capnpy Documentation

Release 0.0

Antonio Cuni

Nov 04, 2019

Contents

`capnpy` is an implementation of Cap'n Proto for Python. Its primary goal is to provide a library which is fast, both on CPython and PyPy, and which offers a pythonic API and feeling whenever possible.

CHAPTER 1

Usage

1.1 Installation and requirements

To install `capnpy`, just type:

```
$ pip install capnpy
```

`capnpy` relies on the official capnproto implementation to parse the schema files, so it needs to be able to find the `capnp` executable to compile a schema. It requires `capnp 0.5.0` or later.

1.2 Quick example

Suppose to have a capnp schema called `example.capnp`:

```
@0xe62e66ea90a396da;
struct Point {
    x @0 :Int64;
    y @1 :Int64;
}
```

You can use `capnpy` to read and write messages of type `Point`:

```
import capnpy
# load the schema using dynamic loading
example = capnpy.load_schema('example')
# create a new Point object
p = example.Point(x=1, y=2)
# serialize the message and load it back
message = p.dumps()
p2 = example.Point.loads(message)
print('p2.x ==', p2.x)
print('p2.y ==', p2.y)
```

```
p2.x == 1
p2.y == 2
```

1.3 Compiling schemas

`capnpy` supports different ways of compiling schemas:

`setuptools` integration to compile and distribute schemas using `setup.py`.

Dynamic loading to compile and load capnproto schemas on the fly.

Manual compilation to compile schemas manually.

If you use `setup.py` or *manual compilation*, you need `capnp` to compile the schema, but not to load it later; this means that you can distribute the precompiled schemas, and the client machines will be able to load it without having to install the official capnproto distribution.

If you use *dynamic loading*, you always need the `capnp` executable whenever you want to load a schema.

1.3.1 Integration with `setuptools`

If you use `setuptools`, you can use the `capnpy_schema` keyword to automatically compile your schemas from `setup.py`:

```
from setuptools import setup
setup(name='foo',
      version='0.1',
      packages=['mypkg'],
      capnpy_schemas=['mypkg/example.capnp'],
      )
```

You can specify additional *compilation options* by using `capnpy_options`:

```
from setuptools import setup
setup(name='foo',
      version='0.1',
      packages=['mypkg'],
      capnpy_options={
          'pyx': False,           # do NOT use Cython (default is 'auto')
          'convert_case': False,  # do NOT convert camelCase to camel_case
                                  # (default is True)
      }
      capnpy_schemas=['mypkg/example.capnp'],
      )
```

1.3.2 Manual compilation

You can manually compile a capnproto schema by using `python -m capnpy compile`:

```
$ python -m capnpy compile example.capnp
```

This will produce `example.py` (if you are using py mode) or `example.so` (if you are using pyx mode). Run `python -m capnpy --help` for additional options.

1.3.3 Dynamic loading

To dynamically load a capnproto schema, use `capnpy.load_schema`; its full signature is:

```
def load_schema(modname=None, importname=None, filename=None,
                pyx='auto', options=None):
    ...
```

`modname`, `importname` and `filename` corresponds to three different ways to specify and locate the schema file to load. You need to pass exactly one of them.

`modname` (the default) is interpreted as if it were the name of a Python module with the `.capnp` extension. This means that it is searched in all the directories listed in `sys.path` and that you can use dotted names to load a schema inside packages or subpackages:

```
>>> import capnpy
>>> import mypackage
>>> mypackage
<module 'mypackage' from '/tmp/mypackage/__init__.pyc'>
>>> example = capnpy.load_schema('mypackage.mysub.example')
>>> example
<module 'example' from '/tmp/mypackage/mysub/example.capnp'>
```

This is handy because it allows you to distribute the capnproto schemas along the Python packages, and to load them with no need to care where they are on the filesystem, as long as the package is importable by Python.

`importname` is similar to `modname`, with the difference that it uses the same syntax you would use in capnproto's *import expressions*. In particular, if you use an absolute path, `load_schema` searches for the file in each of the search path directories, which by default correspond to the ones listed in `sys.path`. Thus, the example above is completely equivalent to this:

```
>>> example = capnpy.load_schema(importname='/mypackage/mysub/example.capnp')
>>> example
<module 'example' from '/tmp/mypackage/mysub/example.capnp'>
```

Finally, `filename` specifies the exact file name of the schema file. No search will be performed.

`pyx` specifies whether to use pyx or py mode. `options` can be used to change the default *compilation options*:

```
>>> from capnpy.annotate import Options
>>> example = capnpy.load_schema('example', options=Options(convert_case=False))
```

1.3.4 Compilation options

The `capnpy` schema compiler has two modes of compilation:

py mode Generate pure Python modules, which can be used either on CPython or PyPy: it is optimized to be super fast on PyPy. It produces slow code on CPython, but it has the advantage of not requiring `cython`. This is the default on PyPy.

pyx mode Generate pyx modules, which are then compiled into native extension modules by `cython` and `gcc`. It is optimized for speed on CPython. This is the default on CPython, if `cython` is available.

Moreover, it supports the following options:

`version_check` If enabled, the compiled schema contains a check which is run at import time to ensure that the current version of capnpy matches to the one we compiled the schema with. See note below for more details. The default is **True**.

`convert_case` If enabled, `capnpy` will automatically convert field names from camelCase to under-score_delimiter: i.e., `fooBar` will become `foo_bar`. The default is **True**.

text_type Can be `bytes` or `unicode`, Determines the default Python type for *Text* fields. The default is `bytes`.

include_reflection_data If enabled, `capnpy` will embed *Reflection data* into the compiled schemas.

Note: **Version checking** is needed in particular if you are using pyx mode, which is the default on CPython. Capnproto `struct` are represented by Python classes which inherits from `capnpy.struct_.Struct`: in pyx mode, this is a Cython `cdef class`, and it has a certain C layout which depends on the number and type of its fields. If the C layout at compilation and import time don't match, you risk segfault and/or misbehavior. Since the internal layout of classes might change between capnpy version, the version check prevents this risk.

1.3.5 Options annotation

`capnpy` options can also be configured by using the `$Py.options` annotation, which can be applied to `file`, `struct` and `field` nodes. The annotation recurively applies also to all the children nodes and can be used to override the options used by the parents.

This can be used to have a more granular control on how certain capnproto types are translated into Python. For example, you could use it to apply the `convert_case` option only to certain structs or fields:

```
@0x97a960ad8d4cf616;
using Py = import "/capnpy/annotate.capnp";

# don't convert the case by default
$Py.options(convertCase=false);

struct A {
    fieldOne @0 :Int64;
}

struct B $Py.options(convertCase=true) {
    fieldOne @0 :Int64;
    fieldTwo @1 :Int64;
    fieldThree @2 :Int64 $Py.options(convertCase=false);
}
```

```
>>> mod = capnpy.load_schema('example_options')
>>> mod.A.fieldOne
<property object at ...>
>>> mod.B.field_one
<property object at ...>
>>> mod.B.field_two
<property object at ...>
>>> mod.B.fieldThree
<property object at ...>
```

In the example above, `A.fieldOne` is not converted because of the file-level annotation. `B.field_one` and `B.field_two` are converted because the annotation on the struct overrides it. Finally, `B.fieldThree` overrides it again.

Note: Note the different spelling of options names: when you specify them in `setup.py`, they follow Python's `naming_convention` and thus are spelled e.g. `convert_case` and `text_type`. However, when you specify them as annotation, the capnproto schema language mandates `camelCase`.

1.4 Loading and dumping messages

The API to read and write capnproto messages is inspired by the ones offered by `pickle` and `json`:

- `capnpy.load(f,payload_type)`: load a message from a file-like object
- `capnpy.loads(s,payload)`: load a message from a string
- `capnpy.load_all(f,payload_type)`: return a generator which yields all the messages from the given file-like object
- `capnpy.dump(obj)`: write a message to a file-like object
- `capnpy.dumps(obj)`: write a message to a string

For example:

```
>>> import capnpy
>>> example = capnpy.load_schema('example')
>>> p = example.Point(x=100, y=200)
>>> mybuf = capnpy.dumps(p)
>>> mybuf

↪'\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00d\x00\x00\x00\x00\x00\x00\x00\xc8'
↪'
>>> p2 = capnpy.loads(mybuf, example.Point)
>>> print(p2.x, p2.y)
100 200
```

Alternatively, you can call `load`/`loads` directly on the class, and `dump`/`dumps` directly on the objects:

```
>>> p = example.Point(x=100, y=200)
>>> mybuf = p.dumps()
>>> p2 = example.Point.loads(mybuf)
>>> print(p2.x, p2.y)
100 200
```

By default, `dump` and `dumps` try to use a fast path which is faster if you pass an object which is *compact*. If the fast path can be taken, it is approximately 5x faster on CPython and 10x faster on PyPy. However, if the object is **not** compact, the fast path check makes it ~2x slower. If you are sure that the object is not compact, you can disable the check by passing `fastpath=False`:

```
>>> mybuf = p.dumps(fastpath=False)
```

1.5 Loading from sockets

In case you want to load your messages from a `socket`, you can use `capnpy.buffered.BufferedSocket` to wrap it into a file-like object:

```
>>> from capnpy.buffered import BufferedSocket
>>> sock = socket.create_connection(('localhost', 5000))
>>> buf = BufferedSocket(sock)
>>> example.Point.load(buf)
```

Warning: The obvious solution to wrap a socket into a file-like object would be to use `socket.makefile()`. However, because of this bug it is horribly slow. **Don't use it**. See also the *benchmarks*.

1.6 Raw dumps

Raw dumps are intented primarly for debugging and should **never** be used as a general transmission mechanism. They dump the internal state of the segments and the offsets used to identify a given capnproto object.

In particular, they dump the whole buffer in which the object is contained, which might be much larger that the object itself.

If you encounter a canpy bug, you can use `_raw_dumps` and `_raw_loads` to save the offending object to make it easier to reproduce the bug:

```
>>> p = example.Point(x=100, y=200)
>>> mydump = p._raw_dumps()
>>> p2 = example.Point._raw_loads(mydump)
>>> print(p2.x, p2.y)
100 200
```

1.7 capnproto types

1.7.1 Text

Capnproto defines `Text` fields as "always UTF-8 encoded and NUL-terminated". There are at least two reasonable ways to represent this in Python:

- as `bytes`: this will contain the undecoded UTF-8 string.
- as `unicode`: this will automatically do `.decode('utf-8')` for you. However, it is potentially less efficient because capnpy needs to re-decode the string again and again any time you read the field.

By default, `Text` fields are represented as `bytes`. You can change the default behavior by setting the appropriate *Compilation options*. In case you are using *Integration with setuptools*, you need to pass `capnpy_options={'text_type': 'unicode'}` in your `setup.py`.

If you want more granular control, you can annotate single files/struct/fields by using the *Options annotation*.

1.7.2 Struct

`capnpy` turns each capnproto struct into a Python class. The API is inspired by `namedtuples`:

- the fields of the struct are exposed as plain attributes
- objects are **immutable**; it is not possible to change the value of a field once the object has been instantiated. If you need to change the value of a field, you can instantiate a new object, as you would do with namedtuples
- objects can be made *comparable and hashable* by specifying the `$Py.key` annotation

Moreover, in case the type of a field is a pointer (e.g. `Text`, `Data`, structs and lists), `capnpy` generates two different accessors. For a field named `foo`:

- `has_foo()`: return `True` if `foo` is not `NULL`, `False` otherwise

- `get_foo()`: if `has_foo()` is `True`, it is equivalent to `foo`. Else, it returns the default value for that field

Note that in case of a `struct` field, the default value is a struct whose fields have all the default value, recursively:

```
@0xe62e66ea90a396da;
struct Point {
    x @0 :Int64;
    y @1 :Int64;
    name @2 :Text;
}

struct Rectangle {
    a @0 :Point;
    b @1 :Point;
}
```

```
>>> mod = capnpy.load_schema('example_struct')
>>> p = mod.Point()
>>> p
<Point: (x = 0, y = 0)>
>>> print(p.name)
None
>>> p.has_name()
False
>>> p.get_name()
''
>>> rect = mod.Rectangle()
>>> print(rect.a)
None
>>> print(rect.has_a())
False
>>> print(rect.get_a())
<Point: (x = 0, y = 0)>
>>> rect.get_a().get_name()
''
```

The rationale is that `get_foo()` and `has_foo()` are modeled after the semantics of the original C++ implementation of capnproto, while `.foo` is modeled after the Pythonic `namedtuple` API. In particular `.foo` returns `None` instead of the default value to avoid unpythonic and surprising cases such as `Point(name=None).name == ''`

1.7.3 Enum

capnproto enums are represented as subclasses of `int`, so that we can easily use both the numeric and the symbolic values:

```
@0x8eecd1de76ded4c4;
enum Color {
    red @0;
    green @1;
    blue @2;
    yellow @3;
}
```

```
>>> mod = capnpy.load_schema('example_enum')
>>> Color = mod.Color
>>> Color.green
<Color.green: 1>
```

```
>>> int(Color.green)
1
>>> str(Color.green)
'green'
>>> Color.green + 2
3
>>> Color(2)
<Color.blue: 2>
>>> Color.__members__
('red', 'green', 'blue', 'yellow')
```

1.7.4 Union

capnproto uses a special enum value, called *tag*, to identify the field which is currently set inside an union; `capnpy` follows this semantics by automatically creating an *Enum* whose members correspond to fields of the union.

```
@0x8ced518a09aa7ce3;
struct Shape {
  area @0 :Float64;

  union {
    circle @1 :Float64;       # radius
    square @2 :Float64;       # width
  }
}

struct Type {
  union {
    void @0 :Void;
    bool @1 :Void;
    int64 @2 :Void;
    float64 @3 :Void;
    text @4 :Void;
  }
}
```

```
>>> mod = capnpy.load_schema('example_union')
>>> Shape, Type = mod.Shape, mod.Type
>>> Shape.__tag__
<class 'example_union.Shape__tag__'>
>>> Shape.__tag__.__members__
('circle', 'square')
>>> Type.__tag__.__members__
('void', 'bool', 'int64', 'float64', 'text')
```

You can query which field is set by calling `which()`, or by calling one of the `is_*()` methods which are automatically generated:

```
>>> s = Shape(area=16, square=4)
>>> s.which()
<Shape__tag__.square: 1>
>>> s.__which__()
1
>>> s.is_circle()
False
```

```
>>> s.is_square()
True
```

The difference between `which()` and `__which__()` is that the former return an `Enum` value, while the latter a raw integer: on CPython, `which()` is approximately *4x slower*, so you might consider to use the raw form in performance-critical parts of your code. On PyPy, the two forms have the very same performance.

Since `capnpy` objects are immutable, union fields must be set when instantiating the object. The first way is to call the default constructor and set the field as usual:

```
>>> s = Shape(area=3*3*math.pi, circle=3)
>>> s.is_circle()
True
```

If you try to specify two conflicting fields, you get an error:

```
>>> Shape(area=16, square=4, circle=42)
Traceback (most recent call last):
  ...
TypeError: got multiple values for the union tag: circle, square
```

The second way is to use one of the special `new_*()` alternate constructors:

```
>>> s = Shape.new_square(area=16, square=4)
>>> s.is_square()
True
```

```
>>> s = Shape.new_square(area=16, square=4, circle=42)
Traceback (most recent call last):
  ...
TypeError: new_square() got an unexpected keyword argument 'circle'
```

The alternate constructors are especially handy in case of `Void` union fields, because in that case you don't need to specify the (void) value of the field:

```
>>> t = Type.new_int64()
>>> t.which()
<Type__tag__.int64: 2>
>>> t.is_int64()
True
```

1.7.5 Groups

```
@0x97a960ad8d4cf616;
struct Point {
    position :group {
        x @0 :Int64;
        y @1 :Int64;
    }
    color @2 :Text;
}
```

Group fields are initialized using a tuple, and accessed using the usual dot notation:

```
>>> mod = capnpy.load_schema('example_group')
>>> Point = mod.Point
>>> p = Point(position=(3, 4), color='red')
>>> p.position.x
3
>>> p.position.y
4
```

capnpy also generates a **group constructor**, which is a `staticmethod` named as the capitalized group name. It is useful because you can use keyword arguments and get the desired tuple in the right order:

```
>>> Point.Position(y=6, x=5)
(5, 6)
>>> p2 = Point(position=Point.Position(x=5, y=6), color='red')
>>> p2.position.x
5
>>> p2.position.y
6
```

By using the group constructor, you can also **omit** some parameters; in this case, they will get the default value, as usual:

```
>>> Point.Position(y=7)
(0, 7)
```

Note: Make sure to notice the difference between the lowercase `Point.position` which is a property used to read the field, and the capitalized `Point.Position` which is the group constructor:

```
>>> Point.position
<property object at ...>
>>> Point.Position
<function Position at ...>
```

1.7.6 Virtual groups

You can use the `$Py.group` annotation on a `Void` field to generate a virtual group, which fishes the data from normal "flat" fields.

```
@0x97a960ad8d4cf616;
using Py = import "/capnpy/annotate.capnp";

struct Point {
    x @0 :Int64;
    y @1 :Int64;
    color @2 :Text;
    position @3 :Void $Py.group("x, y") $Py.key("*");
}
```

This becomes particularly handy in conjuction with `$Py.key` (see *Equality and hashing*), because it allows to get an hashable/comparable subset of the fields without affecting other parts of the code which want to access the flat fields:

```
>>> mod = capnpy.load_schema('example_py_group')
>>> p = mod.Point(x=1, y=2, color='red')
```

```
>>> p.x
1
>>> p.position.x
1
>>> p.position == (1, 2)
True
```

1.7.7 Named unions

Named unions are a special case of groups.

```
@0xe1f94ddddf8858c4;
struct Person {
  name @0 :Text;
  job :union {
      unemployed @1 :Void;
      employer @2 :Text; # this is the company name
      selfEmployed @3 :Void;
  }
}
```

You can instantiate new objects as you would do with a normal group, by using the group constructor. If you want to specify a `Void` union field, you can use `None`:

```
>>> mod = capnpy.load_schema('example_named_union')
>>> Person = mod.Person
>>> p1 = Person(name='Alice', job=Person.Job(unemployed=None))
>>> p2 = Person(name='Bob', job=Person.Job(employer='Capnpy corporation'))
```

Reading named unions is the same as anonymous ones:

```
>>> p1.job.which()
<Person_job__tag__.unemployed: 0>
>>> p1.job.is_unemployed()
True
>>> p2.job.employer
'Capnpy corporation'
```

Note: The reason why you have to use the group constructor is that it automatically insert the special `undefined` value in the right positions:

```
>>> from capnpy.struct_ import undefined
>>> undefined
<undefined>
>>> Person.Job(unemployed=None)
(None, <undefined>, <undefined>)
>>> Person.Job(employer='Capnpy corporation')
(<undefined>, 'Capnpy corporation', <undefined>)
```

1.8 "Compact" structs

A struct object is said to be "compact" if:

1. there is no gap between the data and pointers sections
2. there is no gap between the children
3. the pointers to the children are ordered
4. the children are recursively compact

The compactness of a message depends on the implementation which generates it. The most natural way to generate Cap'n Proto messages is to write them in pre-order (i.e., you write first the root, then its children in order, recursively). If the messages are generated this way and without introducing gaps, it is automatically compact.

Messages created by `capnpy` are always compact.

You can check for compactness by calling the `_is_compact` method:

```
>>> mod = capnpy.load_schema('example_compact')
>>> p = mod.Point(1, 2)
>>> p._is_compact()
True
```

1.8.1 List items

Cap'n Proto lists are implemented in such a way that items are placed one next to the other, and the children of the items are placed at the end of the list body. This means that, if the items have children, surely there will be a gap.

Hence, as soon as you have a Cap'n Proto list whose items have pointers, the items are **not** compact, even if the list as a whole is.

```
>>> mod = capnpy.load_schema('example_compact')
>>> p0 = mod.Point(1, 2, name='p0')
>>> p1 = mod.Point(3, 4, name='p1')
>>> poly = mod.Polygon(points=[p0, p1])
>>> poly._is_compact()
True
>>> poly.points[0]._is_compact()
False
```

1.8.2 The `compact()` method

Cap'n Proto message can be arbitrarly large and occupy a big amount of memory; moreover, when you access a struct field or a list item, the resulting object keeps alive the whole message.

However, sometimes you are interested in keeping alive only a smaller part it: you can accomplish this by calling the `compact()` method, which creates a new, smaller message containing only the desired subset. Also, as the name suggests, the newly created message is guaranteed to be compact:

```
>>> mod = capnpy.load_schema('example_compact')
>>> poly = mod.Polygon([mod.Point(1, 2, 'p0'), mod.Point(3, 4, 'p1')])
>>> len(poly._seg.buf)
80
>>> p0 = poly.points[0]
>>> len(p0._seg.buf)    # p0 keeps the whole segment alive
80
>>> p0._is_compact()
False
>>> pnew = p0.compact()
```

```
>>> len(pnew._seg.buf) # pnew keeps only a subset alive
40
>>> pnew._is_compact()
True
```

1.9 Equality and hashing

By default, structs are not hashable and cannot be compared:

```
>>> p1 = example.Point(x=1, y=2)
>>> p2 = example.Point(x=1, y=2)
>>> p1 == p2
Traceback (most recent call last):
  ...
TypeError: Cannot hash or compare capnpy structs. Use the $Py.key annotation to␣
↪enable it
```

By specifying the `$Py.key` annotation, you explicitly tell `capnpy` which fields to consider when doing equality testing and hashing:

```
@0xaff59c0b39ac4242;
using Py = import "/capnpy/annotate.capnp";

# the name will be ignored in comparisons, as it is NOT in the key
struct Point $Py.key("x, y") {
    x @0 :Int64;
    y @1 :Int64;
    name @2 :Text;
}
```

```
>>> mod = capnpy.load_schema('example_key')
>>> Point = mod.Point
>>> p1 = Point(1, 2, "p1")
>>> p2 = Point(1, 2, "p2")
>>> p3 = Point(3, 4, "p3")
>>>
>>> p1 == p2
True
>>> p1 == p3
False
```

You can also use them as dictionary keys:

```
>>> d = {}
>>> d[p1] = 'hello'
>>> d[p2]
'hello'
```

Tip: If you have many fields, you can use `$Py.key("*")` to include all of them in the comparison key: this is equivalent of explicitly listing all the fields which are present in the schema **at the moment of compilation**. In particular, be aware that if later get objects which come from a *newer* schema, the additional fields will **not** be considered in the comparisons.

Moreover, the structs are guaranteed to hash and compare equal to the corresponding tuples:

```
>>> p1 == (1, 2)
True
>>> p3 == (3, 4)
True
>>> d[(1, 2)]
'hello'
```

1.9.1 Rationale

Why are not structs comparable by defaults but you have to manually specify `$Py.key`? Couldn't `capnpy` be smart enough to figure out by itself?

We choose to use `$Py.key` because it is not obvious what is the right thing to do in presence of schema evolution. For example, suppose you start with previous version of `struct Point` which contains only `x` and `y`:

```
struct OlderPoint {
    x @0 :Int64;
    y @1 :Int64;
}
```

```
>>> OlderPoint = mod.OlderPoint
>>> p1 = OlderPoint(1, 2) # there is no "name" yet
```

Then, you receive some other object created with a newer schema which contains an additional field, such as our `Point`. Since `Point` is an evolution of `OlderPoint`, it is perfectly legit to load it:

```
>>> p_with_name = Point(1, 2, 'this is my name')
>>> message_from_the_future = p_with_name.dumps()
>>> p2 = OlderPoint.loads(message_from_the_future)
>>> p2.x, p2.y
(1, 2)
```

Now, note that the underyling data contains the name, although we don't have the `name` field (because we are using an older schema):

```
>>> hasattr(p2, 'name')
False
>>> 'this is my name' in p2._seg.buf
True
```

So, what should `p1 == p2` return? We might choose to simply ignore the `name` and return `True`. Or choose to consider `p1.name` equal to the empty string, or to `None`, and thus return `False`. Or we could declare that two objects are equal when their canonical representation is the same, which introduces even more subtle consequences.

According to the Zen of Python:

> *Explicit is better than implicit.*
> *In the face of ambiguity, refuse the temptation to guess.*

Hence, we require you to explicity specify which fields to consider.

1.10 Extending capnpy structs

As described above, each capnproto `struct` is converted into a Python class. With `capnpy` you can easily add methods by using the `__extend__` class decorator:

```
>>> import math
>>> import capnpy
>>> Point = example.Point
>>>
>>> @Point.__extend__
... class Point:
...     def distance(self):
...         return math.sqrt(self.x**2 + self.y**2)
...
>>>
>>> p = Point(x=3, y=4)
>>> p.distance()
5.0
```

Although it seems magical, `__extend__` is much simpler than it looks: what it does is simply to copy the content of the new class body `Point` into the body of the automatically-generated `example.Point`; the result is that `example.Point` contains both the original fields and the new methods.

When loading a schema, e.g. `example.capnp`, `capnpy` also searches for a file named `example_extended.py` in the same directory. If it exists, the code is executed in the same namespace as the schema being loaded, meaning that it is the perfect place where to put the `__extend__` code to be sure that it will be immediately available. For example, suppose to have the following `example_extended.py` in the same directory as `example.capnp`:

```
# example_extended.py
import math

# Note that the Point class is already available, as this code is executed
# inside the namespace of the module loaded from example.capnp
@Point.__extend__
class Point:
    def distance(self):
        return math.sqrt(self.x**2 + self.y**2)
```

Then, the `distance` method will be immediately available as soon as we load the schema:

```
>>> import capnpy
>>> example = capnpy.load_schema('example')
>>> p = example.Point(3, 4)
>>> print(p.distance())
5.0
```

1.11 Reflection API

Using the reflection API, it is possible to programmatically query information about a schema, for example what are the fields inside a struct.

The main entry point is the function `capnpy.get_reflection_data()`, which returns the metadata for a given module as an instance of `ReflectionData`.

```
>>> mod = capnpy.load_schema('example')
>>> reflection = capnpy.get_reflection_data(mod)
```

Under the hood, the `capnp` compiler produces a capnproto representation of the parsed schema, where most capnproto entities are represented by nodes. You can use `get_node` to get the capnproto node corresponding to a given Python-level entity:

```
>>> # get the node for the Point struct
>>> node = reflection.get_node(mod.Point)
>>> type(node)
<class 'capnpy.schema.Node__Struct'>
>>> node.displayName[-19:]
'example.capnp:Point'
>>> node.which()
<Node__tag__.struct: 1>
>>> node.is_struct()
True
>>> for f in node.struct.fields:
...     print(f)
...
<Field 'x': int64>
<Field 'y': int64>
```

Note: By default, reflection data is included into all compiled schemas. You can change the behavior by setting the *option* `include_reflection_data` to `False`.

1.11.1 Nodes vs Python entities

When compiling a schema `capnpy` generates Python entities from nodes: for example, `Struct` are compiled as Python classes, and fields as Python properties. Although closely related, they are not always equivalent: for example, `Field.name` is always `camelCase`, but the Python property might be called differently, depending on the *compilation options*.

For example, consider the following schema:

```
@0xe62e66ea90a396da;
struct Foo {
    myField @0 :Int64;
}

enum Color {
    lightRed @0;
    darkGreen @1;
}
```

To get the correct Python-level name, you can call `reflection.field_name()`:

```
>>> mod = capnpy.load_schema('example_reflection')
>>> reflection = capnpy.get_reflection_data(mod)
>>> node = reflection.get_node(mod.Foo)
>>> f = node.get_struct_fields()[0]
>>> f
<Field 'myField': int64>
```

```
>>> reflection.field_name(f)
'my_field'
```

This works also for enums:

```
>>> node = reflection.get_node(mod.Color)
>>> node.is_enum()
True
>>> enumerants = node.get_enum_enumerants()
>>> enumerants[0].name
'lightRed'
>>> reflection.field_name(enumerants[0])
'light_red'
```

1.11.2 Inspecting annotations

The Reflection API provides methods to inspect capnproto annotations. Consider the following schema, in which we use custom annotations to map structs to database tables:

```
@0x801e5c7f340eaf8f;

annotation dbTable(struct) :Text;
annotation dbPrimaryKey(field) :Void;

struct Person $dbTable("Persons") {
    id @0 :UInt64 $dbPrimaryKey;
    firstName @1 :Text;
    lastName @2 :Text;
    school @3 :UInt64;
}

struct School $dbTable("Schools") {
    id @0 :UInt64 $dbPrimaryKey;
    name @1 :Text;
    city @2 :Text;
}
```

You can use `has_annotation()` and `get_annotation()` to query about them:

```
>>> mod = capnpy.load_schema('example_reflection_db')
>>> reflection = capnpy.get_reflection_data(mod)
>>> reflection.has_annotation(mod.Person, mod.dbTable)
True
>>> reflection.get_annotation(mod.Person, mod.dbTable)
'Persons'
```

The following shows a complete example of how to use annotations to create a simple dump of the DB structure. It is also worth noticing the usage of `reflection.field_name()` to convert from e.g. `firstName` to `first_name`:

```
>>> def print_table(node):
...     table = reflection.get_annotation(node, mod.dbTable)
...     print('DB Table:', table)
...     for f in node.get_struct_fields():
...         print('   ', reflection.field_name(f), end='')
...         if reflection.has_annotation(f, mod.dbPrimaryKey):
```

```
...             print(' PRIMARY KEY', end='')
...         print()
>>>
>>> for node in reflection.allnodes.values():
...     if reflection.has_annotation(node, mod.dbTable):
...         print_table(node)
...
DB Table: Persons
    id PRIMARY KEY
    first_name
    last_name
    school
DB Table: Schools
    id PRIMARY KEY
    name
    city
```

1.12 capnpy vs pycapnp

To be written

CHAPTER 2

Changelog

2.1 0.8.1

- Fix the Reflection API in presence of large schemas, which `capnp` compiles using multiple segments and far pointers.

2.2 0.8.0

- Improve the `shortrepr()` method and consequently the `__repr__` of capnpy structs: the goal is to make the output of shortrepr() fully compatible with the standard `capnp encode` tool: this way it is possible to reconstruct the original binary message from a capnpy textual dump.
- Fix a corner case when reading far pointers: this bug prevented capnpy to parse large schemas under some conditions.
- Add a new compilation option to control whether to include the Reflection data: see *Compilation options*.
- Improve support for `const` inside capnproto schemas: it is now possible to declare struct and list constants.

2.3 0.7.0

- Add the *Reflection API*, which makes it possible to programmatically query information about a schema, for example what are the fields inside a struct.

2.4 0.6.4

- Fix `$Py.groups` collisions (PR #45).

2.5 0.6.3

- Fix the repr text fields when `textType=unicode`.

2.6 0.6.2

- Don't crash if we can't determine the version of `capnp` (PR #43).

2.7 0.6.1

- Improve `load()` and `load_all()`. Try harder to distinguish between a clean close of the connection and an unclean one: now we raise EOFError *only* if we read an empty string at the very beginning of the message.
- Fix constructors when using a `$Py.nullable` on a group value.

2.8 0.6

- Add the new `text_type` option (see *Compilation options*). It is now possible to choose whether `Text` fields are represented as bytes or unicode.

CHAPTER 3

Benchmarks

Every time we push new code to github, our Continuous Integration System re-runs all the benchmarks and regenerates these charts.

This section shows the current benchmark results and compares `capnpy` to various alternative implementations. *Evolution over time* shows how `capnpy` performance has evolved.

3.1 How to read the charts

For each benchmark we show two charts, one for CPython and one for PyPy. Make sure to notice the different scale on the Y axis: PyPy is often an order of magnitue faster than CPython, so it does not make sense to directly compare them, but inside each chart it is useful to compare the performance of `capnpy` to the other reference points.

Moreover, all benchmarks are written so that they repeat the same operation for a certain number of iteration inside a loop. The charts show the total time spent into the loop, not the time per iteration. Again, it is most useful to just compare `capnpy` to the other reference points.

Most benchmarks compare the performance of `capnpy` objects against alternative implementations. In particular:

instance objects are instances of plain Python classes. This is an useful reference point because often it represents the best we can potentially do. The goal of `capnpy` is to be as close as possible to instances.

namedtuple same as above, but using `collections.namedtuple` instead of Python classes.

pycapnp the default Cap'n Proto implementation for Python. It does not work on PyPy.

3.2 Get Attribute

This benchmark measures how fast is to read an attribute out of an object, for different types of attribute.

The benchmarks for `group`, `struct` and `list` are expected to take a bit longer than the others, because after getting the attribute, they "do something" with the result, i.e. reading another attribute in case of `group` and `struct`, and getting the length of a `list`.

The PyPy charts shows that `uint64` fields are much slower than the others: this is because the benchmarks are run on PyPy 5.4, which misses an optimization in that area. With PyPy 5.6, `uint64` is as fast as `int64`.

3.3 Special union attributes

If you have an *Union*, you can inspect its tag value by calling `which()`, `__which__()` or one of the `is_*()` methods. Ultimately, all of them boil down to reading an `int16` field, so the corresponding benchmark is included as a reference.

Note that on CPython, `which()` is slower than `__which__()`: this is because the former returns an *Enum*, while the latter returns a raw integer. On the other hand, PyPy is correctly able to optimize away all the abstraction overhead.

3.4 Lists

These benchmark measure the time taken to perform various operations on lists. The difference with the `list` benchmark of the previous section is that here we do not take into account the time taken to **read** the list itself out of its containing struct, but only the time taken to perform the operations after we got it.

The `iter` benchmark iterates over a list of 4 elements.

3.5 Hashing

If you use `$Py.key` (see *Equality and hashing*), you can `hash` your objects, and the return value is guaranteed to be the same as the corresponding tuple.

The simplest implementation would be to create the tuple call `hash()` on it. However, `capnpy` uses an ad-hoc implementation so that it can compute the hash value **without** creating the tuple. This is especially useful if you have `text` fields, as you completely avoid the expensive creation of the string.

3.6 Constructors

This benchmark measure the time needed to create new objects. Because of the Cap'n Proto specs, this **has** to be more expensive than creating e.g. a new instance, as we need to do extra checks and pack all the objects inside a buffer. However, as the following charts show, creating new `capnpy` objects is almost as fast as creating instances. As shown by the charts, the performances are different depending on the type of the fields of the target struct.

List fields are special: normally, if you pass a list object to an instance or namedtuple, you store only a reference to it. However, if you need to construct a new Cap'n Proto object, you need to copy the whole content of the list into the new buffer. In particular, if it is a list of structs, you need to deeep-copy each item of the list, separately. This explains why `test_list` looks slower than the rest.

3.7 Deep copy

Sometimes we need to perform a deep-copy of a Cap'n Proto object. In particular, this is needed:

- if you construct a new object having a struct field
- if you construct a new object having a list of structs field
- if you `dump()` an object which is not "compact"

capnpy includes a generic, schema-less implementation which can recursively copy an arbitrary Capn'n Proto pointer into a new buffer. It is written in pure Python but compiled with Cython, and heavily optimized for speed. PyCapnp relies on the official capnproto implementation written in C++.

The copy_pointer benchmarks repeatedly copies a big recursive tree so that the majority of the time is spent inside the deep-copy function and we can ignore the small amout of time spent outside. Thus, we are effetively benchmarking our Cython-based function against the heavily optimized C++ one. The resulting speed is very good. On some machine, it has measured to be even **faster** than the C++ version.

3.8 Loading messages

These benchmark measure the performance of reading a stream of Cap'n Proto messages, either from a file or from a TCP socket.

Note: pycapnp delegates the reading to the underlying C++ library, so you need to pass anything with a fileno() method: so, we pass a socket object directly. On the other hand, capnpy needs a file-like object, so we pass a BufferedSocket.

3.9 Buffered streams

As explained in the section *Loading from sockets*, capnpy provides its own buffered wrapper around socket, which is immensely faster than socket.makefile().

3.10 Dumping messages

These benchmark measure the performance of dumping an existing capnpy object into a message to be sent over the wire. At mimimum, to dump a message you need to copy all the bytes which belongs to the object: this is measured by test_copy_buffer, which blindly copy the entire buffer and it is used as a baseline.

The actual implementation of dumps() needs to do more: in particular, it needs to compute the exact range of bytes to copy. Thus, the goal is that dumps() should be as close as possible to copy_buffer.

If the structure was inside a capnpy list, it will be "non compact": in other words, it is not represented by a contiguous amount of bytes in memory. In that case, dumps() needs to do even more work to produce the message. At the moment of writing, the implementation of .compact() is known to be slow and non-optimized.

3.11 Evolution over time