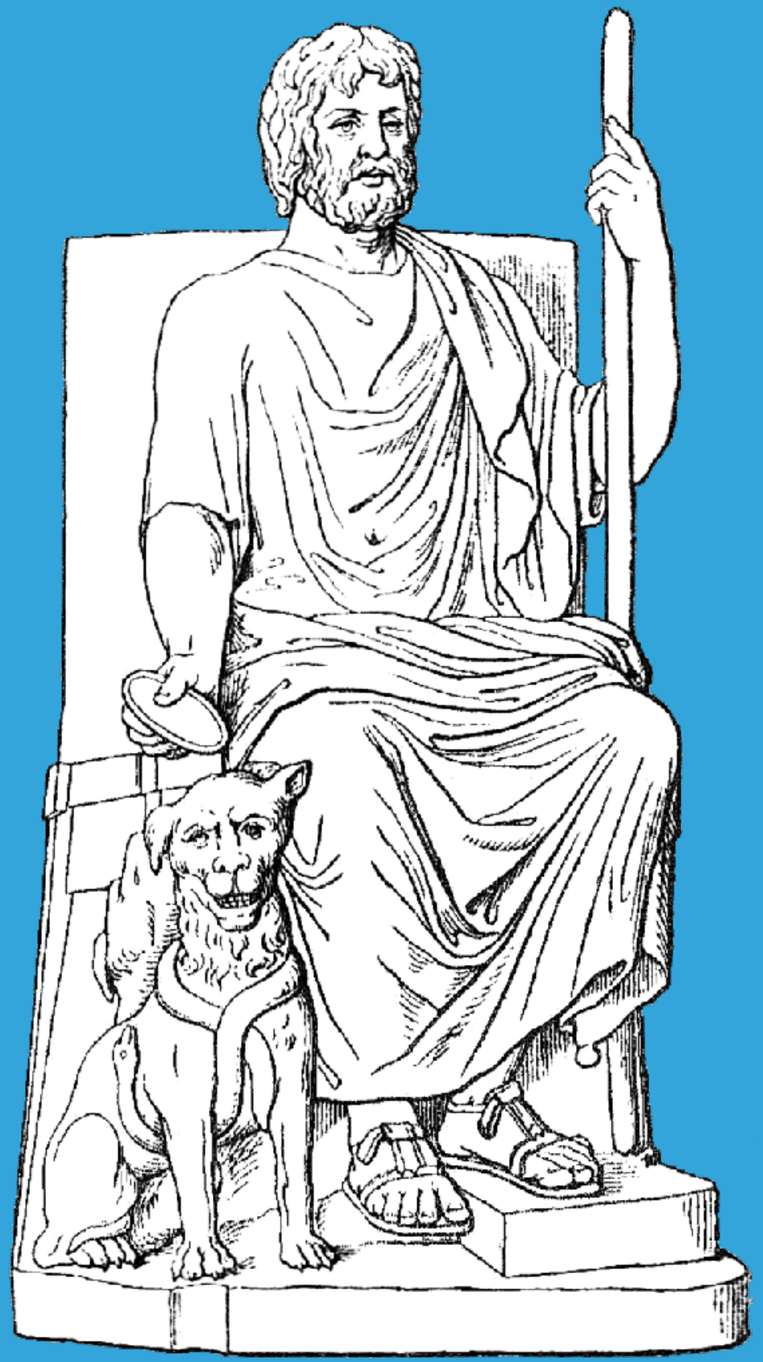
DATA VALIDATION FOR HUMANS

# CERBERUS

# NICOLA IAROCCI

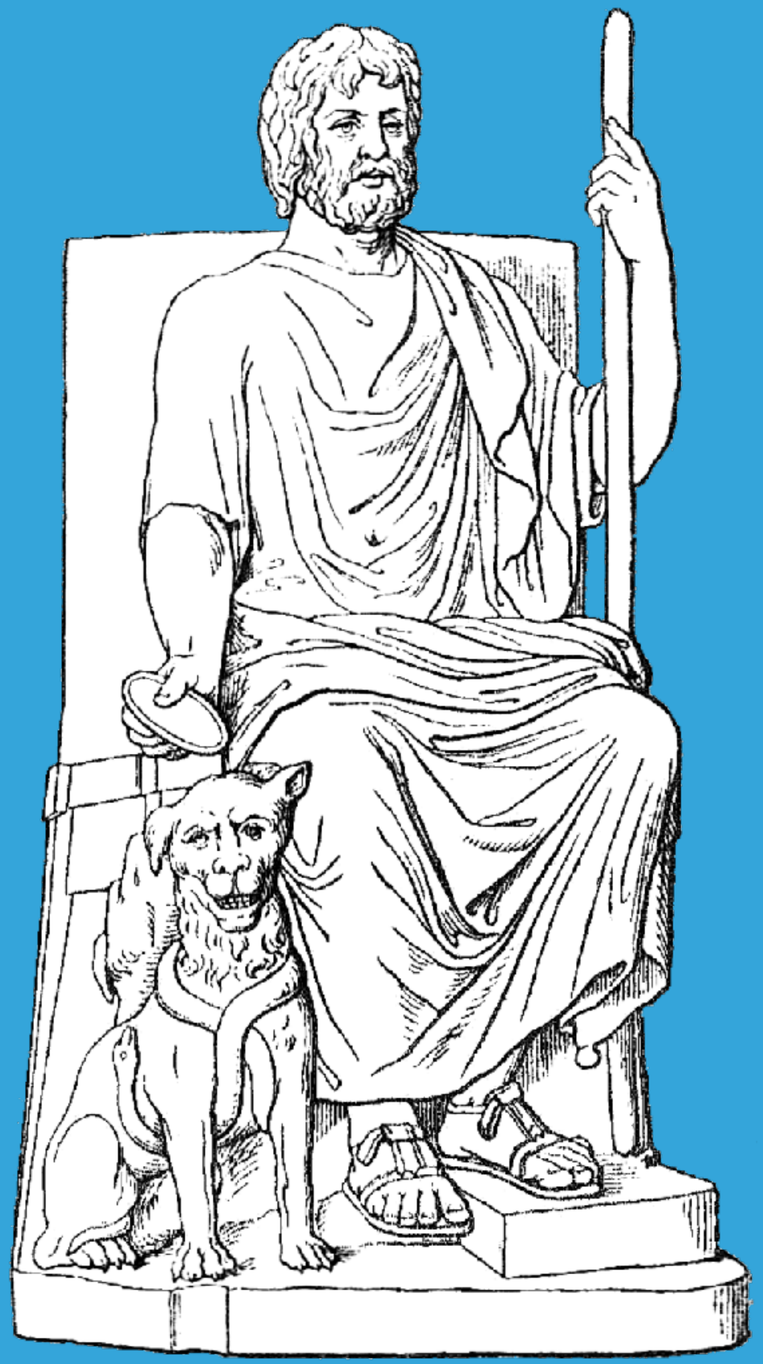
CO-FOUNDER @CIR2K
LEAD DEV @AMICA
MICROSOFT MVP
MONGODB MASTER
CODERDOJO
OPEN SOURCE

TL;DR CODER AT HEART



# CERBERUS IS A LIGHTWEIGHT AND EXTENSIBLE DATA VALIDATION LIBRARY FOR PYTHON

**python-cerberus.org**



# BSD LICENSED
# NO DEPENDENCIES
# PYTHON 2.6+
# PYTHON 3.3+
# PYPY 3

**github.com/pyeve/cerberus**

CERBERUS, NOUN. THE WATCH-DOG OF HADES, WHOSE DUTY IT WAS TO GUARD THE ENTRANCE. EVERYBODY, SOONER OR LATER, HAD TO GO THERE, AND NOBODY WANTED TO CARRY OFF THE ENTRANCE.

Ambrose Bierce

The Devil's Dictionary

CERBERUS

# AT A GLANCE

# 1. DEFINE A SCHEMA AND PASS IT TO A VALIDATOR INSTANCE

```
>>> from cerberus import Validator

>>> schema = {'name': {'type': 'string'}}
>>> v = Validator(schema)
```

# 2. PASS A DOCUMENT TO THE VALIDATE() METHOD

```
>>> document = {'name': 'john doe'}
>>> v.validate(document)
True
```

# ALL TOGETHER NOW

```
>>> from cerberus import Validator
>>> v = Validator()
>>> schema = {'name': {'type': 'string'}}
>>> document = {'name': 'john doe’}
>>> v.validate(document, schema)
True
```

# ‘ERRORS’ RETURNS HUMAN-READABLE ERRORS

```
>>> schema = {
...     'name': {'type': 'string'},
...     'age': {'type': 'integer', 'min': 10}
... }

>>> document = {'name': 'Little Joe', 'age': 5}
>>> v.validate(document, schema)
False

>>> v.errors
{'age': [‘min value is 10’]}
```

# UNKNOWN FIELDS NOT ALLOWED BY DEFAULT

```
>>> v.validate({'name': 'john', 'sex': 'M'})
False

>>> v.errors
{'sex': [‘unknown field’]}
```

# BUT YOU CAN SET ‘ALLOW_UNKNOWN’ TO ALLOW FOR… THE UNKNOWN

```
>>> v.allow_unknown = True
>>> v.validate({'name': 'john', 'sex': 'M'})
True
```

# OPTIONALLY YOU CAN SET A VALIDATION SCHEMA FOR UNKNOWN FIELDS

```
>>> v.schema = {}
>>> v.allow_unknown = {'type': 'string'}
>>> v.validate({'an_unknown_field': 'john'})
True

>>> v.validate({'an_unknown_field': 1})
False

>>> v.errors
{'an_unknown_field': 'must be of string type'}
```

# 'ALLOW_UNKNOWN' CAN BE USED IN NESTED MAPPINGS

```
>>> v = Validator()
>>> v.allow_unknown
False

>>> v.schema = {
...     'name': {'type': 'string'},
...     'a_dict': {
...         'type': 'dict',
...         'allow_unknown': True,
...         'schema': {
...             'address': {'type': 'string'}
...         }
...     }
... }
```

OVERRIDES THE BEHAVIOUR, JUST FOR 'A_DICT'

# ‘ALLOW_UNKNOWN’ CAN BE USED IN NESTED MAPPINGS

```
>>> v.validate({'name': 'john',
...             'a_dict': {'an_unknown_field': 'is allowed'}})
True

>>> v.validate({'name': 'john',
...             'a_dict':{'an_unknown_field': 'is allowed’},
...             'but_this': 'is not allowed'})
False

>>> v.errors
{‘but_this’: ['unknown field’]}
```

# VALIDATED() RETURNS THE VALIDATED DOCUMENT

```
>>> v = Validator(schema)
>>> valid_documents = [
...     valid for valid in [
...       v.validated(doc) for doc in documents]
...     if valid is not None]
```

# NORMALIZED() RETURNS A NORMALIZED COPY (NO VALIDATION)

```
>>> schema = {'amount': {'coerce': int}}
>>> document = {'model': 'consumerism', 'amount': '1'}

>>> normalized_document = v.normalized(document,
...    schema)

>>> type(normalized_document['amount'])
<type 'int'>
```

# USING YAML TO DEFINE A VALIDATION SCHEMA

```
>>> import yaml
>>> schema_text = '''
... name:
...   type: string
... age:
...   type: integer
...   min: 10
... '''

>>> schema = yaml.load(schema_text)
>>> document = {'name': 'Little Joe', 'age': 5}
>>> v.validate(document, schema)
False

>>> v.errors
{'age': [‘min value is 10’]}
```

CERBERUS

# VALIDATION RULES

# ‘TYPE’ CHECKING

OR: BOOLEAN, DATE, DATETIME, MAPPING/DICT, FLOAT, INTEGER, SEQUENCE/LIST, NUMBER, SET, STRING, BINARY

```
>>> v.schema = {'quotes': {'type': 'string'}}
>>> v.validate({'quotes': 1})
False
```

# 'TYPE' CHECKING

IF A LIST, ALL LISTED TYPES ARE ALLOWED

```
>>> v.schema = {
...     'quotes': {'type':['string','list']}}

>>> v.validate({'quotes': 'Hello world!'})
True
>>> v.validate({'quotes':
...     ['Do not disturb my circles!', 'Heureka!']})
True
```

# REGEX VALIDATION

```
>>> schema = {'email': {
...   'type': 'string',
...   'regex': '^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]…'}}

>>> document = {'email': 'john@example.com'}
>>> v.validate(document, schema)
True

>>> document = {'email': 'john_at_example_dot_com'}
>>> v.validate(document, schema)
False

>>> v.errors
{'email': ["value does not match regex '^[a-z…'"]}
```

# ALLOWED VALUES (STRING, LIST, INT)

```
>>> v.schema = {
...     'role': {'type': 'list',
...     'allowed': ['agent', 'client', 'supplier']}}

>>> v.validate({'role': ['agent', 'supplier']})
True
>>> v.validate({'role': ['intern']})
False
>>> v.errors
{'role': ["unallowed values [‘intern']"]}
```

# DEPENDENCIES

```
>>> schema = {'field1': {'required': False},
...           'field2': {'required': False,
...                      'dependencies': ['field1']}}

>>> document = {'field1': 7}
>>> v.validate(document, schema)
True

>>> document = {'field2': 7}
>>> v.validate(document, schema)
False

>>> v.errors
{'field2': ["field 'field1' is required"]}
```

# DEPENDENCIES, WITH RESTRICTED VALUES

```
>>> schema = {'field1': {'required': False},
...           'field2': {'required': True,
...                      'dependencies': {'field1': ['one', 'two']}}}

>>> document = {'field1': 'one', 'field2': 7}
>>> v.validate(document, schema)
True
>>> document = {'field1': 'three', 'field2': 7}
>>> v.validate(document, schema)
False
>>> v.errors
{'field2':
  ["depends on these values: {'field1': ['one', ‘two']}"]}
```

# EXCLUDES

```
>>> v.schema = {
...     'first': {'type': 'dict',
...               'excludes': 'second'},
...     'second': {'type': 'dict',
...                'excludes': 'first'}}

>>> v.validate({'first': {}, 'second': {}})
False
>>> v.validate({'first': {}})
True
>>> v.validate({'second': {}})
True
>>> v.validate({})
True
```

# EXCLUDES, WITH EXCLUSIVE OR

```
>>> v.schema = {
...     'first': {'type': 'dict',
...       'excludes': 'second', 'required': True},
...     'that_field': {'type': 'dict',
...       'excludes': 'first', 'required': True}}

>>> v.validate({'first': {}, 'second': {}})
False
>>> v.validate({'first': {}})
True
>>> v.validate({'second': {}})
True
>>> v.validate({})
False
```

# MULTIPLE FIELD EXCLUDE

```
>>> v.schema = {
...     'first': {'type': 'dict',
...               'excludes': ['second', 'third']},
...     'second': {'type': 'dict',
...                'excludes': 'first'},
...     'three': {'type': 'dict'}}

>>> v.validate({'first': {}, 'third': {}})
False
```

# SUBDOCUMENT VALIDATION

```
>>> v.schema = {
...     'a_dict': {
...         'type': 'dict',
...         'schema': {
...             'address': {'type': 'string'},
...             'city': {'type': 'string',
...                      'required': True}}}}

>>> v.validate({'a_dict':
...     {'address': 'my address', 'city': 'my town'}})
True
```

# LISTS AND OTHER SEQUENCES OF ARBITRARY VALUES

```
>>> schema = {
...     'a_list': {
...         'type': 'list',
...         'schema': {'type': 'integer'}}}

>>> document = {'a_list': [3, 4, 5]}
>>> v.validate(document, schema)
True
```

# SUBDOCUMENTS LIST

```
>>> v.schema = {'rows':
...     {'type': 'list',
...      'schema': {
...                 'type': 'dict',
...                 'schema': {
...                    'sku': {'type': 'string'},
...                    'price': {'type': 'integer'}}}}}

>>> document = {
...     'rows': [{'sku': 'KT123', 'price': 100}]}
>>> v.validate(document)
True
```

# ALLOF / ANYOF / NONEOF / ONEOF

```
>>> schema = {'prop':
...    {'type': 'number',
...     'anyof':
...       [{'min': 0, 'max': 10}, {'min': 100, 'max': 110}]}}

>>> document = {'prop': 5}
>>> v.validate(document, schema)
True
>>> document = {'prop': 105}
>>> v.validate(document, schema)
True
```

# ALLOF / ANYOF / NONEOF / ONEOF

```
>>> schema = {'prop':
...    {'type': 'number',
...     'anyof':
...       [{'min': 0, 'max': 10}, {'min': 100, 'max': 110}]}}

>>> document = {'prop': 55}
>>> v.validate(document, schema)
False
```

# *OF-RULES MAGIC TYPESAVER

```
{'foo': {'anyof_type': ['string', 'integer']}}

# is equivalent to:

{'foo': {
...     'anyof': [
...         {'type': 'string'},
...         {'type': ‘integer'}
...     ]}}
```

# *OF-RULES ALLOW VALIDATION AGAINST MULTIPLE SCHEMAS

```
>>> schemas = [{'department': {'required': True, 'regex': '^IT$'},
...            'phone': {'nullable': True}},
...            {'department': {'required': True},
...             'phone': {'required': True}}]

>>> employee_vldtr = Validator({'employee': {'oneof_schema': schemas,
...                                          'type': 'dict'}},
...                            allow_unknown=True)

>>> invalid_employees_phones = []
>>> for employee in employees:
...   if not employee_vldtr.validate(employee):
...     invalid_employees_phones.append(employee)
```

# ALLOWS TO ‘MAGICALLY’ VALIDATE AGAINST MULTIPLE SCHEMAS

```
>>> schemas = [{'department': {'required': True, 'regex': '^IT$'},
...             'phone': {'nullable': True}},
...            {'department': {'required': True},
...             'phone': {'required': True}}]

>>> employee_vldtr = Validator({'employee': {'oneof_schema': schemas,
...                                          'type': 'dict'}},
...                            allow_unknown=True)

>>> invalid_employees_phones = []
>>> for employee in employees:
...   if not employee_vldtr.validate(employee):
...     invalid_employees_phones.append(employee)
```

# STANDARD RULES SET ALSO INCLUDES

- required
- nullable
- forbidden
- readonly
- empty *(wether a string can be empty)*
- min, max *(arbitrary types)*
- minlength, maxlength *(iterable)*
- valueschema *(validation schema for all values of a dict/mapping)*
- keyschema *(validates the keys of a dict/mapping)*
- *and (a lot) more.*

CERBERUS

# SCHEMA REGISTRIES

# SCHEMA REGISTRY

```
>>> from cerberus import schema_registry

>>> schema_registry.add(
...     'user', {'uid': {'min': 1000, 'max': 0xffff}})

>>> schema = {
...     'sender': {'schema': 'user',
...                'allow_unknown': True},
...     'receiver': {'schema': 'user',
...                  'allow_unknown': True}}
```

# RULES SET REGISTRY

```
>>> from cerberus import rules_set_registry

>>> rules_set_registry.extend((
...     ('boolean', {'type': 'boolean'}),
...     ('booleans', {'valueschema': 'boolean'})))

>>> schema = {'foo': 'booleans'}
```

# RULES SET REGISTRY

```
>>> from cerberus import rules_set_registry

>>> rules_set_registry.extend((
...     ('boolean', {'type': 'boolean'}),
...     ('booleans', {'valueschema': 'boolean'})))

>>> schema = {'foo': 'booleans'}
```

CERBERUS

# NORMALIZATION RULES

# FIELD RENAMING

```
>>> v = Validator({'foo': {'rename': 'bar'}})
>>> v.normalized({'foo': 0})
{'bar': 0}
```

# FIELD RENAMING, WITH CUSTOM HANDLER

```
>>> v = Validator(
...         {},
...         allow_unknown={'rename_handler': int}
... )

>>> v.normalized({'0': 'foo'})
{0: 'foo'}
```

# RENAMING, WITH PIPELINE

```
>>> even_digits = lambda x: '0' + x if len(x) % 2 else x

>>> v = Validator(
...     {},
...     allow_unknown={
...         'rename_handler': [str, even_digits]})

>>> v.normalized({1: 'foo'})
{'01': 'foo'}
```

APPLY STR() FIRST, THEN EVEN_DIGITS()

# PURGING OF UNKNOWN FIELDS

```
>>> v = Validator({
...         'foo': {'type': 'string'}}, purge_unknown=True)

>>> v.normalized({'bar': 'not really', 'foo': 'yup'})
{'foo': 'yup'}
```

# DEFAULT VALUES

```
>>> v.schema = {
...     'amount': {'type': 'integer'},
...     'kind': {'type': 'string', 'default': 'purchase'}}

>>> v.normalized({'amount': 1})
{'amount': 1, 'kind': 'purchase'}

>>> v.normalized({'amount': 1, 'kind': None})
{'amount': 1, 'kind': 'purchase'}

>>> v.normalized({'amount': 1, 'kind': 'other'})
{'amount': 1, 'kind': 'other'}
```

# DEFAULT VALUES, WITH CUSTOM SETTER

```
>>> v.schema = {
...     'a': {'type': 'integer', 'required': True},
...     'b': {
...         'type': 'integer',
...         'default_setter': lambda doc: doc['a'] + 1}}

>>> v.normalized({'a': 1})
{'a': 1, 'b': 2}
```

# VALUE COERCION

```
>>> to_bool = lambda v: v.lower() in ['true', '1']
>>> v.schema = {'flag': {'type': 'boolean', 'coerce': to_bool}}

>>> v.validate({'flag': 'true'})
True

>>> v.document
{'flag': True}
```

CERBERUS

# EXTENDING

# CURSTOM RULES

```
>>> schema = {'amount':
...   {'isodd': True, 'type': 'integer'}}

>>> class MyValidator(Validator):
...   def _validate_isodd(self, isodd, field, value):
...     if isodd and not bool(value & 1):
...       self._error(field, "Must be an odd number")

>>> v = MyValidator(schema)
>>> v.validate({'amount': 10})
False
>>> v.errors
{'amount': 'Must be an odd number'}
>>> v.validate({'amount': 9})
True
```

# CUSTOM DATA TYPES

```
>>> schema = {'id': {'type': 'objectid'}}

>>> class MyValidator(Validator):
...  def _validate_type_objectid(self, field, value):
...    if not re.match('[a-f0-9]{24}', value):
...      self._error(field, errors.BAD_TYPE)

>>> v = MyValidator(schema)
>>> v.validate({'id': 'dont think so'})
False

>>> v.validate({'id': '56a245e738345b1ca2b25164'})
True
```

# CUSTOM VALIDATION FUNCTIONS

```
>>> def oddity(field, value, error):
...     if not value & 1:
...         error(field, "Must be an odd number")

>>> schema = {'amount': {'validator': oddity}}
>>> v = Validator(schema)
>>> v.validate({'amount': 10})
False

>>> v.errors
{'amount': 'Must be an odd number'}

>>> v.validate({'amount': 9})
True
```

# CUSTOM COERCES

```
class MyNormalizer(Validator):
    def __init__(self, multiplier, *args, **kwargs):
        super(MyNormalizer, self).__init__(*args, **kwargs)
        self.multiplier = multiplier

    def _normalize_coerce_multiply(self, value):
        try:
            return value * self.multiplier
        except Exception as e:
            self._error(field, errors.COERCION_FAILED, e)

>>> schema = {'foo': {'coerce': 'multiply'}}
>>> document = {'foo': 2}
>>> MyNormalizer(2).normalized(document, schema)
{'foo': 4}
```

# CUSTOM DEFAULT SETTERS

```
from datetime import datetime

class MyNormalizer(Validator):
    def _normalize_default_setter_utcnow(self, document):
        return datetime.utcnow()

>>> schema = {'date': {'type': 'datetime',
...                    'default_setter': 'utcnow'}}

>>> MyNormalizer().normalized({}, schema)
{'date': datetime.datetime(...)}
```

# UNORTHODOX USE: VALIDATING USER OBJECTS (SIMPLE)

```
>>> class ObjectValidator(Validator):
...     def validate_object(self, obj):
...         return self.validate(obj.__dict__)
...

>>> v = ObjectValidator(schema)
>>> v.validate_object(person)
False

>>> v.errors
{'age': 'min value is 0'}

>>> person.age = 44
>>> v.validate_object(person)
True
```

http://nicolaiarocci.com/validating-user-objects-cerberus/

CERBERUS

# API

# LEVERAGE THE CERBERUS API

- Validator. Normalizes and/or validates any mapping against a validation-schema.
- ErrorHandlers. BaseErrorHandler is the base class for all error handlers. Subclasses are identified as error-handlers with an instance-test.
- ValidationError. A simple class to store and query basic error information.
- ErrorList. A list for ValidationError instances that can be queried with the in keyword for a particular error code.
- ErrorTree. Base class for DocumentErrorTree and SchemaErrorTree.
- validator_factory. Dynamically create a Validator subclass.

# TESTIMONIALS

# WE LOVE IT AT GITPRIME.
# THE WORLD NEEDS MORE CERBERUS!

Vincent Driessen
CTO, GitPrime
@nvie

# THANKS!

Cerberus

python-cerberus.org

Nicola Iarocci

@nicolaiarocci / nicolaiarocci.com / me@nicolaiarocci.com