` key may be specified.

change (*defaults: Mapping[KT, VT_co] = {}, **kwargs*) → int

Update and return number of rows that actually changed.

For triggering on-change logic without fetching first.

if `qs.change(status=...)`: status actually changed

`qs.change({'last_modified': now}, status=...)` last_modified only updated if status updated

Parameters defaults – optional mapping which will be updated conditionally, as with `update_or_create`.

changed (***kwargs*) → dict

Return first mapping of fields and values which differ in the db.

Also efficient enough to be used in boolean contexts, instead of `exists`.

exists (*count: int = 1*) → bool

Return whether there are at least the specified number of rows.

groupby (**fields, **annotations*) → model_values.QuerySet

Return a grouped *QuerySet*.

The queryset is iterable in the same manner as `itertools.groupby`. Additionally the *reduce()* functions will return annotated querysets.

items (**fields, **annotations*) → model_values.QuerySet

Return annotated `values_list`.

max ()

Max

mean ()

Avg

min ()

Min

reduce (**funcs*)

Return aggregated values, or an annotated *QuerySet* if *groupby()* is in use.

Parameters funcs – aggregation function classes

sort_values (*reverse=False*) → model_values.QuerySet

Return *QuerySet* ordered by selected values.

std ()

StdDev

sum ()

Sum

update (***kwargs*) → int

Update extended to also handle mapping values, as a *Case* expression.

Parameters kwargs – `field={Q_obj: value, ...}, ...`

value_counts (*alias: str = 'count'*) → model_values.QuerySet

Return annotated value counts.

var ()

Variance

7.4 Manager

class `model_values.Manager`

Bases: `django.db.models.manager.Manager`

__contains__ (*pk*)

Return whether primary key is present using `exists`.

__delitem__ (*pk*)

Delete row with primary key.

__getitem__ (*pk*) → `model_values.QuerySet`

Return *QuerySet* which matches primary key.

To encourage direct db access, instead of always using `get` and `save`.

bulk_change (*field*, *data*: *Mapping*[*KT*, *VT_co*], *key*: *str* = 'pk', *conditional*=*False*, ***kwargs*) → *int*

Update changed rows with a minimal number of queries, by inverting the data to use `pk__in`.

Parameters

- **field** – value column
- **data** – {pk: value, ...}
- **key** – unique key column
- **conditional** – execute select query and single conditional update; may be more efficient if the percentage of changed rows is relatively small
- **kwargs** – additional fields to be updated

bulk_changed (*field*, *data*: *Mapping*[*KT*, *VT_co*], *key*: *str* = 'pk') → *dict*

Return mapping of values which differ in the db.

Parameters

- **field** – value column
- **data** – {pk: value, ...}
- **key** – unique key column

get_queryset ()

Return a new *QuerySet* object. Subclasses can override this method to customize the behavior of the *Manager*.

upsert (*defaults*: *Mapping*[*KT*, *VT_co*] = {}, ***kwargs*) → *Union*[*int*, `django.db.models.base.Model`]

Update or insert returning number of rows or created object.

Faster and safer than `update_or_create`. Supports combined expression updates by assuming the identity element on insert: `F(...) + 1`.

Parameters defaults – optional mapping which will be updated, as with `update_or_create`.

7.5 Case

class `model_values.Case` (*conds*, *default*=*None*, ***extra*)

Bases: `django.db.models.expressions.Case`

Case expression from mapping of when conditionals.

Parameters

- **conds** – {Q_obj: value, ...}
- **default** – optional default value or F object
- **output_field** – optional field defaults to registered types

types = {<class 'str'>: <class 'django.db.models.fields.CharField'>, <class 'int'>:
mapping of types to output fields

7.6 classproperty

class model_values.classproperty

Bases: property

A property bound to a class.

7.7 EnumField

model_values.EnumField(enum, display: Callable = None, **options) →
django.db.models.fields.Field

Return a CharField or IntegerField with choices from given enum.

By default, enum names and values are used as db values and display labels respectively, returning a CharField with computed max_length.

Parameters **display** – optional callable to transform enum names to display labels, thereby using enum values as db values and also supporting integers.

7.8 Example

An example Model used in the tests.

```
from django.db import models
from model_values import F, Manager, classproperty

class Book(models.Model):
    title = models.TextField()
    author = models.CharField(max_length=50)
    quantity = models.IntegerField()
    last_modified = models.DateTimeField(auto_now=True)

    objects = Manager()
```

7.8.1 Table logic

Django recommends model methods for row-level functionality, and custom managers for table-level functionality. That's fine if the custom managers are reused across models, but often they're just custom filters, and specific to a model. As evidenced by [django-model-utils](#)' QueryManager.

There's a simpler way to achieve the same end: a model `classmethod`. In some cases a proliferation of classmethods is an anti-pattern, but in this case functions won't suffice. It's Django that attached the `Manager` instance to a class.

Additionally a `classproperty` wrapper is provided, to mimic a custom `Manager` or `Queryset` without calling it first.

```
@classproperty
def in_stock(cls):
    return cls.objects.filter(F.quantity > 0)
```

7.8.2 Row logic

Some of the below methods may be added to a model mixin in the future. It's a delicate balance, as the goal is to *not* encourage object usage. However, sometimes having an object already is inevitable, so it's still worth considering best practices given that situation.

Providing wrappers for any manager method that's pk-based may be worthwhile, particularly a filter to match only the object.

```
@property
def object(self):
    return type(self).objects[self.pk]
```

From there one can easily imagine other useful extensions.

```
def changed(self, **kwargs):
    return self.object.changed(**kwargs)

def update(self, **kwargs):
    for name in kwargs:
        setattr(self, name, kwargs[name])
    return self.object.update(**kwargs)
```

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

m

`model_values`, [15](#)

Symbols

__abs__ (*model_values.F attribute*), 17
 __add__ () (*model_values.QuerySet method*), 20
 __call__ () (*model_values.F method*), 17
 __ceil__ (*model_values.F attribute*), 17
 __contains__ () (*model_values.Manager method*), 22
 __contains__ () (*model_values.QuerySet method*), 20
 __delitem__ () (*model_values.Manager method*), 22
 __eq__ () (*model_values.F method*), 17
 __eq__ () (*model_values.QuerySet method*), 20
 __floor__ (*model_values.F attribute*), 17
 __ge__ () (*model_values.Lookup method*), 15
 __getattr__ () (*model_values.F method*), 17
 __getitem__ () (*model_values.F method*), 17
 __getitem__ () (*model_values.Manager method*), 22
 __getitem__ () (*model_values.QuerySet method*), 20
 __gt__ () (*model_values.Lookup method*), 15
 __hash__ () (*model_values.F method*), 17
 __iter__ () (*model_values.QuerySet method*), 20
 __le__ () (*model_values.Lookup method*), 15
 __lshift__ () (*model_values.Lookup method*), 15
 __lt__ () (*model_values.Lookup method*), 15
 __mod__ (*model_values.F attribute*), 17
 __mod__ () (*model_values.QuerySet method*), 20
 __mul__ () (*model_values.QuerySet method*), 20
 __ne__ () (*model_values.F method*), 17
 __ne__ () (*model_values.Lookup method*), 15
 __pow__ (*model_values.F attribute*), 17
 __pow__ () (*model_values.QuerySet method*), 20
 __reversed__ (*model_values.F attribute*), 17
 __round__ (*model_values.F attribute*), 17
 __rshift__ () (*model_values.Lookup method*), 15
 __setitem__ () (*model_values.QuerySet method*), 20
 __sub__ () (*model_values.QuerySet method*), 20
 __truediv__ () (*model_values.QuerySet method*), 20

A

above () (*model_values.Lookup method*), 15
 annotate () (*model_values.QuerySet method*), 20

B

below () (*model_values.Lookup method*), 15
 bulk_change () (*model_values.Manager method*), 22
 bulk_changed () (*model_values.Manager method*), 22

C

Case (*class in model_values*), 22
 cast (*model_values.F attribute*), 18
 change () (*model_values.QuerySet method*), 20
 changed () (*model_values.QuerySet method*), 21
 classproperty (*class in model_values*), 23
 coalesce (*model_values.F attribute*), 18
 concat (*model_values.F attribute*), 18
 contained () (*model_values.Lookup method*), 15
 contains () (*model_values.Lookup method*), 15
 count () (*model_values.F method*), 18
 coveredby () (*model_values.Lookup method*), 16
 covers () (*model_values.Lookup method*), 16
 crosses () (*model_values.Lookup method*), 16
 cume_dist (*model_values.F attribute*), 18

D

dense_rank (*model_values.F attribute*), 18
 disjoint () (*model_values.Lookup method*), 16

E

endswith () (*model_values.Lookup method*), 16
 EnumField () (*in module model_values*), 23
 equals () (*model_values.Lookup method*), 16
 exists () (*model_values.QuerySet method*), 21
 extract (*model_values.F attribute*), 18

F

F (*class in model_values*), 17

`find()` (*model_values.F method*), 18
`first_value` (*model_values.F attribute*), 18

G

`get_queryset()` (*model_values.Manager method*), 22
`greatest` (*model_values.F attribute*), 18
`groupby()` (*model_values.QuerySet method*), 21

I

`icontains()` (*model_values.Lookup method*), 16
`iendswith()` (*model_values.Lookup method*), 16
`iexact()` (*model_values.Lookup method*), 16
`intersects()` (*model_values.Lookup method*), 16
`iregex()` (*model_values.Lookup method*), 16
`is_valid` (*model_values.Lookup attribute*), 16
`isin()` (*model_values.Lookup method*), 16
`startswith()` (*model_values.Lookup method*), 16
`items()` (*model_values.QuerySet method*), 21

L

`lag` (*model_values.F attribute*), 18
`last_value` (*model_values.F attribute*), 18
`lead` (*model_values.F attribute*), 18
`least` (*model_values.F attribute*), 18
`left()` (*model_values.Lookup method*), 16
`ljust()` (*model_values.F method*), 18
`log()` (*model_values.F method*), 18
`Lookup` (*class in model_values*), 15
`lookups` (*model_values.F attribute*), 17
`lstrip` (*model_values.F attribute*), 18

M

`Manager` (*class in model_values*), 22
`max` (*model_values.F attribute*), 18
`max()` (*model_values.QuerySet method*), 21
`mean` (*model_values.F attribute*), 18
`mean()` (*model_values.QuerySet method*), 21
`min` (*model_values.F attribute*), 19
`min()` (*model_values.QuerySet method*), 21
`model_values` (*module*), 15

N

`now` (*model_values.F attribute*), 19
`nth_value` (*model_values.F attribute*), 19
`ntile` (*model_values.F attribute*), 19
`nullif` (*model_values.F attribute*), 19

O

`overlaps()` (*model_values.Lookup method*), 16

P

`percent_rank` (*model_values.F attribute*), 19

Q

`QuerySet` (*class in model_values*), 20

R

`range()` (*model_values.Lookup method*), 16
`rank` (*model_values.F attribute*), 19
`reduce()` (*model_values.QuerySet method*), 21
`regex()` (*model_values.Lookup method*), 16
`relate()` (*model_values.Lookup method*), 16
`repeat` (*model_values.F attribute*), 19
`replace()` (*model_values.F method*), 19
`right()` (*model_values.Lookup method*), 16
`rjust()` (*model_values.F method*), 19
`row_number` (*model_values.F attribute*), 19
`rstrip` (*model_values.F attribute*), 19

S

`sha1` (*model_values.F attribute*), 19
`sha224` (*model_values.F attribute*), 19
`sha256` (*model_values.F attribute*), 19
`sha384` (*model_values.F attribute*), 19
`sha512` (*model_values.F attribute*), 19
`sort_values()` (*model_values.QuerySet method*), 21
`startswith()` (*model_values.Lookup method*), 16
`std` (*model_values.F attribute*), 19
`std()` (*model_values.QuerySet method*), 21
`strip` (*model_values.F attribute*), 19
`sum` (*model_values.F attribute*), 19
`sum()` (*model_values.QuerySet method*), 21

T

`touches()` (*model_values.Lookup method*), 16
`trunc` (*model_values.F attribute*), 19
`types` (*model_values.Case attribute*), 23

U

`update()` (*model_values.QuerySet method*), 21
`upsert()` (*model_values.Manager method*), 22

V

`value_counts()` (*model_values.QuerySet method*), 21
`var` (*model_values.F attribute*), 20
`var()` (*model_values.QuerySet method*), 21

W

`within()` (*model_values.Lookup method*), 16