Harness the power of Python magic methods and lazy objects.

By Sep Dehpour

Aug 2016

zepworks.com

sep at zepworks.com

https://github.com/seperman/redisworks

Lazy Loading

Defer initialization of an object until the point at which it is needed.

Why lazy?

- Better performace (depending on your design)
- Better **illusion** of performance (when dealing with heavy objects)
- **Less** hits to your database (depending on your design)

Why not lazy?

- Inconsistent state.
- Code complexity.
- **More** hits to your database (depending on your design).

Why Lazy?

Better performance, less hits to database

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q) # <-- evaluated here
```

Why Lazy?

Better performance, less hits to database

Example:

https://github.com/seperman/benchmark/blob/master/pyredis_benchmark.py

Fetching 100 keys from Redis

100 x Get one key: 4.84 miliseconds

1 x Mget 100 keys: 0.93 miliseconds

Why Lazy?

Better *illusion* of performance

Example:

- Unlimited scroll.
- Load chunks of big graph as you need.

Why Lazy?

Less memory usage

- Load chunks of big graph as you need.

Why *not* lazy?

Code maintainability

Why *not* lazy?

Inconsistent state

Why *not* lazy?

More hits to your database

```
for obj in all_lazy_objects:
    print(obj)  # evaluates one by one in a bad design.
```

Case Study

Create a Redis client.

```
<html>
<head>
<title>{{ title }}</title>
</head>
<body>
Hello,
We do {{ x }}, {{ y }} and {{ z }}.
{{ footer }}
</body>
</html>
```

option 1: multiple get

Multiple get requests to Redis

```
context = dict(
    title = redis.get('root.homepage.title'),
    x = redis.get('root.things.x'),
    y = redis.get('root.things.y'),
    z = redis.get('root.things.z'),
    footer = redis.get('root.footer')
)
```

```
<html>
<head>
<title>{{ title }}</title>
</head>
<body>
Hello,
We do {{ x }}, {{ y }} and {{ z }}.
{{ footer }}
</body>
</html>
```

option 2: mget

Multiple get requests to Redis

```
context = dict(
    title, x, y, z, footer = redis.mget('root.homepage.title',
                                       'root.things.x',
                                       'root.things.y',
                                       'root.things.z',
                                       'root.footer')
)
```

```
<html>
<head>
<title>{{ title }}</title>
</head>
<body>
Hello,
We do {{ x }}, {{ y }} and {{ z }}.
{{ footer }}
</body>
</html>
```

option 3: lazy

Let the frontend guys handle it.

```
context = {'root': root}
```

```
<html>
<head>
<title>{{ root.homepage.title }}</title>
</head>
<body>
Hello,
We do {{ root.things.x }}, {{ root.things.y }} and {{ root.things.z }}.

{{ root.footer }}
</body>
</html>
```

Create a Redis Client

Redis data types overview

- String

  `get`, `mget`, `set`, `mset (m = multi)`
- List

  `lrange (get a range)`, `rpush (append)`, `rpop (pop)`
- Hash (Dictionary)

  `hset (set a key)`, `hsetall (set a whole dict)`, `hgetall (get a whole dict)`
- Set

  `sadd`, `smembers`

Create a Redis client

Step 1: Load strings

```
>>> print(root.something)
value of root.something in Redis
```

Python magic methods

`__method__`

Python magic methods

`__init__`

Python magic methods

`__new__`

Python magic methods

`__del__`

Garbage Collector runs `__del__`

Magic methods

Protocol for containers (to define containers like lists,...)	Descriptor (custom class attributes that we fully control)	Called when no attribute found
`__getitem__(self, key)`	`__get__(self, obj, cls=None)`	`__getattr__(self, name)`
`__setitem__(self, key, value)`	`__set__(self, obj, val)`	
`__delitem__(self, key)`	`__delete__(self, obj)`	

Magic methods

Called whether the attribute is found or not	Called when garbage collecting
.	
`__setattr__(self, name, value)`	
`__delattr__(self, name)`	`__del__(self)` Not recommended. Instead use `__exit__` (context manager)

Step 1: load

Goal:

```
>>> print(root.something)
value of root.something in Redis
```

__getattr__

```
class Root:
    def __getattr__(self, key):
        class_name = self.__class__.__name__.lower()
        return redis.get("{}.{}".format(class_name, key))

>>> print(root.something)
value of root.something in Redis

>>> print(root.anotherthing)
value of root.anotherthing in Redis
```

Step 2: save

Goal:

```
>>> root.something = "value"
```

```
$ redis-cli
127.0.0.1:6379> get root.something
"value"
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
    return redis.get("{}.{}".format(self.class_name, key))
RecursionError: maximum recursion depth exceeded while calling a Python object
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
    return redis.get("{}.{}".format(self.class_name, key))
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Hint: in Python when maximum depth recursion: `key = "class_name"`

2 keys in Redis: `class_name` and `something`

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
    return redis.get("{}.{}".format(self.class_name, key))
RecursionError: maximum recursion depth exceeded while calling a Python object
```

`__getattr__` Called when no attribute found

`__setattr__` Called whether the attribute is found or not

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

1. **__init__** runs.

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

1. **__init__** runs.
2. self.class_name needs to be set.

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
6. print(root.something)
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
6. print(root.something)
7. __getattr__ for `something` is run.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
6. print(root.something)
7. __getattr__ for `something` is run.
8. in order to get the key from Redis, it needs to get self.class_name.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
6. print(root.something)
7. __getattr__ for `something` is run.
8. in order to get the key from Redis, it needs to get self.class_name.
9. But self.class_name was never set on the object. It was saved into Redis.
```

```python
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def __getattr__(self, key):
        return redis.get("{}.{}".format(self.class_name, key))

    def __setattr__(self, key, value):
        redis.set(key, value)

root = Root()
root.something = "value"
print(root.something)
```

```
1. __init__ runs.
2. self.class_name needs to be set.
3. __setattr__ is run to set self.class_name.
4. The key `class_name` is saved into Redis.
5. root.something = "value" sets key `something` into redis too.
6. print(root.something)
7. __getattr__ for `something` is run.
8. in order to get the key from Redis, it needs to get self.class_name.
9. But self.class_name was never set on the object. It was saved into Redis.
10. Tries to get self.class_name which needs self.class_name itself.
```

Solution

- Keep track of native attributes.
- Save to and read from `__dict__`

```python
NATIVE_ATTRIBUTES = {'class_name'}

class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def get_redis_key_path(self, key):
        return "{}.{}".format(self.class_name, key)

    def __getattr__(self, key):
        if key in NATIVE_ATTRIBUTES:
            return self.__dict__[key]
        else:
            key = self.get_redis_key_path(key)
            return redis.get(key)

    def __setattr__(self, key, value):
        if key in NATIVE_ATTRIBUTES:
            self.__dict__[key] = value
        else:
            key = self.get_redis_key_path(key)
            redis.set(key, value)

root = Root()
root.something = 10
print(root.something)  # b"10"
```

Step 3: Make it lazy

```
>>> obj = root.something
>>> obj
<Lazy root.something>
>>> print(root.something)
b"10"
```

OLD

```
NATIVE_ATTRIBUTES = {'class_name'}

class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def get_redis_key_path(self, key):
        return "{}.{}".format(self.class_name, key)

    def __getattr__(self, key):
        if key in NATIVE_ATTRIBUTES:
            return self.__dict__[key]
        else:
            key = self.get_redis_key_path(key)
            return redis.get(key)

    def __setattr__(self, key, value):
        if key in NATIVE_ATTRIBUTES:
            self.__dict__[key] = value
        else:
            key = self.get_redis_key_path(key)
            redis.set(key, value)
```

NEW

```
NATIVE_ATTRIBUTES = {'class_name'}

class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def get_redis_key_path(self, key):
        return "{}.{}".format(self.class_name, key)

    def __getattr__(self, key):
        if key in NATIVE_ATTRIBUTES:
            return self.__dict__[key]
        else:
            key = self.get_redis_key_path(key)
            return Lazy(key)

    def __setattr__(self, key, value):
        if key in NATIVE_ATTRIBUTES:
            self.__dict__[key] = value
        else:
            key = self.get_redis_key_path(key)
            redis.set(key, value)
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value
```

```
>>> root = Root()
>>> root.something = 10

>>> root.something
<Lazy root.something>
>>> print(root.something)  # evaluates
b"10"
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value
```

```python
>>> root = Root()
>>> root.something = 10

>>> root.something > 8
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value
```

```python
>>> root = Root()
>>> root.something = 10

>>> root.something > 8
TypeError: unorderable types: Lazy() > int()
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    def __gt__(self, other):
        return float(self.value) > other
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    def __gt__(self, other):
        return float(self.value) > other
```

```python
>>> root.something > 8
True
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        # Redis keeps strings as bytes
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    def __gt__(self, other):
        return float(self.value) > other

    def __lt__(self, other):
        return float(self.value) < other
```

```
>>> root.something < 11
True
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')

    def __gt__(self, other):
        return float(self.value) > other

    def __lt__(self, other):
        return float(self.value) < other
```

```python
>>> root.something = 10
>>> root.something == 10
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    def __gt__(self, other):
        return float(self.value) > other

    def __lt__(self, other):
        return float(self.value) < other
```

```
>>> root.something = 10
>>> root.something == 10
False
```

```
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')

    def __repr__(self):
        return "<Lazy {}>".format(self.key)

    def __str__(self):
        return self.value

    def __gt__(self, other):
        return float(self.value) > other

    def __lt__(self, other):
        return float(self.value) < other

    def __eq__(self, other):
        return float(self.value) == other
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')
        ...

    def __lt__(self, other):
        return float(self.value) < other

    def __gt__(self, other):
        return float(self.value) > other

    def __eq__(self, other):
        return float(self.value) == other
```

```python
>>> root.something = 10
>>> root.something == 10
True
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')
        ...

    def __lt__(self, other):
        return float(self.value) < other

    def __gt__(self, other):
        return float(self.value) > other

    def __eq__(self, other):
        return float(self.value) == other
```

```python
>>> root.something = 10
>>> root.something == 10
True
>>> root.something is 10
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')
        ...

    def __lt__(self, other):
        return float(self.value) < other

    def __gt__(self, other):
        return float(self.value) > other

    def __eq__(self, other):
        return float(self.value) == other
```

```python
>>> root.something = 10
>>> root.something == 10
True
>>> root.something is 10
False
```

```
>>> root.something.another = 10
```

```
>>> root.something.another = 10
# works!
```

```
>>> root.something.another = 10
# works!
>>> print(root.something.another)
```

```
>>> root.something.another = 10
# works!
>>> print(root.something.another)
AttributeError: 'Lazy' object has no attribute 'another'
```

```
>>> root.something.another = 10
# works!
>>> print(root.something.another)
Solution:
1. root.something is the lazy object
```

```
>>> root.something.another = 10
# works!
>>> print(root.something.another)
Solution:
1. root.something is the lazy object
2. root.something.another should return the same lazy object using __getattr__
```

```
>>> root.something.another = 10
# works!
>>> print(root.something.another)
Solution:
1. root.something is the lazy object
2. root.something return self for any non-native attribute
3. Update self.key from `root.something` to `root.something.another`
```

Step 5: save lists

```
root.sides = ["fries", "salad"]
```

OLD

```
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def get_redis_key_path(self, key):
        return "{}.{}".format(self.class_name, key)

    def __getattr__(self, key):
        if key in NATIVE_ATTRIBUTES:
            return self.__dict__[key]
        else:
            key = self.get_redis_key_path(key)
            return Lazy(key)

    def __setattr__(self, key, value):
        if key in NATIVE_ATTRIBUTES:
            self.__dict__[key] = value
        else:
            key = self.get_redis_key_path(key)
            redis.set(key, value)
```

NEW

```
class Root:
    def __init__(self):
        self.class_name = self.__class__.__name__.lower()

    def get_redis_key_path(self, key):
        return "{}.{}".format(self.class_name, key)

    def __getattr__(self, key):
        if key in NATIVE_ATTRIBUTES:
            return self.__dict__[key]
        else:
            key = self.get_redis_key_path(key)
            return Lazy(key)

    def __setattr__(self, key, value):
        if key in NATIVE_ATTRIBUTES:
            self.__dict__[key] = value
        else:
            key = self.get_redis_key_path(key)
            if isinstance(value, strings):
                redis.set(key, value)
            elif isinstance(value, Iterable):
                redis.delete(key)
                redis.rpush(key, *value)
```

Works!

```
>>> root.sides = ["fries", "salad"]
```

```
$ redis-cli
127.0.0.1:6379> lrange root.sides 0 -1
1) "fries"
2) "salad"
```

Step 5: load lists

```
root.sides = ["fries", "salad"]
print(root.sides)
["fries", "salad"]
```

OLD

```
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        return redis.get(self.key).decode('utf-8')
```

NEW

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result
```

```python
>>> root.sides = ["fries", "salad"]
print(root.sides)
"fries"
```

```
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result
```

```
>>> root.sides = ["fries", "salad"]
print(root.sides[0])
["fries", "salad"]
>>> print(root.sides[0])
```

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result
```

```
>>> root.sides = ["fries", "salad"]
print(root.sides[0])
["fries", "salad"]
>>> print(root.sides[0])
TypeError: 'Lazy' object does not support indexing
```

Magic methods

Protocol for containers (to define containers like lists,…)	Descriptor (custom class attributes that we fully control)	Called when no attribute found
`__getitem__(self, key)`	`__get__(self, obj, cls=None)`	**`__getattr__(self, name)`**
`__setitem__(self, key, value)`	`__set__(self, obj, val)`	
`__delitem__(self, key)`	`__delete__(self, obj)`	

OLD

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result
```

NEW

```python
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result

    def __getitem__(self, key):
        return self.value[key]
```

After

```
class Lazy:
    def __init__(self, key):
        self.key = key

    @property
    def value(self):
        thetype = redis.type(self.key)
        if thetype == b'string':
            result = redis.get(self.key).decode('utf-8')
        elif thetype == b'list':
            result = redis.lrange(self.key, 0, -1)
            result = [i.decode('utf-8') for i in result]
        return result

    def __getitem__(self, key):
        return self.value[key]
```

```
>>> root.sides = ["fries", "salad"]
print(root.sides[0])
["fries", "salad"]
>>> print(root.sides[0])
fries
```

It can get complicated.

RedisWorks

Based on DotObject

https://github.com/seperman/redisworks

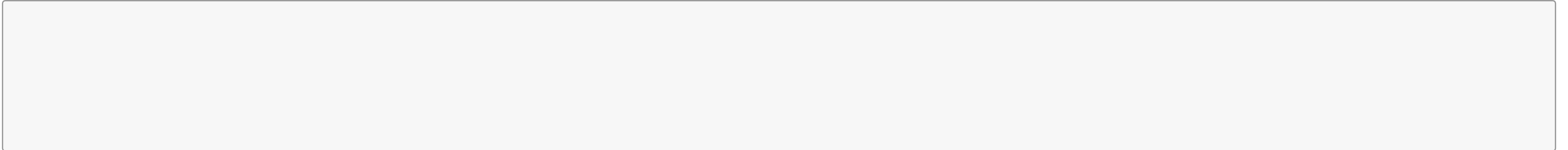
https://github.com/seperman/dotobject

- Lazy Redis Queries
- Multi Query evaluation
- Dynamic Typing
- Ease of use

PyRedis

```
>>> from redis import StrictRedis
>>> redis = StrictRedis()
>>> redis.set("root.something", "value")
```

RedisWorks



PyRedis

```
>>> from redis import StrictRedis
>>> redis = StrictRedis()
>>> redis.set("root.something", "value")
```

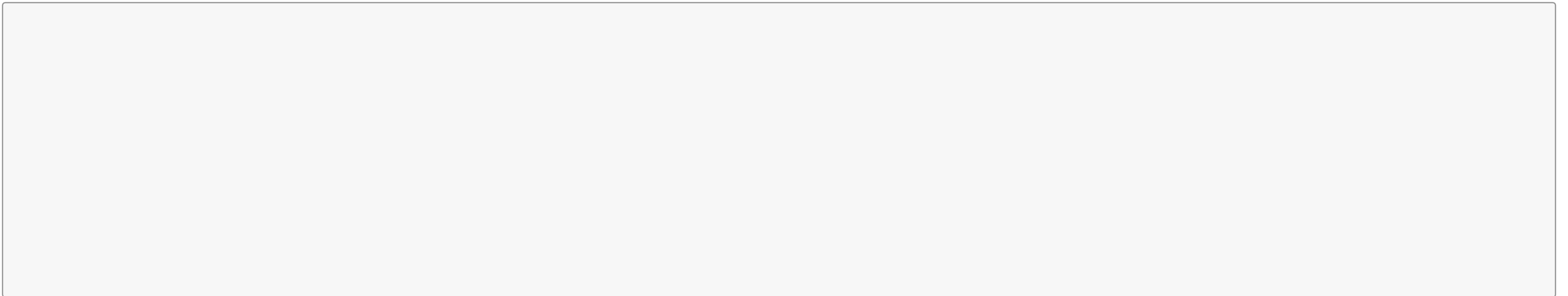
RedisWorks

```
>>> from redisworks import Root
>>> root=Root()
>>> root.something = "value"
```

PyRedis

```
>>> redis.rpush("root.sides", 10, "root.something", "value")
>>> values = redis.lrange("root.sides", 0, -1)
>>> values
[b'10', b'root.something', b'value']
```

RedisWorks



PyRedis

```
>>> redis.rpush("root.sides", 10, "root.something", "value")
>>> values = redis.lrange("root.sides", 0, -1)
>>> values
[b'10', b'root.something', b'value']
```

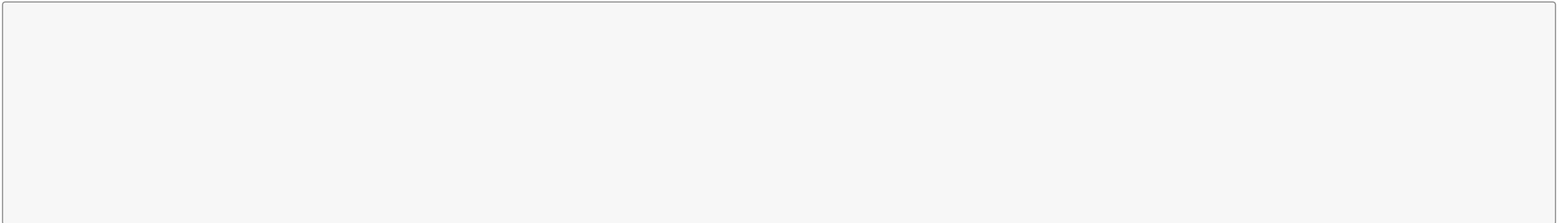
RedisWorks

```
>>> root.sides = [10, "fries", "coke"]
>>> root.sides[1]
'fries'
>>> "fries" in root.sides
True
>>> type(root.sides[0])
int
```

PyRedis

```
>>> values = [10, [1, 2]]
>>> redis.rpush("root.sides", *values)
2
>>> redis.lrange("root.sides", 0, -1)
[b'10', b'[1, 2]']
```

RedisWorks



PyRedis

```
>>> values = [10, [1, 2]]
>>> redis.rpush("root.sides", *values)
2
>>> redis.lrange("root.sides", 0, -1)
[b'10', b'[1, 2]']
```

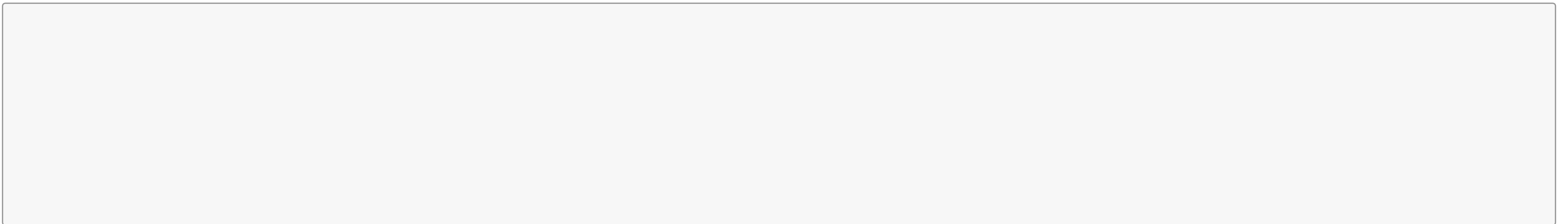
RedisWorks

```
>>> root.sides = [10, [1, 2]]
>>> root.sides
[10, [1, 2]]
>>> type(root.sides[1])
<class 'list'>
```

PyRedis

```
>>> redis.hmset("root.something", {1:"a", "b": {2: 2}})
>>> val = redis.hgetall("root.something")
>>> val
{b'b': b'{2: 2}', b'1': b'a'}
>>> val[b'b']
b'{2: 2}'
```

RedisWorks



PyRedis

```
>>> redis.hmset("root.something", {1:"a", "b": {2: 2}})
>>> val = redis.hgetall("root.something")
>>> val
{b'b': b'{2: 2}', b'1': b'a'}
>>> val[b'b']
b'{2: 2}'
```

RedisWorks

```
>>> root.something = {1:"a", "b": {2: 2}}
>>> root.something
{'b': {2: 2}, 1: 'a'}
>>> root.something['b'][2]
2
```

Harness the power of Python magic methods and lazy objects.

By Sep Dehpour

Aug 2016

zepworks.com

sep at zepworks.com

https://github.com/seperman/redisworks

http://zepworks.com/blog/redisworks-the-pythonic-redis-client/

Thanks to ChowNow for hosting.